BCG GAMMA

nibble

Advanced Testing Training

An introduction to Software testing in a Machine Learning context

October 2020

Agenda

I. Context and general presentation

II. Focus on End-to-End Testing

III. A few words about Performance Testing

IV. Focus on Integration Testing

Day 2

V. Focus on Unit Testing

VI. Code Refactoring principles

VII. Introduction to Continuous Integration

Part III — Performance testing & Data Quality

- Rationale

- Challenges

- Examples

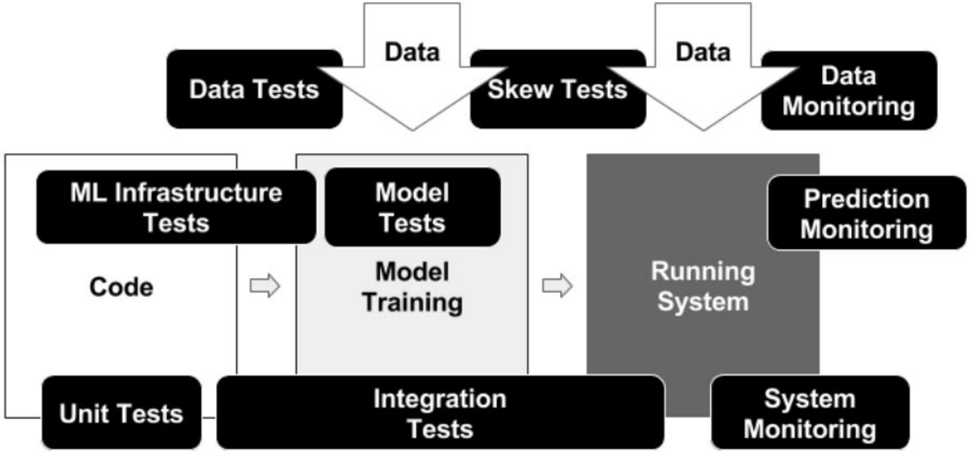
Part III — Performance testing & Data Quality

- **Rationale**

- Challenges

- Examples

Critical part of testing ML projects



ML-Based System Testing and Monitoring

Data evolves

Models need to evolve as well

Part III — Performance testing & Data Quality

- Rationale

- **Challenges**

- Example

Schema updates

Data types changes

Statistical drift

…

Model correctness

Model improvements

Automated checks?

Part IV — Performance testing & Data Quality

- Rationale

- Challenges

- **Example**

Part IV — Performance testing & Data Quality

- Rationale

- Challenges

- **Example…. During the workshop!**

And now, for something completely different.

Agenda

I. Context and general presentation

II. Focus on End-to-End Testing

III. A few words about Performance Testing

IV. Focus on Integration Testing

V. Focus on Unit Testing

VI. Code Refactoring principles

VII. Introduction to Continuous Integration

Part IV — Focus on Integration Testing

- Where do they stand

- Benefits

- Usage

Part IV — Focus on Integration Testing

- **Where do they stand**
- Benefits
- Usage

They focus on the abstraction **between end-to-end and units**

But why?

Gives us more knowledge about points of failure

Part IV — Focus on Integration Testing

- Where do they stand
- **Benefits**
- Usage

Check interaction with 3rd-parties

Verify that modules communicate reliably

What they are **not** used for…

~~Checking business logic~~

~~Simplify refactoring~~

~~Identify code regressions~~

Part III — Focus on Integration Testing

- Where do they stand
- Benefits
- **Usage**

Complexity of integration can make them slow…

Again, good candidate for CI servers

We need an extra tool
for efficient
implementation...

Have you met

pytest ?

