Ing. María Andrea Vignau

The Python Testing Zen

Workshop

Testing

Importance

Strategies

- TDD
- BDD

Tests types

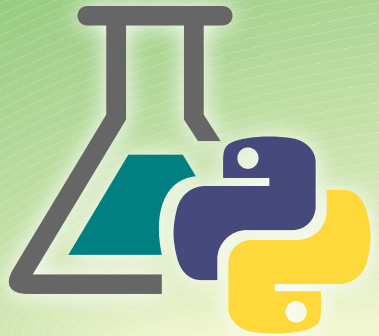
- Unitarios
- Funcionales
- de Integración
- de Rendimiento
- de Usabilidad

Libraries

- UnitTest
- Pytest
- Pytest-BDD
- Unitest.mock Pytest-Mock

Best practices

- Test simples y claros
- Automatizar
- Fixtures y parametros
- Test en capas
- Mantenimiento
- Casos límite



Automated Testing

Testing advantages

Security for

Mantenaibility **Optimizing**

Refactoring

Live documentation of code

Testing advantages

Improve **productivity**

Better **availability**

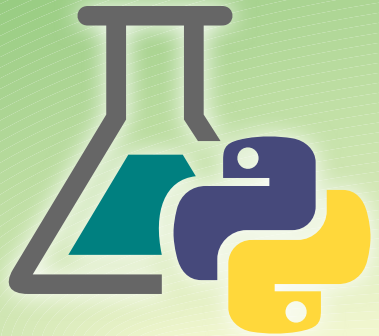
Testing advantages

Quality: early bug detection

Foster **modular** design

Testing advantages

- prepared for **teams**
- **auditable**
- **solid** & **quality**



Development Strategies

Test driven design

First write tests

Then the code

Optimize

Test driven design

1 Write tests

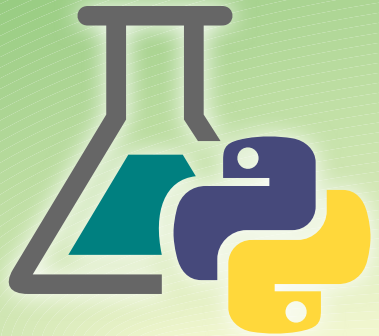
2 Pass tests

3 Refactor

Describe the expected **functionality** in a **readable** form for non-technical people and **use it for testing!**

Behavior driven design

Tests are written as **use cases**. They serve as **documentation**.



Test types

Unit Test

They test a single unit of code (function, method) independently.

Example

Check if a function for adding numbers returns the expected result.

Functional Test

They verify if a system functionality is working properly.

Black box: Implementation details are not known..

Example

Ensure that a registration form saves user data.

Test de integración

They verify that the modules or components are correctly coordinated.

Example

Check that the authentication works against the database.

Alcance de tests

Unit Test

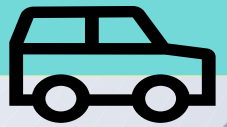
Check loose parts

Functional Test

Check functions such as accelerate

Integration Test

Ensure that the whole car works together



Tipos de test

Unit

Functional

Integration

Performance Test

Evaluate speed, scalability and responsiveness.

Example

Test the system with more users than usual to see its performance.

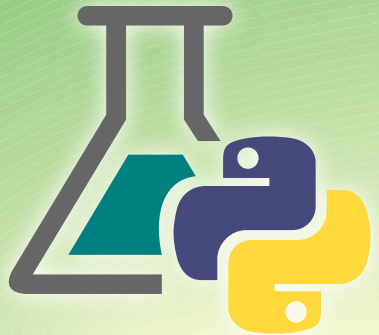
Usability Test

Check if the application is easy to use.

Example

Generate tests with end users.

Perform A/B testing.



