Unit Testing
Test Driven Development
Continuous Integration

Maximilian Nöthe

Astroparticle Physics, TU Dortmund

ESCAPE
European Science Cluster of Astronomy & Particle physics ESFRI research Infrastructures

Summer School – 2021-06-10

Overview

Warning

Copying commands or code from PDF files is **dangerous**

Copy from the example files in the repository or type by hand.

Typing by hand is best for learning.

Introduction

Automated Software Testing

- → Verifying that a software works as intended is crucial
- → Doing this manually using whatever method you can think of
  - → is very tedious
  - → is errorprone
  - → will result in the tests not being done most of the time
- ⇒ We need automated tests that verify our software
- → Tests fall into three categories
  1. Unit tests
  2. Integration tests
  3. Performance tests

Unit tests

- → Test single “units” of the code in isolation
- → Require modular design of the code base
- → Are the bedrock of any more complicated tests
- → Must be fast and easy to run ⇒ or they would not be run most of the time

Properties of good unit tests

Existence ☺

Correctness The code under test behaves according to requirements / specifications

Completeness The tests cover all required features / use cases

Readability Writing tests for tests would result in infinite recursion
$\Rightarrow$ tests must readable, so they can be easily verified by inspection

Demonstrability Good tests show how your code is meant to be used

Resilience Tests should only fail if what they test breaks

Frameworks

All modern languages have one or more frameworks for tests, a small selection:

Python pytest

C++ Catch2, GoogleTest

Java JUnit

Rust Part of the language

Julia `Test` module in the standard library

Integration tests

- → Test that multiple *units* are working together
- → E. g. testing a whole command line application
- → Can grow arbitrarily large / complicated

Performance tests

- → Unit and integration tests usually only test the correctness of code
- → Performance tests make sure the code fulfills requirements and does not get slower
- → This introduction focuses on unit tests
- → See the profiling lecture for more information on how to actually measure performance

Example python code

We are going to use this simple function as example for our first unit tests:

examples/step1/fibonacci.py

```python
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

pytest

pytest

→ Standard framework for writing unit tests for Python projects
→ Uses the **`assert`** statement for tests
→ Tests fail if an assertion fails or an exception is raised
→ Uses introspection of the assertion to give detailed error messages
→ Automatic test detection using patterns:
  → Modules matching `test_*.py` or `*_test.py`
  → Functions called `test*`
  → Methods named `test*` of classes named `Test*`
→ Docs: `https://pytest.org`

First Unit Test

examples/step1/test_fibonacci1.py

```python
def test_fiboncacci():
    from fibonacci import fibonacci

    assert fibonacci(4) == 3
    assert fibonacci(7) == 13
```

```
$ pytest
========================== test session starts ===========================
platform linux -- Python 3.9.4, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/maxnoe/Uni/escape-school2021/testing/slides/examples/step1
collected 1 item

test_fibonacci1.py .                                                [100%]

=========================== 1 passed in 0.00s ============================
```

A note on imports

examples/step1/test_fibonacci1.py

```python
def test_fiboncacci():
    from fibonacci import fibonacci

    assert fibonacci(4) == 3
    assert fibonacci(7) == 13
```

- Against usual python style, you should import what you test in the test function
- Like this, the test discovery of pytest will also work when the import would fail and the failure is reported as part of the test
- Everything else, like standard library imports or third-party dependencies, is imported normally at the top

Testing Exceptions

Make sure the correct exception is thrown, e. g. for invalid input:

examples/step2/fibonacci.py

```python
def fibonacci(n):
    if n < 0:
        raise ValueError(f'n must be >= 0, got {n}')
    # rest unchanged
```

examples/step2/test_exception.py

```python
import pytest

def test_invalid_values():
    from fibonacci import fibonacci

    with pytest.raises(ValueError):
        fibonacci(-1)
```

The same can be done for warnings using `pytest.warns`

Careful with floating point numbers

Naive, this fails

```python
def test_addition_naive():
    assert 0.1 + 0.2 == 0.3
```

Correct approach, using `pytest.approx`

