@OxfordRSE
rse@cs.ox.ac.uk
www.rse.ox.ac.uk

UNIVERSITY OF OXFORD

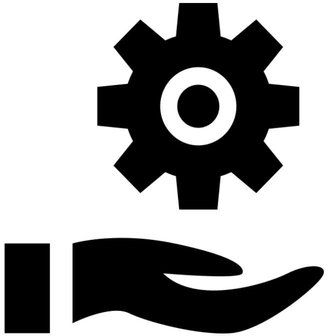
OxRSE

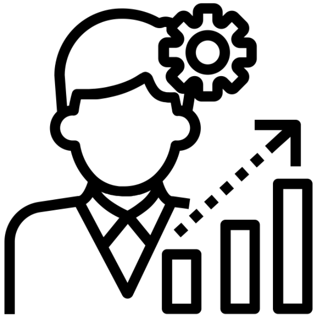
Introduction to Unit Testing with Python and GitHub

Fergus Cooper
Oxford Research Software Engineering

Who are we?

Enable your research by **developing bespoke software**





Facilitate your research by helping you to **improve your software** and **software engineering practice**

Why should we test software?

Being able to demonstrate that a process generates the right results is important in **every** field of research.

- Does the code we develop work the way it should do?
- Can we (and others) **verify** this correctness?
- To what extent are we confident in the results that appear in publications?

If we are unable to demonstrate that our software fulfils these criteria, why would anyone use it?

But I don’t write code with errors in!

If each line we write has a 99% chance of being right, then a 70-line program is more likely than not to be wrong.

We need to do better than that, which means we **need to test** our software to catch these mistakes.

Even if you write some perfect code the first time around, you (or someone else) will later modify it: does it **still** behave the way you expect?

Manual testing

We can often look at the output of code, for example:

- a plot showing convergence with timestep
- running an analysis pipeline on an existing dataset with known output
- visually inspecting that a simulation “does the right thing”

Manual testing

These are important ways of testing during development, but have drawbacks:

- they only test a subset of expected behaviour
- you have to run them manually

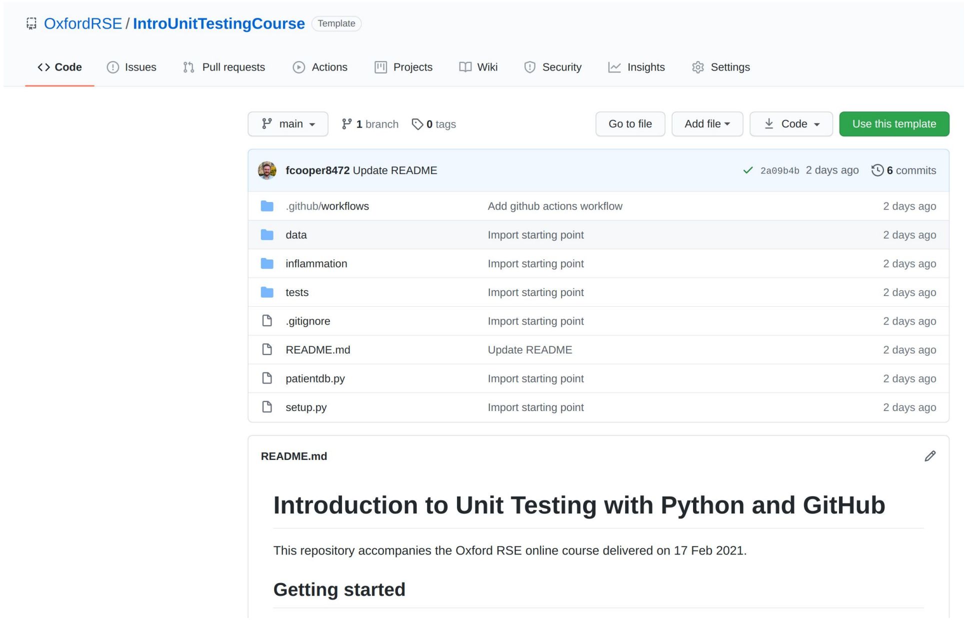
Automated testing

Since computers are very good and efficient at automating repetitive tasks, we should take advantage of this wherever possible.

- **Unit tests** are tests for fairly small and specific units of functionality
- **Functional** or **integration tests** work at a higher level, and test functional paths through your code
- **Regression tests** make sure that your program's output hasn't changed

For the purposes of this course, we'll focus on unit tests.

An example dataset and application (link)


OxfordRSE / IntroUnitTestingCourse Template

fcooper8472 Update README ✓ 2a09b4b 2 days ago 6 commits

.github/workflows	Add github actions workflow	2 days ago
data	Import starting point	2 days ago
inflammation	Import starting point	2 days ago
tests	Import starting point	2 days ago
.gitignore	Import starting point	2 days ago
README.md	Update README	2 days ago
patientdb.py	Import starting point	2 days ago
setup.py	Import starting point	2 days ago

README.md

Introduction to Unit Testing with Python and GitHub

This repository accompanies the Oxford RSE online course delivered on 17 Feb 2021.

Getting started

Follow along exercise

First attempt

As an example, we'll start by testing our code directly using `assert`.