⚠ TEASER

`pytest` is a tool for efficiently **writing and running** automated tests in Python

Cool feature #1

auto-discovery and
results summary

```
# tests/test_something.py

def test_one_thing():
    assert 1 + 1 == 2


def test_another_thing():
    assert 'a' in 'gamma'
```

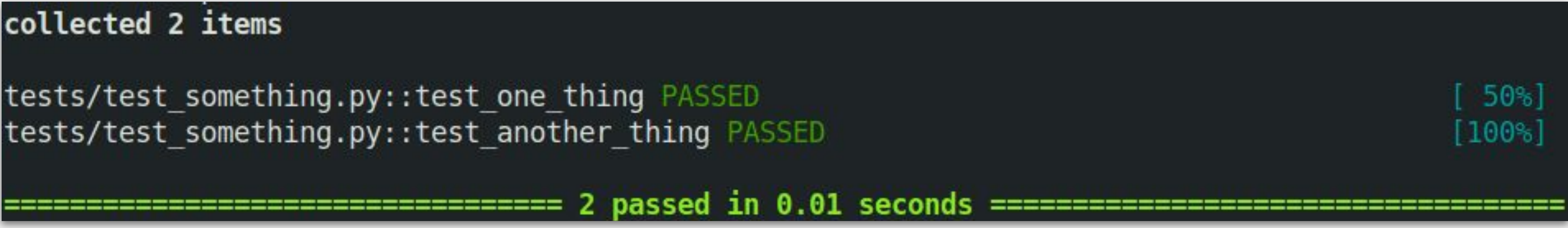
```python
# tests/test_something.py

def test_one_thing():
    assert 1 + 1 == 2


def test_another_thing():
    assert 'a' in 'gamma'
```

All functions named `test_xxx...` are automatically discovered

Shell command

```
$ pytest -v
```

Output



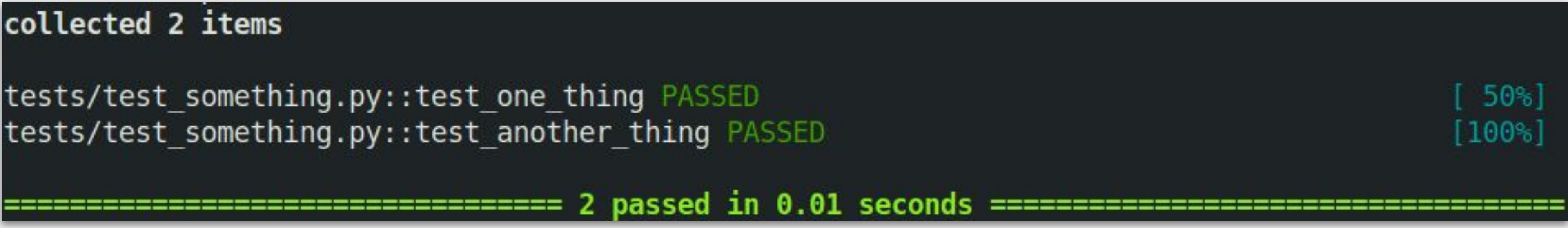
```
collected 2 items

tests/test_something.py::test_one_thing PASSED                                                     [ 50%]
tests/test_something.py::test_another_thing PASSED                                                 [100%]

================================== 2 passed in 0.01 seconds ===================================
```

Shell command

```
$ pytest -v
```

“verbose” output (optional)

Output



```
collected 2 items

tests/test_something.py::test_one_thing PASSED                                  [ 50%]
tests/test_something.py::test_another_thing PASSED                              [100%]

================================== 2 passed in 0.01 seconds ===================================
```

Cool feature #2

Fixtures

```python
# tests/test_with_fixture.py

@pytest.fixture
def user():
    return User(
        first_name='George',
        last_name='Abitbol'
    )

def test_user_has_first_name(user):
    assert user.first_name is not None
```

```python
# tests/test_with_fixture.py

@pytest.fixture
def user():
    return User(
        first_name='George',
        last_name='Abitbol'
    )

def test_user_has_first_name(user):
    assert user.first_name is not None
```

`pytest` will automatically bind the `user` parameter to the value returned by the fixture