```python
def test_addition_correct():
    assert 0.1 + 0.2 == pytest.approx(0.3)
```

See `https://0.30000000000000004.com/`

Using numpy testing utitlities

Using numpy

```python
import numpy as np

def test_sin():
    x = np.array([0, np.pi / 2, np.pi])
    np.testing.assert_array_almost_equal(np.sin(x), [0, 1, 0], decimal=15)

def test_poly():
    def f(x):
        return x**2 + 2 * x + 10

    x = np.array([0.0, 1.0, 2.0])
    np.testing.assert_allclose(f(x), [10.0, 13.0, 18.0], rtol=1e-5)
```

See `https://numpy.org/doc/stable/reference/routines.testing.html`

Using astropy quantity support

Using astropy units

```python
import astropy.units as u

def test_time():
    v = 10 * u.m / u.s
    d = 1 * u.km
    assert u.isclose(d / v, 100 * u.s)


def test_many():
    v = 10 * u.m / u.s
    d = [0, 1, 5] * u.km
    assert u.allclose(d / v, [0, 100, 500] * u.s)
```

Fixtures

- → Data and resources used by tests can be injected into tests using "fixtures"
- → Fixtures are provided by functions decorated with `@fixture`
- → Fixtures have a scope ⇒ same object used per session, module, class or function
- → Default is `scope="function"`

```python
import pytest

@pytest.fixture(scope='session')
def some_data():
    return [1, 2, 3]

def test_using_fixture(some_data):
    assert len(some_data) == 3

def test_also_using_fixture(some_data):
    assert some_data[0] == 1
```

Fixtures provided by pytest

pytest provides several builtin fixtures for

- temporary directories `tmp_path` / `tmp_path_factory`
- Testing output to stdout / stderr `capsys`
- Testing logging `caplog`
- Monkeypatching `monkeypatch`

More at `https://docs.pytest.org/en/6.2.x/fixture.html`

capsys – Fixture for testing the standard streams

```python
def greet(name):
    print(f'Hello, {name}!')

def test_prints(capsys):
    # call the function
    greet('Escape School 2021')

    # test that it wrote what we expect to stdout
    captured = capsys.readouterr()
    # .err would be the stderr output
    assert captured.out == 'Hello, Escape School 2021!\n'
```

caplog – Fixture for testing logging

```python
import logging

def do_work():
    log = logging.getLogger('do_work')
    log.info('Doing work')
    log.info('Done')


def test_do_work_logs(caplog):
    with caplog.at_level(logging.INFO):
        do_work()

    assert len(caplog.records) == 2
    for record in caplog.records:
        assert record.levelno == logging.INFO
```

Temporary paths

- For tests that need to create files, use the `tmp_path` fixture
  ⇒ Avoids cluttering and conflicts when running tests multiple times / between tests
- `tmp_path` has scope *function*, so each test gets its own temporary directory
- These directories are not cleaned up after the tests, so you can inspect the results
- If you need a temporary path with a wider scope, add a new fixture using `tmp_path_factory`

Temporary paths

```python
from astropy.table import Table
import numpy as np


def test_to_csv(tmp_path):

    t = Table({'a': [1, 2, 3], 'b': [4, 5, 6]})
    t.write(tmp_path / 'test.csv')

    read = Table.read(tmp_path / 'test.csv')
    assert np.all(read == t)
```

Run the test and checkout
`/tmp/pytest-of-$USER/pytest-current/test_to_csvcurrent`

Fixtures that need a cleanup step

→ Sometimes, resources or data need to be cleaned up after the test have run

→ This can be implemented using a generator fixture that yields the data and cleans up after the yield

```python
@pytest.fixture()
def database_connection():
    connection = database.connect()
    yield connection
    # close after use
    connection.close()

@pytest.fixture()
def database_connection():
    # even better, with a context manager
    with database.connect() as connection:
        yield connection
```

Parametrized Tests and Fixtures

→ Parametrization allows to run the same test on multiple inputs

→ Very useful to reduce code repetition and get clearer messages

A parametrized test