Create a file "sandbox.py" and paste the following code:

```
from inflammation.models import daily_mean
import numpy as np

assert np.array_equal(np.array([0, 0]), daily_mean(np.array([[0, 0], [0, 0]])))
assert np.array_equal(np.array([3, 4]), daily_mean(np.array([[1, 2], [3, 4], [5, 6]])))
```

First attempt

So far so good. But now let’s add a failing line at the front:

```python
from inflammation.models import daily_mean
import numpy as np

assert np.array_equal(np.array([2, 0]), daily_mean(np.array([[2, 0], [4, 0]])))
assert np.array_equal(np.array([0, 0]), daily_mean(np.array([[0, 0], [0, 0]])))
assert np.array_equal(np.array([3, 4]), daily_mean(np.array([[1, 2], [3, 4], [5, 6]])))
```

```
Traceback (most recent call last):
  File "/home/fergus/GitRepos/fcooper8472/UnitTestingCourse/sandbox.py", line 4, in <module>
    assert np.array_equal(np.array([2, 0]), daily_mean(np.array([[2, 0], [4, 0]])))
AssertionError
```

What are the problems with this?

Using a testing framework

Most people don’t enjoy writing tests, so if we want them to actually do it, it must be easy to:

- Add or change tests,
- Understand the tests that have already been written,
- Run those tests, and
- Understand those tests’ results

Using a testing framework

There are many unit testing frameworks in different languages

- Python: **pytest**, unittest, nose2
- C++: Catch2, GoogleTest, ...
- Java: JUnit
- Fortran: FRUIT
- ...

Testing daily_mean

Let’s add some tests for our library function, `daily_mean`:

```python
def test_daily_mean_zeros():
    """Test that mean function works for an array of zeros."""
    from inflammation.models import daily_mean

    test_array = np.array([[0, 0],
                           [0, 0],
                           [0, 0]])

    # Need to use Numpy testing functions to compare arrays
    npt.assert_array_equal(np.array([0, 0]), daily_mean(test_array))
```

Testing daily_mean

Let's add some tests for our library function, `daily_mean`:

```python
def test_daily_mean_integers():
    """Test that mean function works for an array of positive integers."""
    from inflammation.models import daily_mean

    test_array = np.array([[1, 2],
                           [3, 4],
                           [5, 6]])

    # Need to use Numpy testing functions to compare arrays
    npt.assert_array_equal(np.array([3, 4]), daily_mean(test_array))
```

Run: `pytest tests/test_models.py`

Exercise

Add tests for the library functions `daily_min` and `daily_max`.

What about testing for errors?

Some functions do not support all inputs.

In compiled languages you often get help from the type system (but not so much in Python).

Even then, a function might expect, say, a strictly positive argument and throw an error of some kind if it does not.

How can we test this?

All unit testing frameworks have this functionality built in.

What about testing for errors?

```
def test_daily_min_string():
    """Test for TypeError when passing strings"""
    from inflammation.models import daily_min
    from pytest import raises

    with raises(TypeError):
        daily_min([['Cannot', 'min'], ['string', 'arguments']])
```

Why should we test invalid input data?

Testing the behaviour of inputs, both valid and invalid, is a good idea and is known as **data validation**.

Even if you are developing command-line software that cannot be exploited by malicious data entry, testing behaviour against invalid inputs prevents generation of erroneous results that could lead to serious misinterpretation.

It’s generally best not to assume your user’s inputs will always be rational.

Exercise

Add some tests for errors with invalid input.

Parameterise tests to run over many test cases

We're starting to build up a number of tests that test the same function, but just have different parameters.

Instead of writing a separate function for each different test, we can **parameterise** the tests with multiple test inputs.

For example, we could rewrite the test_daily_mean_zeros() and test_daily_mean_integers() into a single test function:

Parameterise tests to run over many test cases

```
@pytest.mark.parametrize(
    "test, expected",
    [
        ([[0, 0], [0, 0], [0, 0]], [0, 0]),
        ([[1, 2], [3, 4], [5, 6]], [3, 4]),
    ])
def test_daily_mean(test, expected):
    """Test mean function works for array of zeroes and positive integers."""
    from inflammation.models import daily_mean
    npt.assert_array_equal(np.array(expected), daily_mean(np.array(test)))
```

Why is this preferable?

Exercise

Re-write your tests for mean, min and max to be parameterised.

Commit your changes to test_models.py

First, check that running `pytest` run as expected.

Then:

```
git add tests/test_models.py

git commit -m “Added more tests”

git push
```

Recap where we’re up to

We are making use of a unit testing framework (pytest).

We have written tests that verify normal functional behaviour of three specific units.

We have tested that some common cases of function misuse fail in the way we expect them to fail.

We have parameterised tests to cut down on code duplication.

What are the problems we still have with our current setup?

Running tests automatically

It’s critical to not rely on running the tests ourselves.

We’re simply too unreliable to run the tests every time we make changes.

Fortunately, it’s very easy!

>> Demo of using GitHub actions

Exercise

Add the GitHub status badge to your README.md file.

Next steps

Write some unit tests!

Once you have set up the infrastructure it's very easy, especially for the majority of testing.

There is complexity: how to test random functions, for instance.

Don't fall into the trap of thinking "I could be using this time to write more features". Code worthless if you aren't certain it's correct!

Encourage others in your research group to start testing their code.

That’s all for today

Any questions?