Unittest in Python

unittest pytest fixtures

Lê Minh Nguyễn - Lab914 Seminar

CONTENTS

Demo

EXPLORER

OPEN EDITORS

PYTEST [SSH: 43.239.223.87]

- .pytest_cache
- intro-to-pytest
- pytest_examples
- pytest-demo
  - demo-tests
    - test_01.py
    - test_02.py
    - test_03.py
    - test_04.py
    - test_05.py
  - resources
    - genres_and_15_best_o...
  - conftest.py
  - fixtures.py

OUTLINE

TIMELINE

test_04.py | test_05.py | fixtures.py

pytest-demo > fixtures.py > load_review_rating

```python
import random
import pytest
import pandas as pd

data_path = "pytest-demo/resources/genres_and_15_best_of_films.csv"
data = pd.read_csv(data_path)
data = [data.iloc[idx] for idx in range(len(data))]

@pytest.fixture(params=random.sample(data, 20))
def load_review_rating(request):
    return request.param["review_rating"], 80

```

PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

```
(base) [2021-11-25 02:16:55] nguyenlm@VBEE-V100-Tesla:/home2/nguyenlm/Projects/another_stuff/pytest
$
```

bash

Python

Testing Terms You May Have Heard

Unit test

Test of a single unit of code in isolation

Integration test

Test how different components of a system work together

Regression test

Testing a previously working program after a change to ensure no problems have been introduced

Smoke test

A subset of test cases verifying the core of a system

Alpha test

In-house test of final functionality of an application

Beta test

Initial release to a subset of real users

Testing Terms You May Have Heard

Unit test

Test of a single unit of code in isolation

Integration test

Test how different components of a system work together

Regression test

Testing a previously working program after a change to ensure no problems have been introduced

Smoke test

A subset of test cases verifying the core of a system

Alpha test

In-house test of final functionality of an application

Beta test

Initial release to a subset of real users

Benefits of Unit Testing

Well-tested code helps you:

- Find bugs earlier
- Iterate faster
- Debug more easily
- Design better code

Benefits of Unit Testing

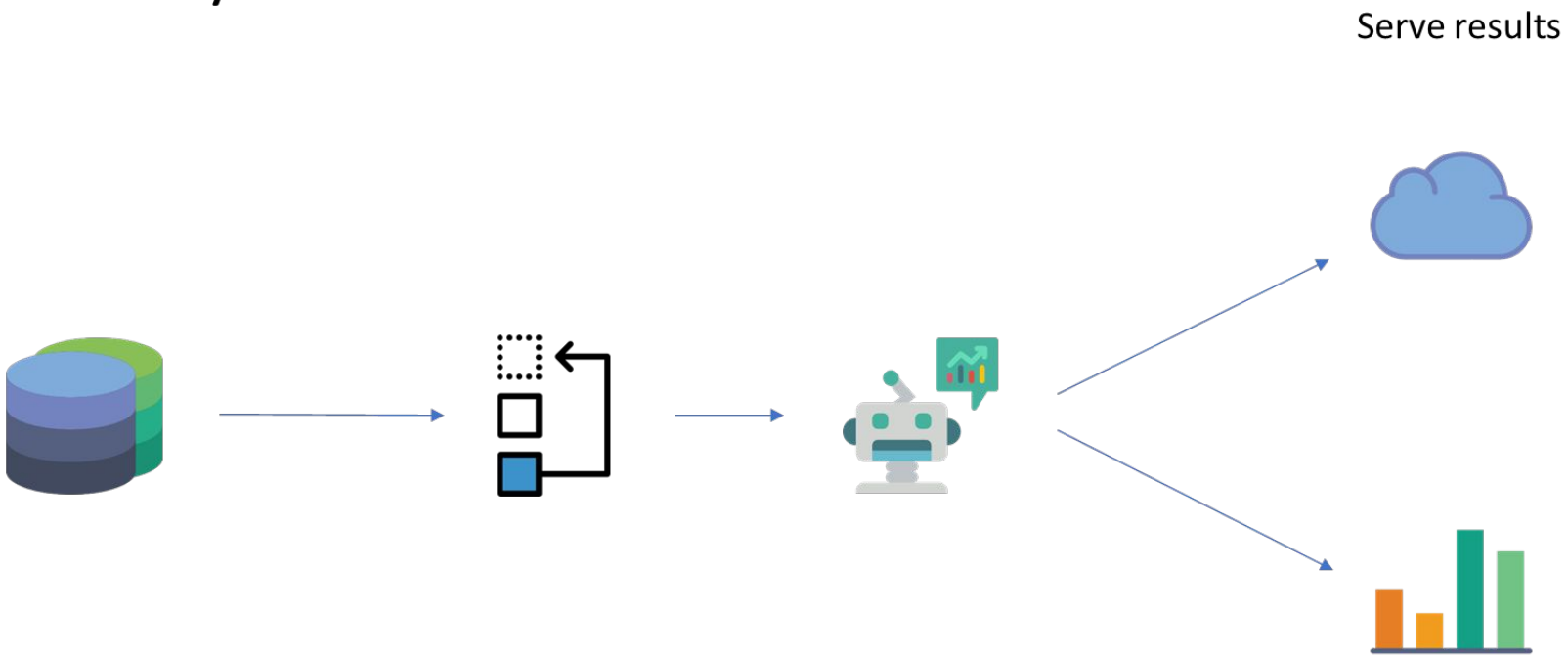
Confidence that your code does what you think it does