```python
import pytest

n = range(9)
fibs = [0, 1, 1, 2, 3, 5, 8, 13, 21]

@pytest.mark.parametrize('n,expected', zip(n, fibs))
def test_fibonacci(n, expected):
    from fibonacci import fibonacci

    assert fibonacci(n) == expected
```

Conditional tests

Some tests can only be run under specific conditions

→ Tests for features requiring optional dependencies

This test is skipped when numpy is not available

```python
def test_using_numpy():
    np = pytest.importorskip("numpy")
    assert len(np.zeros(5)) == 5
```

→ Tests for specific operating systems or versions

This test is only executed on Windows

```python
@pytest.mark.skipif(sys.platform != 'win32', reason="windows only")
def test_windows():
    assert os.path.exists('C:\\')
```

Expected failures

It sometimes makes sense to implement tests that are expected to fail:

- → Planned but not yet implemented features
- → Known but not yet fixed bugs
- → These tests shouldn’t make your whole test suite fail

This test is expected to fail

```python
import pytest

@pytest.mark.xfail
def test_this_fails():
    import math
    assert math.pi == 3
```

Choosing which tests to run

pytest offers fine-grained control over which tests to run

→ Select a specific test:

```
$ pytest test_module.py::test_name
```

→ Run only tests that failed the last time pytest was run

```
$ pytest --last-failed
```

→ Stop after N failures

```
$ pytest --maxfail=2
```

→ Using matching expressions

```
$ pytest -k "fib"
```

→ Run tests for an installed package

```
$ pytest --pyargs fibonacci
```

Choosing which tests to run – Using markers

→ Define markers in `pyproject.toml`

```toml
[tool.pytest.ini_options]
markers = ["slow"]
```

→ Add the marker to a test

```python
@pytest.mark.slow
def test_slow():
    time.sleep(2)
    assert 1 + 1 == 2
```

→ Run tests using marker expressions

```
$ pytest -m "not slow"
$ pytest -m "slow"
```

Debugging

- Unit tests can be very useful for debugging
- E. g. Write a new test that triggers the bug → investigate → make it pass
- pytest allows you to jump into pdb when a test fails:

```
$ pytest --pdb
```

- or any other debugger, e. g. ipython’s:

```
$ pytest --pdb --pdbcls=IPython.terminal.debugger:TerminalPdb
```

Test Coverage

Test Coverage

→ Test coverage is a metric measuring how much of the code is tested:

$$\text{coverage} = \frac{\text{Lines of code executed during tests}}{\text{Total lines of code}}$$

→ Can be helpful to find parts of code that are not tested (enough).

→ Especially useful in CI system to check that new / changed code is tested

→ One more badge 😉! codecov 90%

Create a coverage report

→ Print coverage after test suite

```
$ pytest --cov=fibonacci
```

→ Create a detailed report in html format

```
$ pytest --cov=fibonacci --cov-report=html
```

→ Serve the report using python's built-in http server and explore in the browser:

```
$ python -m http.server -d htmlcov
```

Limitations of line coverage

Executed number of lines of code are not a perfect measure.

```python
if some_condition is True:
    do_stuff()
do_other_stuff()
```

Limitations of line coverage

Executed number of lines of code are not a perfect measure.

```python
if some_condition is True:
    do_stuff()
do_other_stuff()
```

When during the tests `some_condition is True`, this code will have 100 % coverage.
But what about `some_condition is not True`?

Limitations of line coverage

Executed number of lines of code are not a perfect measure.

```python
if some_condition is True:
    do_stuff()
do_other_stuff()
```

When during the tests `some_condition is True`, this code will have 100 % coverage.
But what about `some_condition is not True`?

```python
result = scipy.optimize.minimize(likelihood, ...)
```

Limitations of line coverage

Executed number of lines of code are not a perfect measure.

```python
if some_condition is True:
    do_stuff()
do_other_stuff()
```

When during the tests `some_condition is True`, this code will have 100 % coverage. But what about `some_condition is not True`?