Libraries

Libraries

Unittest

Python standard library

Pytest

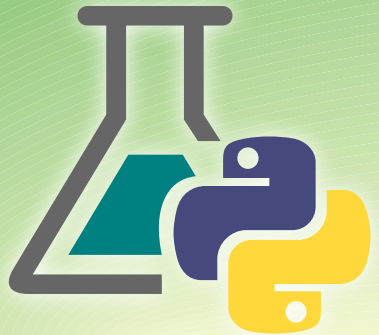
Flexible, expressive and powerful

Pytest-BDD Behave

Behavior Driven Design

Unitest.mock Pytest-mock

Create fake test objects



Unittest

Unittest:

Unit testing.

Configuration and cleaning.

Verification of expected results.

UnitTest

Inherited from

TestCase

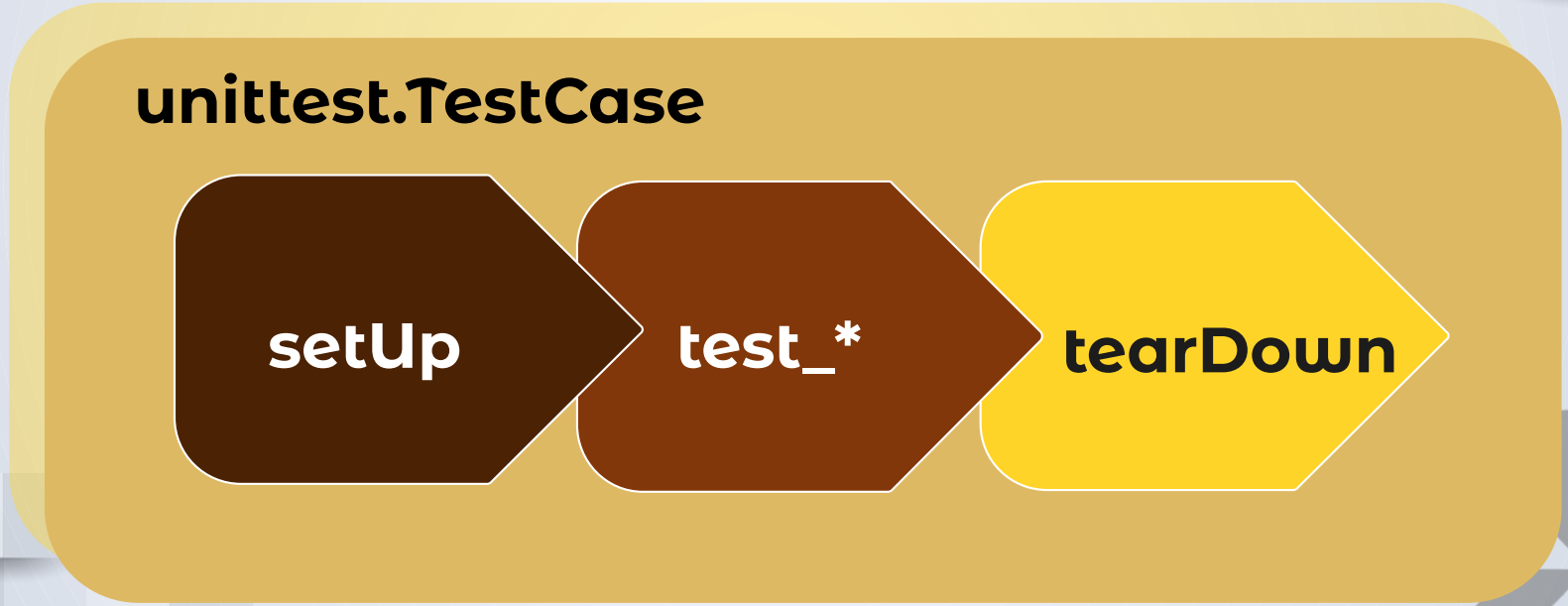
check using

assertNNN

```python
import unittest
def suma(a, b):
    return a + b

class TestSuma(unittest.TestCase):
    def test_suma(self):
        resultado = suma(2, 3)
        self.assertEqual(resultado, 5)

if __name__ == '__main__':
    unittest.main()
```