So why doesn’t everyone write unit tests?

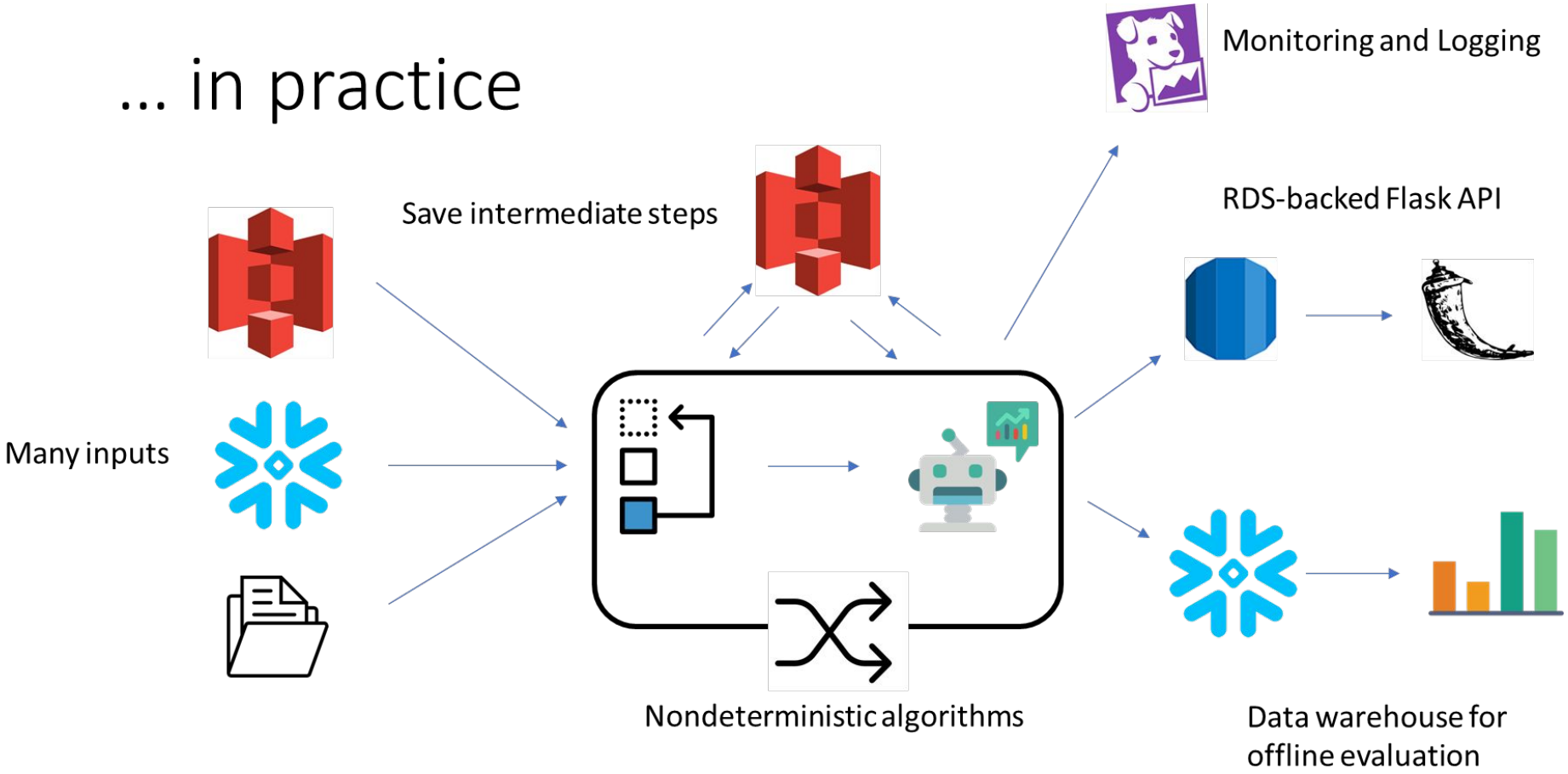
So why doesn’t everyone write unit tests?