```python
result = scipy.optimize.minimize(likelihood, ...)
```

Calling functions from other packages can have arbitrarily many branches

Run pytest with branch coverage

```
$ pytest --cov=fibonacci --cov-report=html --cov-branch
```

Mocking / Monkeypatching

Mocking / Monkeypatching

- Sometimes, classes or functions have behaviour that prevents unit testing
- E. g. code that speaks to specific hardware, makes web requests, relies on system time …
- This is usually a sign of insufficient modularization / separation of concerns
- A solution can be mocking or monkeypatching, if it is not possible to improve the actual code

Mocking / Monkeypatching

```python
import requests
import json

def is_server_healthy():
    ret = requests.get('https://example.org/healthcheck')
    ret.raise_for_status()
    return ret.json()['healthy']

def test_healthy(monkeypatch):
    with monkeypatch.context() as m:
        def get(url):
            resp = requests.Response()
            resp.url = url
            resp.status_code = 200
            resp._content = json.dumps({'healthy': True}).encode('utf-8')
            return resp

        m.setattr(requests, 'get', get)
        assert is_server_healthy()
```

Test Driven Development

Test Driven Development (TDD)

- → Test Driven Development is a powerful paradigm
- → Essentially, to implement a new feature
  1. Write the tests before any implementation code
  2. Run the tests → they should all fail
  3. Write the minimal implementation that makes the test pass
  4. All tests should now pass
  5. Cleanup, refactor, tests must keep passing

Test Drive Development

- → TDD forces you to think about requirements and API *before* writing the actual code
- → Especially usefull when
  - → you have clear specifications
  - → investigating / trying to fix a bug
  - → working on a new greenfield project
- → Not so easy to use when
  - → working in a large, historic codebase without good test coverage
  - → doing explorative work

Doctests

Doctests

- Examples are an important part of every documentations

```python
def fibonacci(n):
    '''Calculate the nth fibonacci number using recursion

    Examples
    --------
    >>> fibonacci(7)
    13
    '''
```

- Important to verify that the examples stay up to date and are correct
- Solution: run all the examples and check the expected output

```
$ pytest --doctest-glob="*.rst" --doctest-modules
```

- This will find and execute code blocks in docstrings and documentation rst-files
- Checks the output is what is expected

Continuous Integration

Continuous Integration

- CI systems run the build, unit tests and code quality checks automatically
- They should run for each push event / opened pull request
- All architectures, operating systems and versions you support should be tested
- Many tools provide detailed reporting that help with code reviews
- You should require the passing of the CI system for pull requests
- All providers also support encrypted secrets for confidential information
  - Automatize upload of releases
  - Access private data needed for tests
  - …

Providers

GitHub Actions `https://docs.github.com/en/actions`

- → Free for all public GitHub repositories
- → Linux, Mac and Windows builds
- → Support for custom runners that you can self-host
- → Recommended for projects on GitHub

GitLab CI `https://docs.gitlab.com/ee/ci/`

- → 400 minutes of build time per month with gitLab.com Free
- → Support for custom runners that you can self-host
- → Also available for self-hosted GitLabs (you have to setup at least one runner)

Jenkins `https://www.jenkins.io/`

- → Open-Source CI platform you can self-host

Many more, including Travis CI, AppVeyor, circle**ci**, Azure Pipelines

Defining a CI Workflow

- → All of these providers use different configuration files to specify a workload
- → Despite these differences, the idea is always the same
  - → Define environments, operating systems, software versions
  - → Get the code
  - → Install dependencies
  - → Compile / build / install the software
  - → Run the tests
  - → Upload results

A minimal github actions example

→ See the demo project at `https://github.com/maxnoe/pyfibonacci`

Some CI helpful Tools

`https://codecov.io/` Explore coverage reports and adds coverage checks to pull requests

`https://reviewnb.com/` Makes reviewing jupyter notebooks easier / possible

`https://codacy.com/` Static code analysis, e. g. style linting