UnitTest

TestSuite



UnitTest

SetUp

Prepare environment

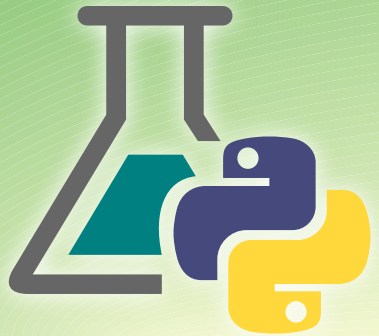
TearDown

Clean environment

```
class TestSuma(unittest.TestCase):
    def setUp(self):
        self.a = 10
        self.b = 5

    def test_suma(self):
        self.assertEqual(
          self.a + self.b, 15)

    def tearDown(self):
        print("Test finalizado")
```

Demo

```
UnitTest
```



Pytest

PyTest vs UnitTest

- Cleaner syntax
- Use assert directly
- Native support for fixtures and parametrize
- Extensible with plugins

- Verbosity in methods
- Methods such as assertEqual
- Need manual configuration
- Does not accept functions

PyTest

functions are called

test_ ...

checks with

assert

```python
import pytest


def suma(a, b):
    return a + b


def test_suma():
    assert suma(2, 3) == 5
```

PyTest: Fixtures

Fixtures are very practical

Automatically used with autouse

Different scopes:

Module, package or global

```python
@pytest.fixture
def datos():
    return {"a": 10, "b": 20}

def test_suma(datos):
    assert datos["a"] + datos["b"] == 30
```

PyTest: Parametrize

Parameterization:

Runs the same test with multiple values.

```
@pytest.mark.parametrize("x, y, esperado", [
    (1, 1, 2), (2, 3, 5),
])
def test_suma(x, y, esperado):
    assert x + y == esperado
```

PyTest

Error Handling

Checking exceptions with `pytest.raises`

```python
def dividir(a, b):
    return a / b

def test_division_por_cero():
    with pytest.raises(ZeroDivisionError):
        dividir(1, 0)
```

PyTest: Plugins

pytest-cov: Measuring code coverage

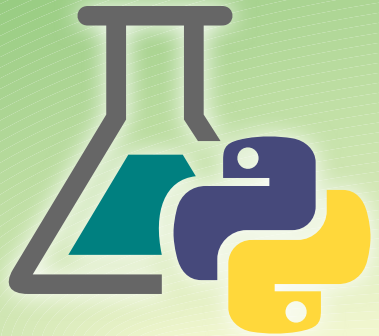
pytest-django: Testing in Django projects.

pytest-flask: Testing in Flask projects.

```
pip install pytest-cov
pytest --cov=mi_modulo
```

Demo

PyTest

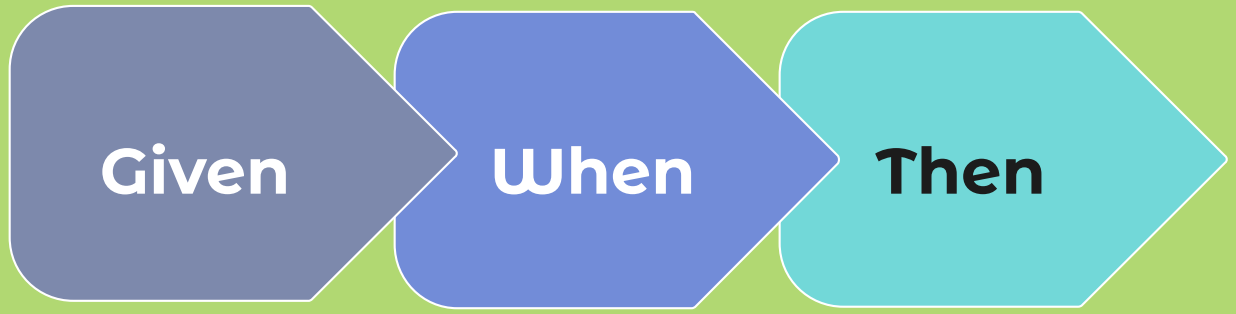


Pytest-BDD

Behavior driven design

Feature

Scenario



Behavior driven design: Lenguaje Gherkin

Key concepts

Feature: Describe a **functionality**

Scenario: An specific use **case**

Given: **Initial state**

When: **Action**.