Learning to write good tests is an investment...

In theory…


Serve results
Read in data
Build features
Build a model
Evaluate offline

… in practice


Monitoring and Logging
Save intermediate steps
RDS-backed Flask API
Many inputs
Nondeterministic algorithms
Data warehouse for
offline evaluation

This quickly gets overwhelming

Where to start?

- Pick a single, specific functionality to verify
- Use available tools to get everything else out of the way
- In an existing codebase, don’t try to write all the tests at once
- Write tests as early as they can be valuable

CONTENTS

1. Easily Configurable
2. Powerful
3. Most Popular Framework
4. Free

Code Example

```
def add_col(df, new_col_name, default_value):
    """Add a new column to the given dataframe with a default value"""
    if new_col_name in df.columns:
        raise ValueError('column already exists')
    df[new_col_name] = default_value
    return df
```

Basic Unittest

```python
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Basic Unittest

give our test a meaningful name starting with test_

```
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Basic Unittest

define the input to the function we're testing

```
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Basic Unittest

call the function
we're testing

```
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Basic Unittest

define the output we are expecting

```
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Basic Unittest

check that the actual output matches the output we were expecting

```
def test_add_col_passes():
    # setup
    df = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })

    # call function
    actual = add_col(df, 'col_d', 'd')

    # set expectations
    expected = pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })

    # assertion
    pd.testing.assert_frame_equal(actual, expected)
```

Useful Options for Pytest

```
pytest [option] test_file|test_folder
```

- -s (print all string output)
- -v (print names of individual tests as they run)
- -x (stop at first failure)
- -k (only run tests matching following keywords)

CONTENTS

Fixtures

- Special functions pytest keeps track of to safely share resources and/or resource definitions
- A modular approach to setup & teardown methods

Defining New Fixtures

```python
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```python
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

Defining New Fixtures

decorator tells pytest this is a fixture

```
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

Defining New Fixtures

normal python function

```python
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```python
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
    })
```

Defining New Fixtures

```
def test_add_col_1(df, df_with_col_d):
    actual   = add_col(df, 'col_d', 'd')
    expected = df_with_col_d
    pd.testing.assert_frame_equal(actual, expected)
```

```
def test_add_col_2(df, df_with_col_d):
    actual   = add_col(df, 'col_d', 'd')
    expected = df_with_col_d
    pd.testing.assert_frame_equal(actual, expected)
```

```
@pytest.fixture(scope="session")
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```
@pytest.fixture()
def df_with_col_d():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
     })
```

Defining New Fixtures

pytest passes in value returned from `df_with_col_d` function

pytest passes in value returned from `df` function

```
def test_add_col(df, df_with_col_d):
    actual   = add_col(df, 'col_d', 'd')
    expected = df_with_col_d
    pd.testing.assert_frame_equal(actual, expected)
```

```
@pytest.fixture()
def df():
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
    })
```

```
@pytest.fixture()
def df_with_col_d(df):
    return pd.DataFrame({
        'col_a': ['a', 'a', 'a'],
        'col_b': ['b', 'b', 'b'],
        'col_c': ['c', 'c', 'c'],
        'col_d': ['d', 'd', 'd'],
     })
```

Useful Built-in Fixtures

The `request` fixture is a special fixture providing information of the requesting test function

```
@pytest.fixture(params=[1,2,3,4,5],
scope="function")
def load_number(request):
    return request.param
```

`capsys` captures any values written to `stderr` or `stdout` during the execution of the test

```
def test_function_1(capsys):
    function_1()
    out, err = capsys.readouterr()
    assert out == 'Inside function 1\n'
```

Flexibility of Fixtures

- Define the scope of a fixture
- Compose fixtures
- Execute custom teardown code when leaving scope
- Access the test context inside fixture
- Parameterize fixtures

Flexibility of Fixtures

define fixture scope

access test context with request argument

```
import pytest
from emaillib import Email, MailAdminClient

@pytest.fixture(scope='session')
def receiving_user(request):
    user = MailAdminClient().create_user()

    def delete_user():
        mail_admin.delete_user(user)

    request.addfinalizer(delete_user)
    return user
```

execute custom teardown code

CONTENTS

Reference and Source Code

[1] Htorrence - PyData Unit Test Talk 2018

htorrence/pytest_examples: Reference package for unit tests (github.com)

[2] Pluralsight - Intro To Pytest

pluralsight/intro-to-pytest: An introduction to PyTest with lots of simple, hackable examples (github.com)

[3] Source Code For This Seminar

leminhnguyen/pytest-seminar (github.com)

Thank You!