Using `yield` allows to execute **tear down code**, *ie* code to be run **after a test has completed**.

```python
from socket import socket, AF_INET, SOCK_STREAM
import pytest


@pytest.fixture
def tcp_server():
    server = socket(AF_INET, SOCK_STREAM)
    server.bind('localhost', 8000)
    server.listen()
    yield server
    server.close()
```

```python
from socket import socket, AF_INET, SOCK_STREAM
import pytest


@pytest.fixture
def tcp_server():
    server = socket(AF_INET, SOCK_STREAM)
    server.bind('localhost', 8000)
    server.listen()
    yield server
    server.close()
```

Setup

```python
from socket import socket, AF_INET, SOCK_STREAM
import pytest


@pytest.fixture
def tcp_server():
    server = socket(AF_INET, SOCK_STREAM)
    server.bind('localhost', 8000)
    server.listen()
    yield server
    server.close()
```

Yield to test case

```python
from socket import socket, AF_INET, SOCK_STREAM
import pytest


@pytest.fixture
def tcp_server():
    server = socket(AF_INET, SOCK_STREAM)
    server.bind('localhost', 8000)
    server.listen()
    yield server
    server.close()
```

Tear down properly

Add a “scope” parameter to **share fixtures** across classes, modules or the whole pytest session.

https://docs.pytest.org/en/stable/fixture.html#scope-sharing-fixtures-across-classes-modules-packages-or-session

```python
from socket import socket, AF_INET, SOCK_STREAM
import pytest

@pytest.fixture(scope='session')
def tcp_server():
    server = socket(AF_INET, SOCK_STREAM)
    server.bind('localhost', 8000)
    server.listen()
    yield server
    server.close()
```

Share fixtures automatically in `conftest.py`

No need to import: `pytest` will load them automatically.

https://docs.pytest.org/en/latest/fixture.html#conftest-py-sharing-fixture-functions

Fixtures are **extremely flexible**, and **useful in many situations**.

Official doc is full of interesting patterns 😉

Cool feature #3

Markers

```python
import pytest


@pytest.mark.xfail
def test_should_fail():
    assert True is False
```

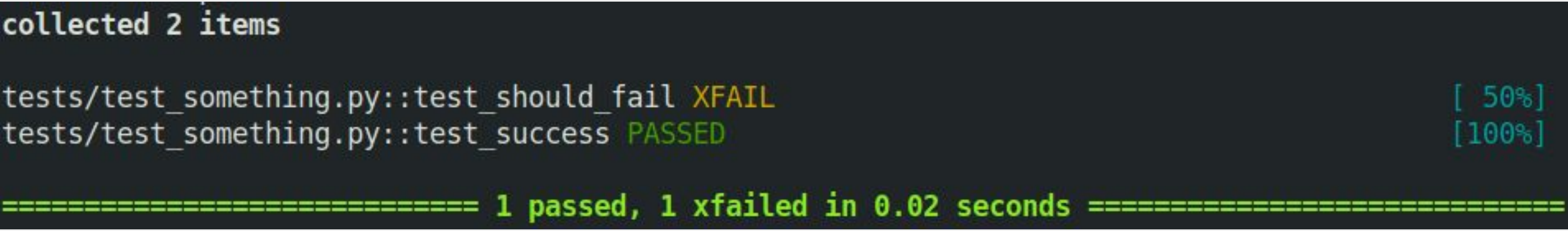
```
import pytest


@pytest.mark.xfail
def test_should_fail():
    assert True is False
```

built-in

Official docs: https://docs.pytest.org/en/3.0.5/mark.html

```python
import pytest

@pytest.mark.xfail
def test_should_fail():
    assert True is False

def test_success():
    assert True
```

Output sample



```
collected 2 items

tests/test_something.py::test_should_fail XFAIL                                         [ 50%]
tests/test_something.py::test_success PASSED                                            [100%]

============================= 1 passed, 1 xfailed in 0.02 seconds =============================
```

```python
import pytest

@pytest.mark.slow
def test_something():
    # do something slow...
    assert True
```

```python
import pytest

@pytest.mark.slow
def test_something():
    # do something slow...
    assert True
```

custom

Official docs: https://docs.pytest.org/en/3.0.5/example/markers.html#mark-examples

