Testing Python Code

Juan F. Imbet Ph.D.

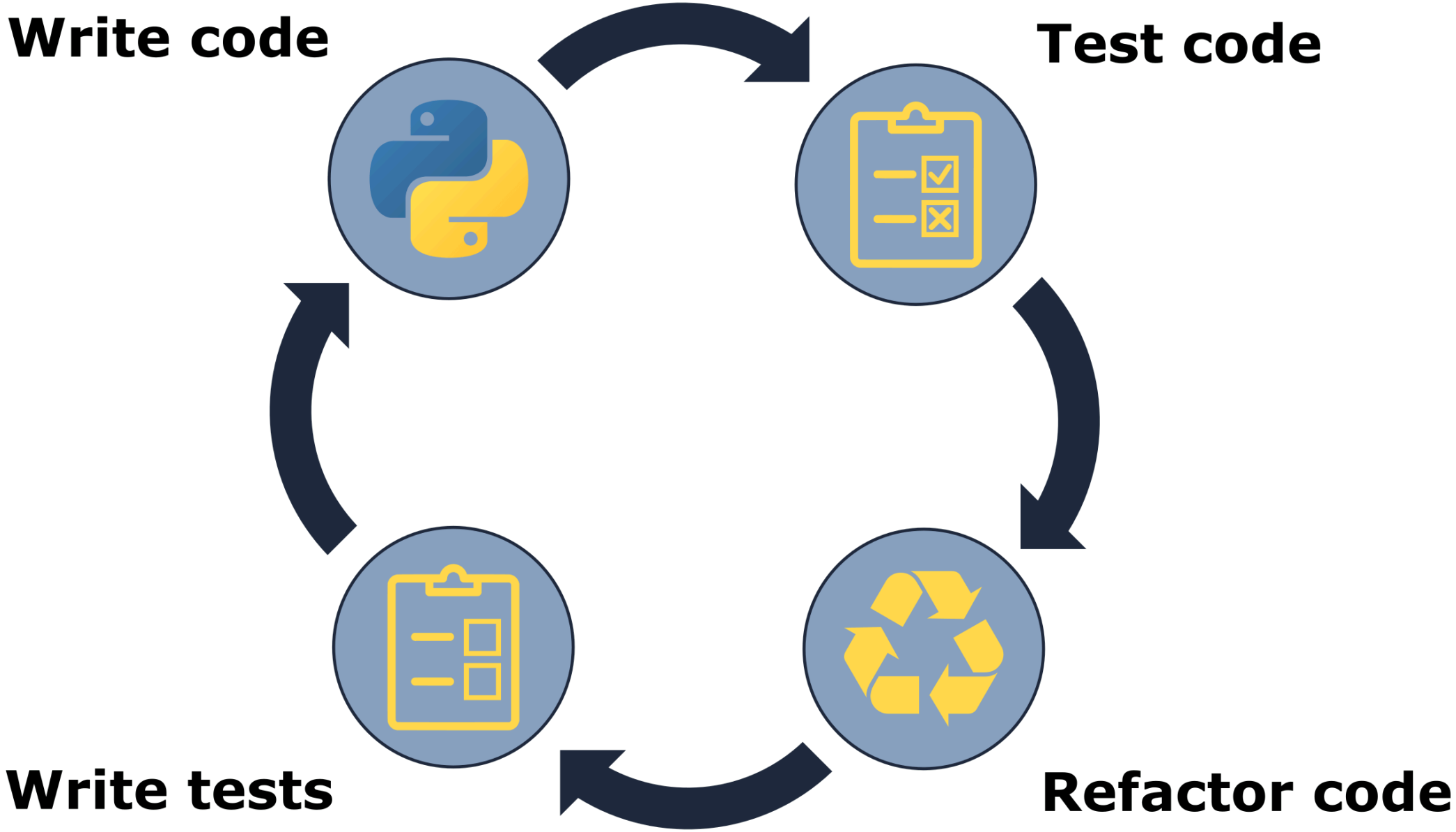
Testing

- In general, the goal of testing is to check that your code produces the results you expect it to. You probably already conduct informal tests of your code in your current workflow.
- In general, the goal of testing is to check that your code produces the results you expect it to. You probably already conduct informal tests of your code in your current workflow.
- In Python, tests are usually written using an `assert` statement, which checks the truth of a given expression, and returns a user-defined error message if the expression is false.

```
assert `expression`, `error message`
```

The testing workflow

- Write a test.
- Write the code to be tested.
- Test the code.
- Refactor code (make small changes).
- Repeat.


Write code
Test code
Write tests
Refactor code

`pytest`

- `pytest` is a testing framework that makes building simple and scalable tests easy.Install it using `pip install pytest`.

```
# content of test_sample.py
def inc(x):
    return x + 1

def test_answer():
    assert inc(3) == 5
```

- Tests are defined as functions prefixed with `test_` and contain one or more statements that `assert` code produces an expected result or raises a particular error.
- Tests are put in files of the form `test_*.py` or `*_test.py`, and are usually placed in a directory called tests/ in a package's root.

```
pytest
================================== test session starts ==================================
platform win32 -- Python 3.11.9, pytest-8.3.3, pluggy-1.5.0
rootdir: ...
configfile: pyproject.toml
plugins: anyio-4.6.0
collected 1 item

test_sample.py F                                                                  [100%]

======================================== FAILURES ========================================
_____________________________________ test_answer ______________________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError
================================ short test summary info ================================
FAILED test_sample.py::test_answer - assert 4 == 5
================================== 1 failed in 0.32s ====================================
```

Tests in a package

Go to the `stable` branch.

```
pybacktestchain
├── CHANGELOG.md
├── CONDUCT.md
├── CONTRIBUTING.md
├── docs
│   └── ...
├── LICENSE
├── README.md
├── poetry.lock
├── pyproject.toml
├── src
│   └── ...
└── tests                          <--------
    └── test_pybacktestchain.py    <--------
```

Testing with `pytest`, e.g. proper imports

```python
# content of test_imports.py
def test_data_import():
    from pybacktestchain.data_module import FirstTwoMoments
    assert FirstTwoMoments is not None
def test_broker_import():
    from pybacktestchain.broker import Backtest, StopLoss
    assert Backtest is not None
    assert StopLoss is not None
def test_blockchain_import():
    from pybacktestchain.blockchain import load_blockchain
    assert load_blockchain is not None
```

```
poetry add --group dev pytest
```

```
pytest tests/
```

Test that a new backtest is added to a new blockchain

Check `test_blockchain.py` in the `tests/` directory.

Unit vs Integration tests

- Unit tests are tests that check that individual units of code (e.g. functions) work as expected.
- Integration tests check that different parts of the code work together as expected `test_new_blockchain.py`.

Parametrization

- Parametrization allows you to run the same test with different inputs and expected outputs.

```
import pytest

@pytest.mark.parametrize("input, expected", [
    (1, 2),
    (2, 3),
    (3, 4),
])
def test_inc(input, expected):
    assert inc(input) == expected
```