Using Pytest in a Research Project

Leonard Sasse & Synchon Mandal

May 3, 2024

Section 1

Introduction: What’s the problem?

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."
— *Edsger W. Dijkstra, "The Humble Programmer" (1972)*

Imagine you write a simple function

```python
def add(a, b):
    """Add two values."""
    return a + b
```

How do we know that this is correct?

Testing our new function

We can go to the REPL and run the code! For example, we know that $2 + 2 = 4$:

```
python3
```

and then:

```
>>> add(2, 2)
4
```

Seems like our function is doing the right thing!

Testing a script

We can test whether a script is running and producing the correct output:

```python
def add(a, b):
    return a + b

def mul(a, b):
    return a * b

def predict(x):
    intercept = 5
    coef = 0.7
    return add(
        mul(coef, x),
        intercept
    )
```

```python
def main():
    for x in range(10):
        y = predict(x)
        print(f"f({x}) = {y}")

if __name__ == "__main__":
    main()
```

Run the script and inspect the output

```
python3 model.py
```

The results will be displayed on the terminal and we can check whether the results are correct or plausible by hand.

```
f(0) = 5.0
f(1) = 5.7
f(2) = 6.4
f(3) = 7.1
f(4) = 7.8
f(5) = 8.5
f(6) = 9.2
f(7) = 9.899999999999999
f(8) = 10.6
f(9) = 11.3
```

What is the problem with this approach?

What is the problem with this approach

- As projects grow, manually re-testing every script is tedious and error prone.
- We forget manual tests that we have done in the past and don't do it again
- We are testing a very limited set of inputs
  - Bugs may only appear for certain edge cases but we are not really searching for those

Section 2

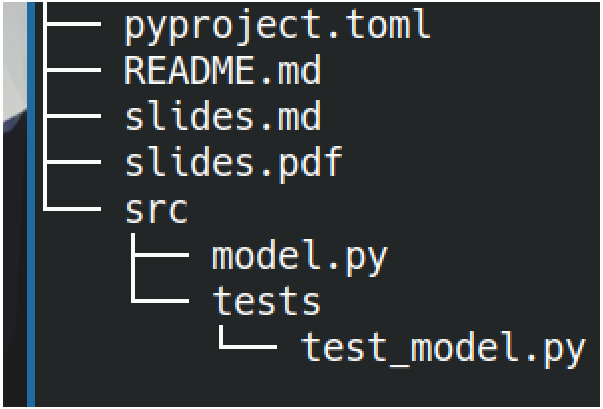
We can automate the testing

How to write tests

1. Set up any needed data or state.
2. Run the code you want to test.
3. Assert the results are what you expect.

Basic project set up

- This is only a folder structure to showcase a basic, minimal pytest example.
- This is NOT an example of how a project SHOULD be organised.
- It is purely intended to demonstrate how pytest can be made functional.



```
├── pyproject.toml
├── README.md
├── slides.md
├── slides.pdf
└── src
    ├── model.py
    └── tests
        └── test_model.py
```

Figure 1: Demo tree

Doing a python project? Make a **pyproject.toml**!

```
[tool.pytest.ini_options]
minversion = "7.0"
testpaths = "src/tests"
log_cli_level = "INFO"
xfail_strict = true
addopts = [
    "-ra",
    "--strict-config",
    "--strict-markers",
    "-vvv",
]
```

Run from the **root** of your **project**:

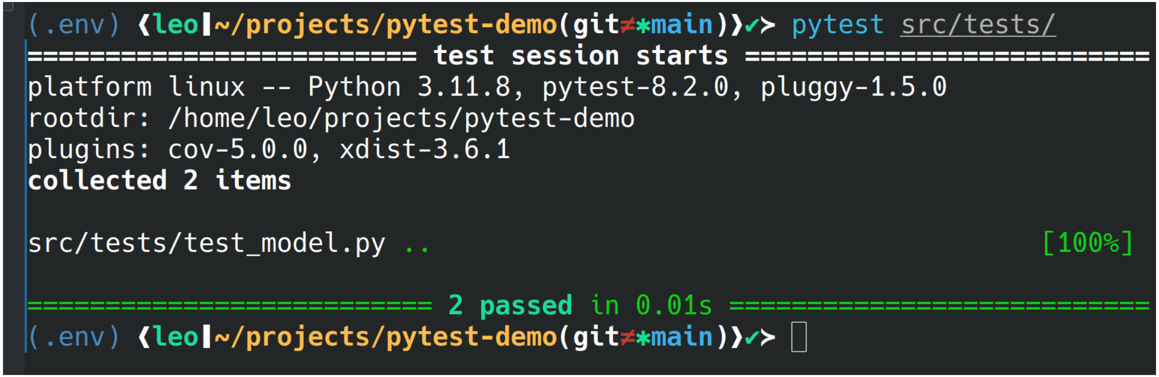
We can run these tests all at once using `pytest` which will tell us if they pass or fail.

```
pip install pytest
pytest
```

```python
def add(a, b):
    return a + b

def mul(a, b):
    return a * b
```

```python
def test_add():
    result = add(2, 2)
    assert result == 4

def test_mul():
    result = mul(2, 2)
    assert result == 4
```

Pass



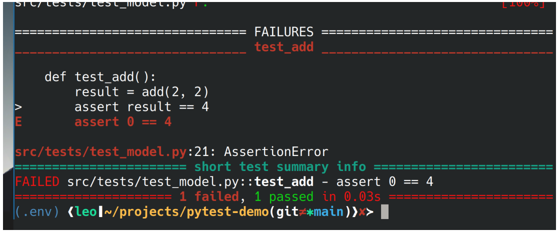
```
(.env) ❰leo❙~/projects/pytest-demo(git≠*main)❱✓> pytest src/tests/
========================= test session starts ==========================
platform linux -- Python 3.11.8, pytest-8.2.0, pluggy-1.5.0
rootdir: /home/leo/projects/pytest-demo
plugins: cov-5.0.0, xdist-3.6.1
collected 2 items

src/tests/test_model.py ..                                        [100%]

========================== 2 passed in 0.01s ===========================
(.env) ❰leo❙~/projects/pytest-demo(git≠*main)❱✓>
```

Figure 2: PASSED

Introduce a bug:

```python
def add(a, b):
    return a - b
```

```
pytest
```



```
src/tests/test_model.py F.                                             [100%]

============================== FAILURES ===============================
______________________________ test_add _______________________________

    def test_add():
        result = add(2, 2)
>       assert result == 4
E       assert 0 == 4

src/tests/test_model.py:21: AssertionError
======================= short test summary info =======================
FAILED src/tests/test_model.py::test_add - assert 0 == 4
===================== 1 failed, 1 passed in 0.03s =====================
(.env) ❬leo❙~/projects/pytest-demo(git≠*main)❭x> 
```

Figure 3: FAIL

Section 3

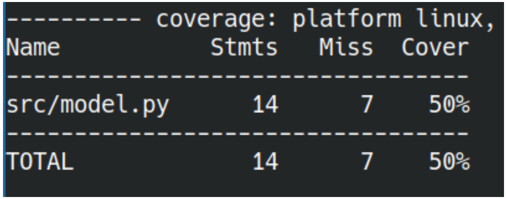
Useful pytest plugins

We can get a report on line coverage using the pytest-cov plugin

```
pip install pytest-cov
```

From the repository root run:

```
pytest --cov=src
```



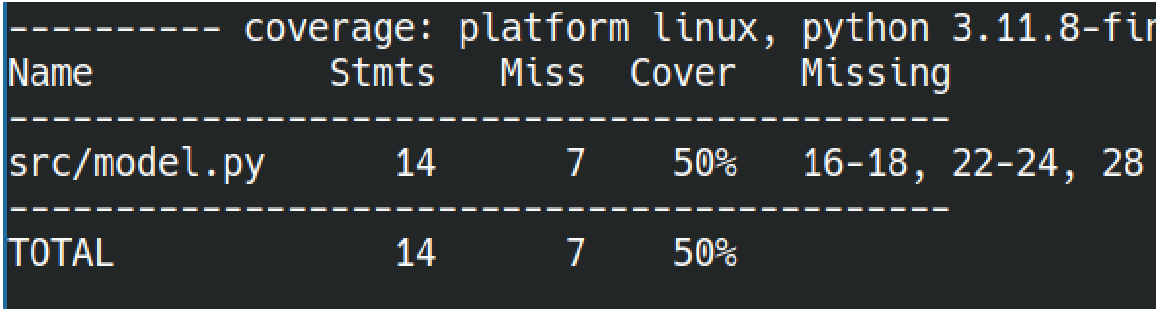
```
---------- coverage: platform linux,
Name                 Stmts   Miss  Cover
----------------------------------------
src/model.py            14      7    50%
----------------------------------------
TOTAL                   14      7    50%
```