Run **only** tests marked as `slow`

```
$ pytest -m slow
```

Run all tests, **except** the ones marked as `slow`

```
$ pytest -m “not slow”
```

```
import pytest

@pytest.mark.bcg_gamma
def test_something():
    # do anything...
    assert True
```

Can be anything you want

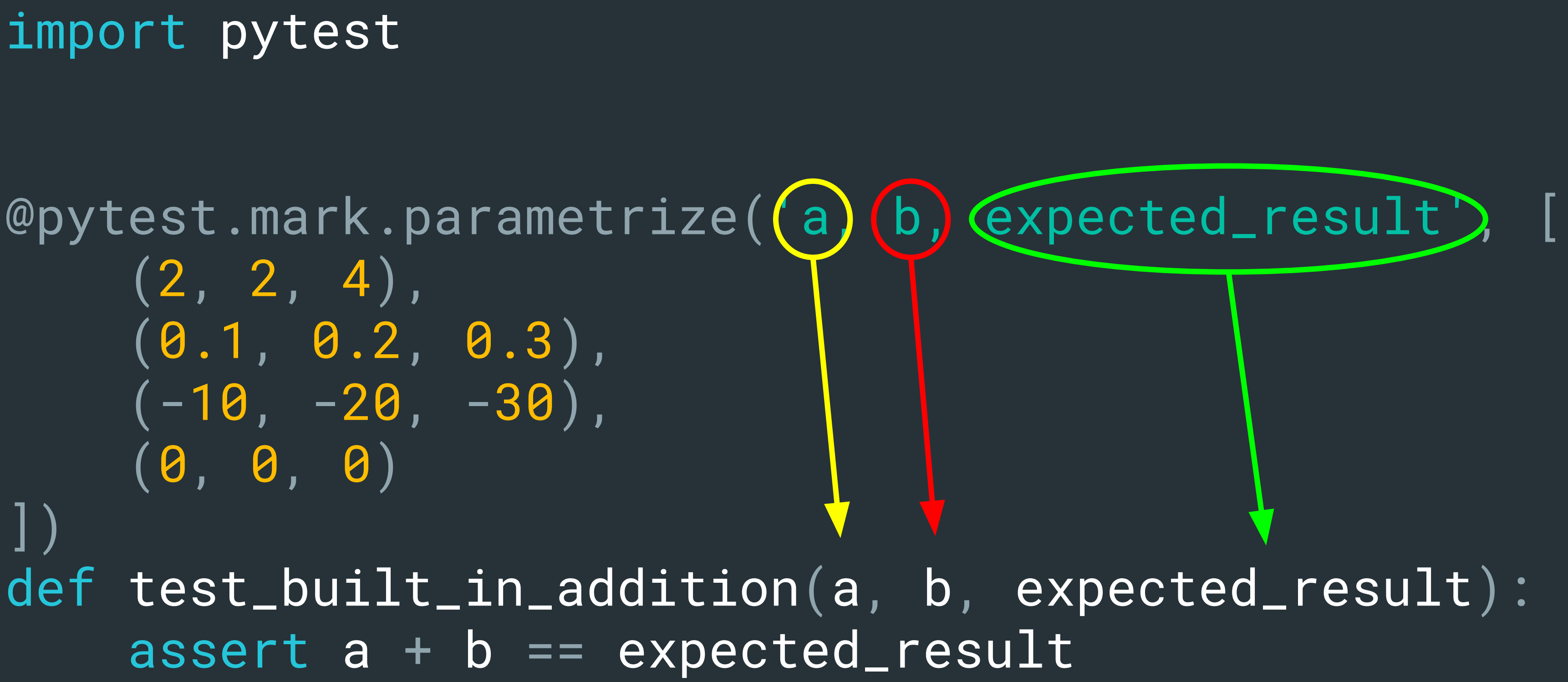
Official docs: https://docs.pytest.org/en/3.0.5/example/markers.html#mark-examples