Then: Expected **result**

Behavior driven design

calculadora.feature

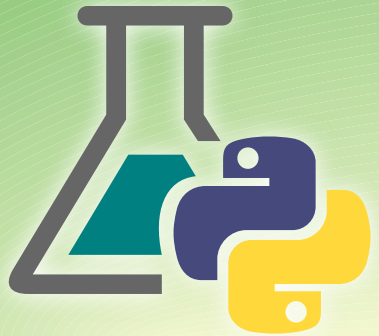
```
Feature: Calculadora
  Scenario: Sumar dos números
    Given tengo los números 2 y 3
    When los sumo
    Then el resultado debe ser 5
```

Behavior driven design

```
from pytest_bdd import scenario, given, when, then

@scenario("calculadora.feature", "Sumar dos números")
def test_suma():
    pass

@given("tengo los números 2 y 3", target_fixture="numeros")
def numeros():
    return 2, 3

@when("los sumo", target_fixture="sumar")
def sumar(numeros):
    return sum(numeros)

@then("el resultado debe ser 5")
def verificar(sumar):
    assert sumar == 5
```

Demo

PyTest-BDD



Patch y mock

Mock

Simulated copy of an object to isolate a part of the system for testing.

Example

Create a simulated sensor that can give all kinds of values.

Patch

Temporarily replacing a real object with a mock to simulate

Example

Using a simulated sensor to test the speedometer

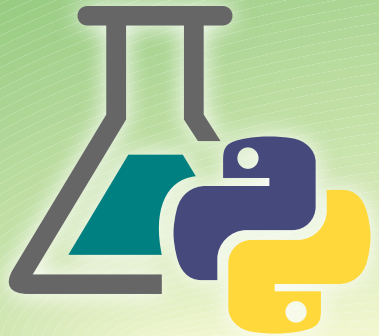
Patch y Mock

```python
def calcular_valor(x):
    return x * 2  # Simulación básica

def procesar_dato(x):
    return f"Resultado: {calcular_valor(x)}"
```

```python
def test_procesar_dato():
    with patch("x.calcular_valor", return_value=42) as mock_calcular:
        resultado = procesar_dato(10)
        mock_calcular.assert_called_once_with(10)
        assert resultado == "Resultado: 42"
```

Demo

Patch y Mock



Best practices

Best practices

Simple tests are easy to maintain and debug.

Test only one thing per test (Arrange-Act-Assert principle).
Use descriptive names for tests
Avoid complex logic within tests.

Best practices

Automating reduces human error and ensures consistency.

Use tools such as pytest or unittest.
Set up a continuous integration pipeline (CI/CD) to run tests automatically with every code change.

Best practices

Isolated tests are faster, more stable and easier to debug.

Use fixtures to prepare and clean data or configurations. Use mocking to simulate interactions with external APIs, databases or other external systems.

Best practices

Mix different scopes tests and types

Different types of tests ensure quality at multiple levels.

Unit tests: individual functions
Functionality testing: interactions between modules.
Integration testing: complete application flow.

Best practices

Outdated tests generate noise and confusion.