Figure 4: coverage

We can also see which lines in our code are not executed in tests:

From the repository root run:

```
pytest --cov=src --cov-report term-missing
```



```
---------- coverage: platform linux, python 3.11.8-fi
Name               Stmts   Miss  Cover   Missing
--------------------------------------------
src/model.py          14      7    50%   16-18, 22-24, 28
--------------------------------------------
TOTAL                 14      7    50%
```

Figure 5: coverage term missing

For long test suites: Use ALL the cores!

If you have lots of tests and they take some time you can use `pytest-xdist` which will run tests in parallel when you use the `-n` flag:

```
pip install pytest-xdist
pytest --cov=src --cov-report term-missing -n 16
```

Note, that in our example this will run slower since we only have two functions and the parallelising overhead is not really worth it. But if your tests are running longer than a few seconds, this will likely already be worth it.

Section 4

Testing your tests: How good are my tests?

What do you think about these tests?

```python
def add(a, b):
    return a + b

def mul(a, b):
    return a * b
```

```python
def test_add():
    result = add(2, 2)
    assert result == 4

def test_mul():
    result = mul(2, 2)
    assert result == 4
```

Consider this bug:

```python
def add(a, b):
    return a * b

def mul(a, b):
    return a + b
```

```python
def test_add():
    result = add(2, 2)
    assert result == 4

def test_mul():
    result = mul(2, 2)
    assert result == 4
```

The behaviour of the functions has completely changed, but the tests will pass, so we might think all is well! (Remember the quote from the beginning?)

Mutation testing

This is precisely what mutation testing does: It takes your code and creates mutated variants of your code. It will then run your tests to see if your tests are good enough to catch the mutation. If your tests still pass, you might have to improve your testing! In this example it may be enough to simply add a few more test cases:

```python
def add(a, b):
    return a * b
```

```python
def test_add():
    result = add(2, 2)
    assert result == 4

    result = add(7, 2)
    assert result == 9

    result = add(1, 2)
    assert result == 3
```

Mutation Testing in Python

Name/Link	Pros	Cons
mutmut	Actively maintained, can cache previous progress	Not written by me
pymute	Written by me	Can't cache previous progress (yet)
mut.py	Couldn't find any	Not actively maintained, not compatible with current versions, do not use
mut.py fork	Was forked because mut.py was not actively maintained	Also not actively maintained

Section 5

Some additional features/pitfalls to consider when testing

Feature: A nicer way of testing multiple inputs

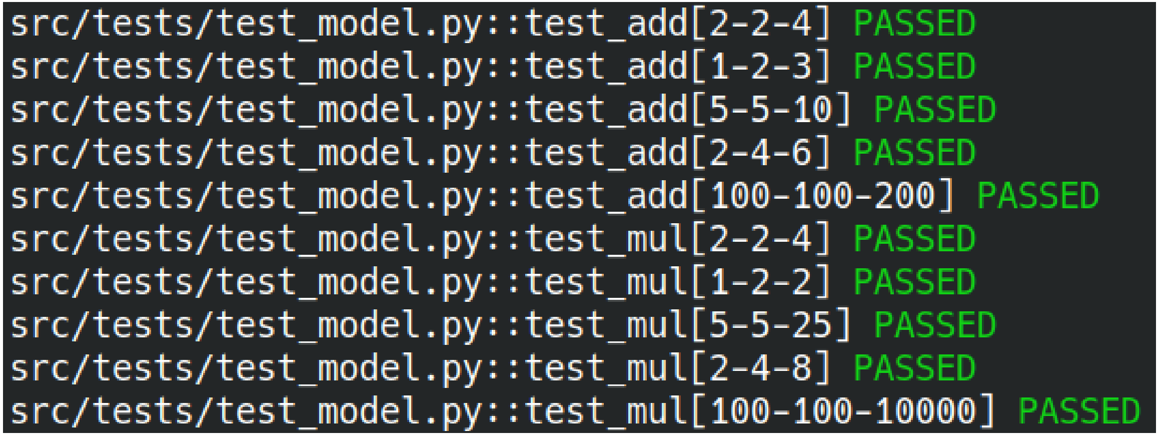
- Previously we called our function multiple times with different inputs
- Pytest allows us something nicer, by parametrising test functions using a decorator