Cool feature #4

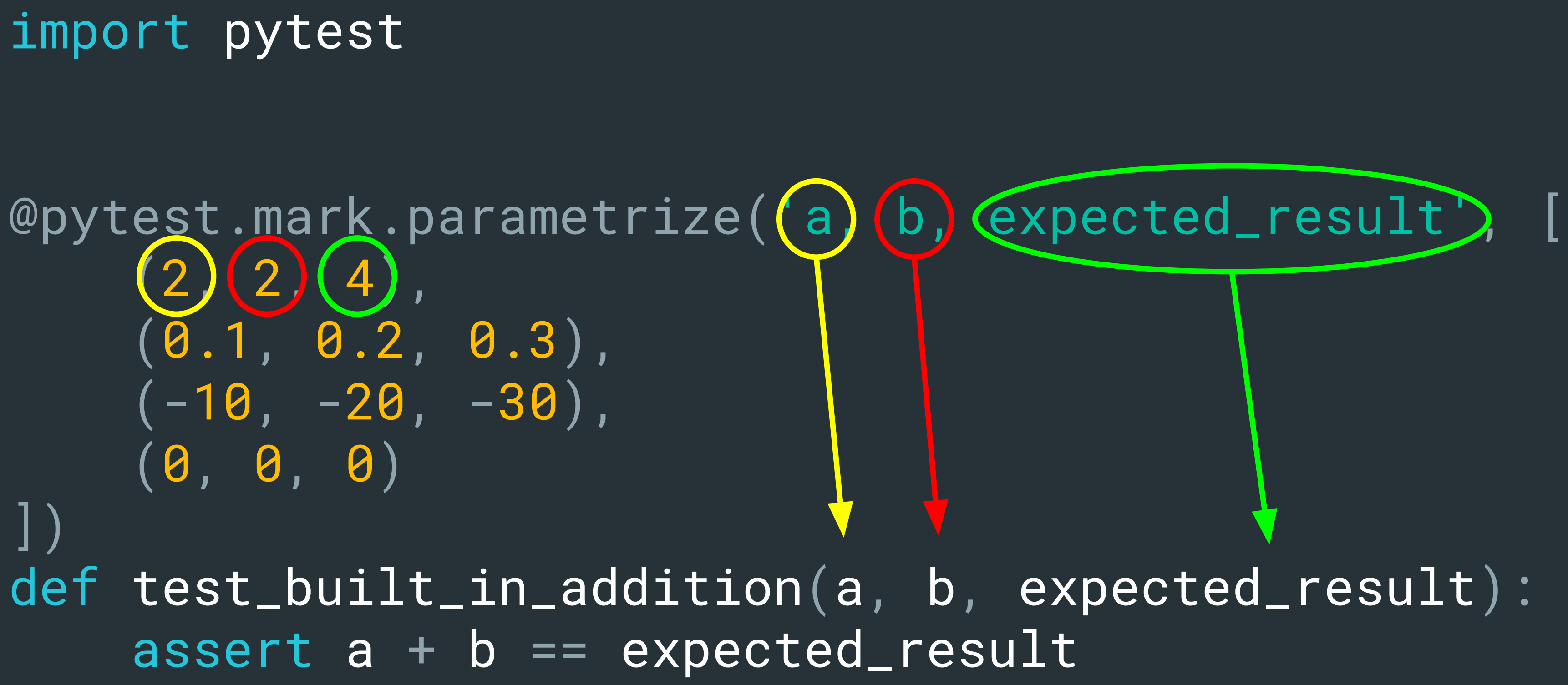
Parametrize

built-in marker

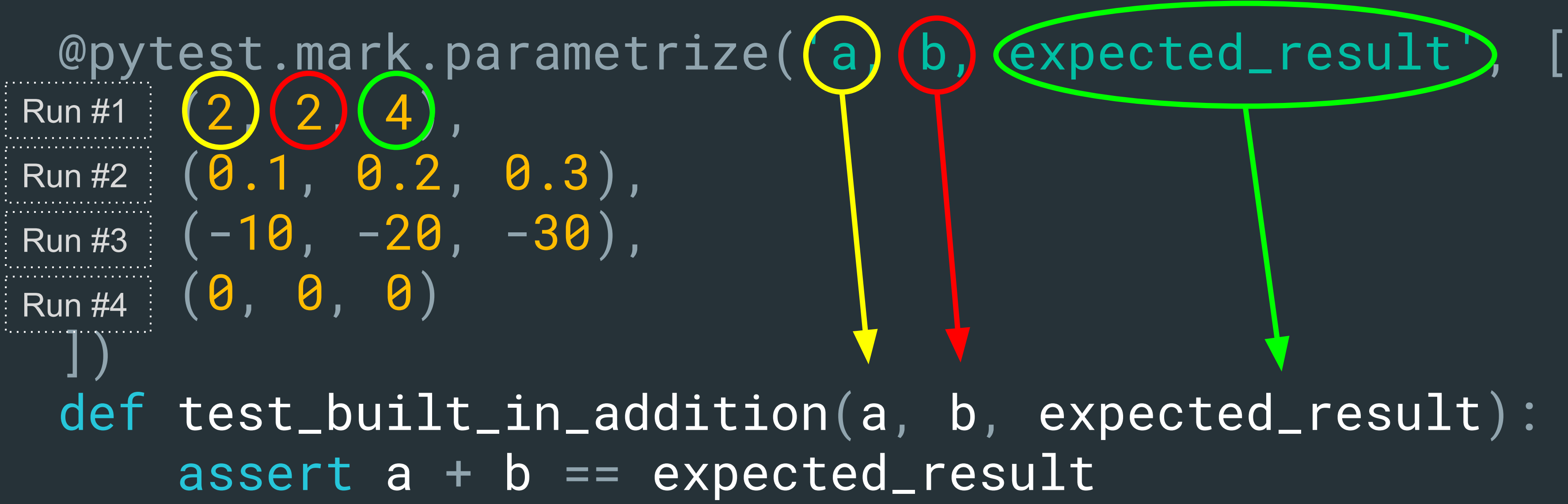
```python
import pytest


@pytest.mark.parametrize('a, b, expected_result', [
    (2, 2, 4),
    (0.1, 0.2, 0.3),
    (-10, -20, -30),
    (0, 0, 0)
])
def test_built_in_addition(a, b, expected_result):
    assert a + b == expected_result
```



```python
import pytest


@pytest.mark.parametrize('a, b, expected_result', [
    (2, 2, 4),
    (0.1, 0.2, 0.3),
    (-10, -20, -30),
    (0, 0, 0)
])
def test_built_in_addition(a, b, expected_result):
    assert a + b == expected_result
```



```python
import pytest


@pytest.mark.parametrize('a, b, expected_result', [
    (2, 2, 4),
    (0.1, 0.2, 0.3),
    (-10, -20, -30),
    (0, 0, 0)
])
def test_built_in_addition(a, b, expected_result):
    assert a + b == expected_result
```



```
import pytest


@pytest.mark.parametrize('a, b, expected_result', [
    (2, 2, 4),
    (0.1, 0.2, 0.3),
    (-10, -20, -30),
    (0, 0, 0)
])
def test_built_in_addition(a, b, expected_result):
    assert a + b == expected_result
```

Cool feature #5

Configuration and flexibility

Many options

To be added at runtime, or in a config file...

```
--max-fails

--testpaths

--xfail_strict

--disable-warnings

...
```

https://docs.pytest.org/en/stable/reference.html#configuration-options

Configurable via text file

setup.cfg
pytest.ini
tox.ini
...

```
# pytest.ini


[pytest]
addopts = -p no:warnings
python_files =
    test_*.py
    *_test.py
    check_*.py
...
```

https://docs.pytest.org/en/stable/reference.html#configuration-options

Extendable via existing or custom plugins

Create your own plugins if you want specific integration with one system or another.

https://docs.pytest.org/en/latest/plugins.html

https://docs.pytest.org/en/latest/writing_plugins.html

Your turn to rock!