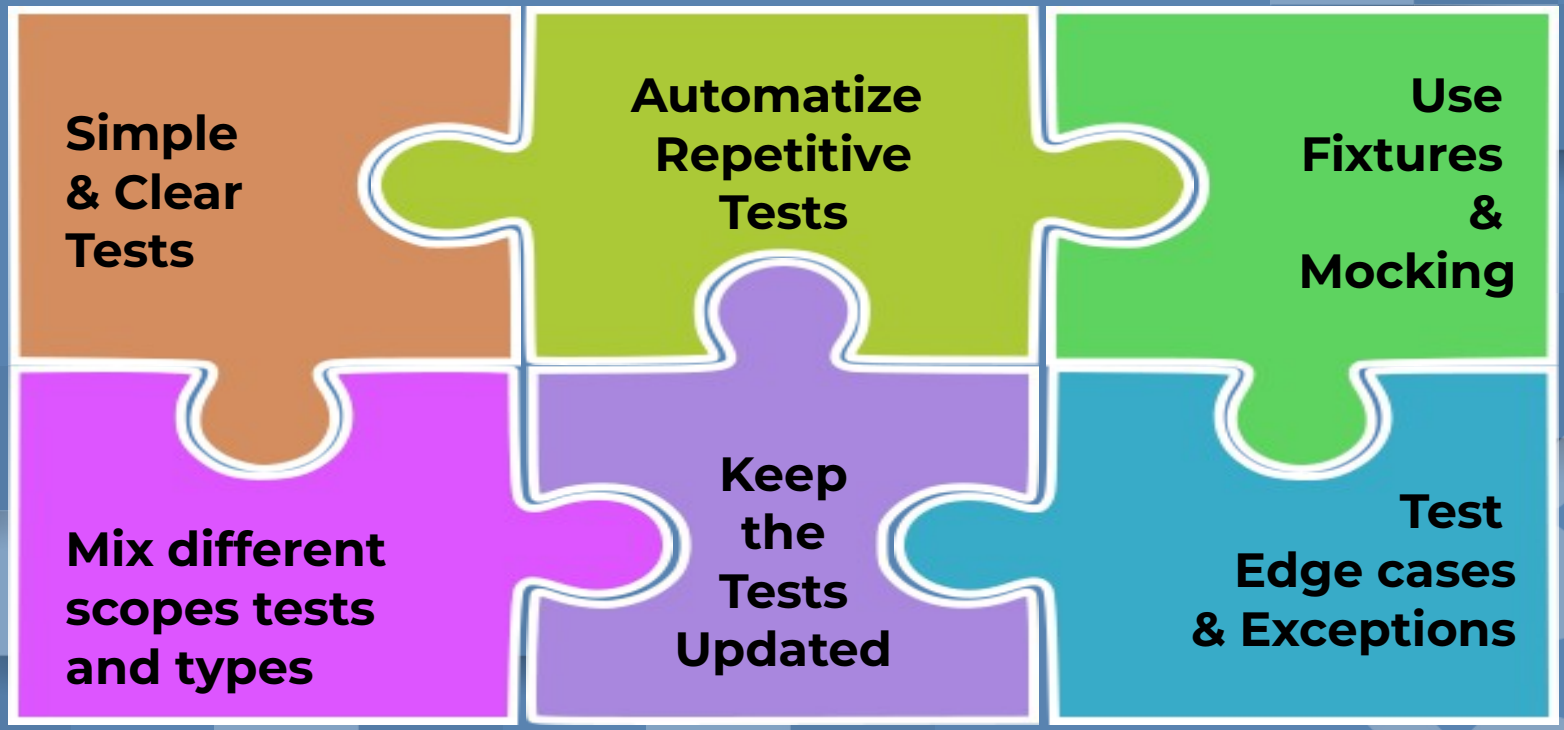
Refactor tests when code changes.
Eliminate redundant or unnecessary tests.
Review code coverage to identify untested areas.

Best practices

Bugs usually appear in extreme or unexpected scenarios.

Test cases such as null entries, negative numbers or out-of-range values.
Simulates network errors, database failures or unavailable resources.

Best practices



Testing

Importance

Strategies

- TDD
- BDD

Tests types

- Unitarios
- Funcionales
- de Integración
- de Rendimiento
- de Usabilidad

Libraries

- UnitTest
- Pytest
- Pytest-BDD
- Unitest.mock Pytest-Mock

Best practices

- Test simples y claros
- Automatizar
- Fixtures y parametros
- Test en capas
- Mantenimiento
- Casos límite

Thank you

mavignau

marian-vignau

mavignau.gitlab.io