Addition Example

```python
@pytest.mark.parametrize(
    "a,b,expected_result",
    [(2, 2, 4), (1, 2, 3), (5, 5, 10), (2, 4, 6), (100, 100, 200)],
)
def test_add(a, b, expected_result):
    result = add(a, b)
    assert result == expected_result
```

Multiplication Example

```python
@pytest.mark.parametrize(
    "a,b,expected_result",
    [(2, 2, 4), (1, 2, 2), (5, 5, 25), (2, 4, 8), (100, 100, 10000)],
)
def test_mul(a, b, expected_result):
    result = mul(a, b)
    assert result == expected_result
```



```
src/tests/test_model.py::test_add[2-2-4] PASSED
src/tests/test_model.py::test_add[1-2-3] PASSED
src/tests/test_model.py::test_add[5-5-10] PASSED
src/tests/test_model.py::test_add[2-4-6] PASSED
src/tests/test_model.py::test_add[100-100-200] PASSED
src/tests/test_model.py::test_mul[2-2-4] PASSED
src/tests/test_model.py::test_mul[1-2-2] PASSED
src/tests/test_model.py::test_mul[5-5-25] PASSED
src/tests/test_model.py::test_mul[2-4-8] PASSED
src/tests/test_model.py::test_mul[100-100-10000] PASSED
```

Figure 6: Parametrised output

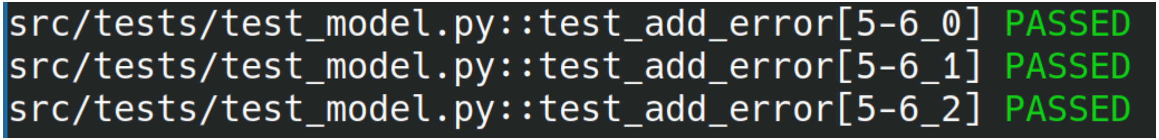
Feature: Check that your program is raising errors

We want to test that this function raises an error if the input is in a bad state:

```python
def add(a, b):
    if isinstance(a, str) or isinstance(b, str):
        raise ValueError("Adding of strings not allowed!")

    return a + b
```

We can use `parametrize` to create multiple inputs with bad states, and `pytest.raises` to assert that the correct error is raised:

```python
@pytest.mark.parametrize("a,b", [(5, "6"), ("5", 6), ("5", "6")])
def test_add_error(a, b):
    with pytest.raises(ValueError, match="Adding of strings not allowed!"):
        add(a, b)
```



```
src/tests/test_model.py::test_add_error[5-6_0] PASSED
src/tests/test_model.py::test_add_error[5-6_1] PASSED
src/tests/test_model.py::test_add_error[5-6_2] PASSED
```

Figure 7: pytest.raises passes

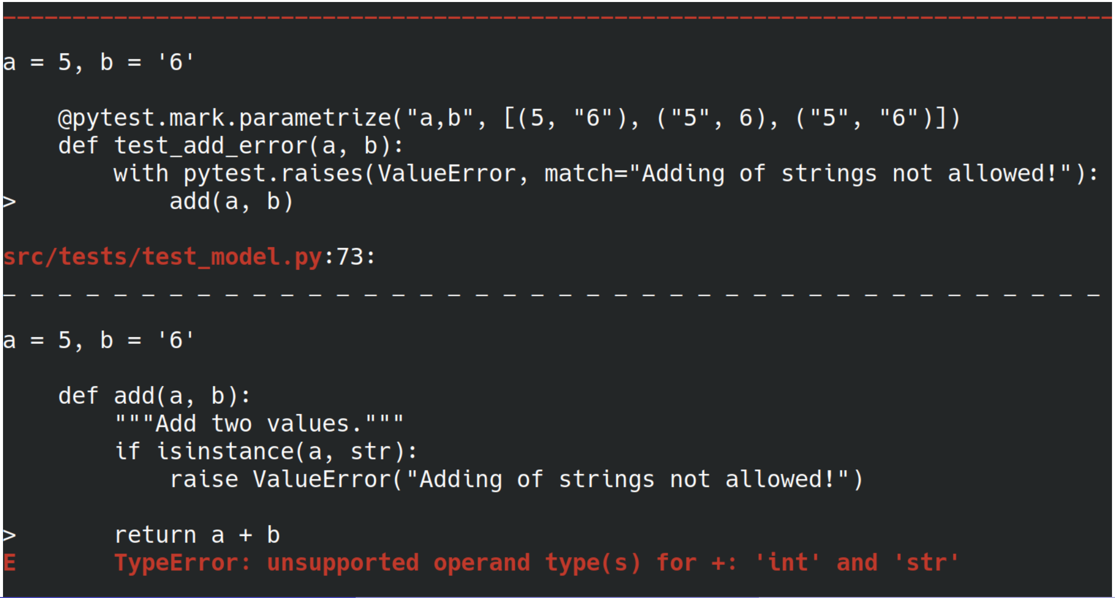
Imagine that in some commit for whatever reason we accidentally change this:

```python
def add(a, b):
    if isinstance(a, str) or isinstance(b, str):
        raise ValueError("Adding of strings not allowed!")

    return a + b
```

To this:

```python
def add(a, b):
    if isinstance(a, str):
        raise ValueError("Adding of strings not allowed!")

    return a + b
```

Our tests will then fail when `b` is in a bad state, since we don't get the error that we expected:



```
----------------------------------------------------------------------------------------------------

a = 5, b = '6'

    @pytest.mark.parametrize("a,b", [(5, "6"), ("5", 6), ("5", "6")])
    def test_add_error(a, b):
        with pytest.raises(ValueError, match="Adding of strings not allowed!"):
>           add(a, b)

src/tests/test_model.py:73:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 5, b = '6'

    def add(a, b):
        """Add two values."""
        if isinstance(a, str):
            raise ValueError("Adding of strings not allowed!")

>       return a + b
E       TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Pitfall and feature: Equality of floats

- see Floating Point Arithmetic: Issues and Limitations

```python
def test_add_array():
    a = np.arange(9).reshape(3, 3).astype(float)
    b = np.arange(10, 19).reshape(3, 3).astype(float)
    result = add(a, b)
    expected_result = np.array(
        [[10.0, 12.0, 14.0], [16.0, 18.0, 20.0], [22.0, 24.0, 26.0]],
    )
    assert result == pytest.approx(expected_result)
```

Pitfall: Imports and packaging

```python
import sys
from pathlib import Path
import pytest
import numpy as np

src = Path(__file__).parents[0] / ".." / "."
print(src)
sys.path.append(str(src))

from model import add, mul, predict
```

Packaging your Project

- if you have a medium/large sized folder of files with code that is shared by different executables, you probably want to properly package it (by that I only mean making the local project installable via pip, NOT necessarily publishing on PyPI)
  - see Publishing a Python Package
  - see Video on packaging with historical context
- python/pytest `import` f***ery can be quite confusing
  - see Pytest: Good Integration Practices
  - see Blog post on package layout and problems it can induce to python/pytest import f***ery
- if your project is packaged and you want to specifically test the installed version rather than local modules available in your `$PYTHONPATH`, you may want to look into using tox
- **The main point here is that you do NOT need to properly package your project in order to write and run tests with pytest. You can do this easily with any project you already have that defines any functions or classes you can import elsewhere.**

A bunch of other potentially useful links and ressources:

1. Use Pytest Approx to for Numerical Accuracy a.k.a don't simply test equality of floats
2. Scientific Python