

El Zen del Testing en Python



Testing

Importancia

Estrategias

TDD

BDD

Alcance

Unitarios

Funcionales

de Integración

de Rendimiento

de Usabilidad

Patch y Mock

Bibliotecas

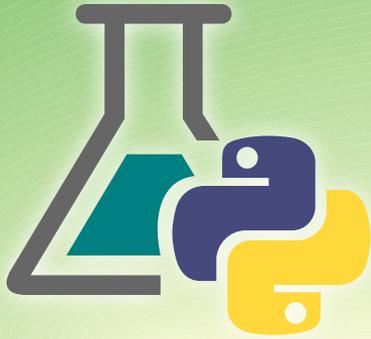
UnitTest

Pytest

Pytest-BDD

Unitest.mock Pytest-Mock

Mejores prácticas



Testing automatizado

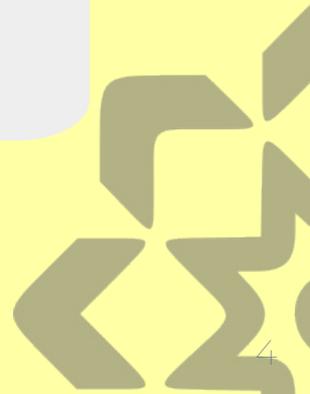
Ventajas del testing

Seguridad para mantener

- optimizar -

- refactorizar -

Documentación viva del código



Ventajas del testing

Aumenta la
productividad

Mayor **disponibilidad**

Ventajas del testing

Calidad: identificación temprana de errores

Fomenta diseño **modular**

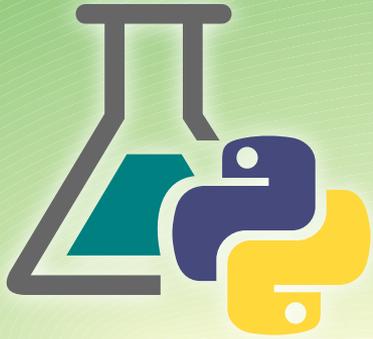
Ventajas del testing

+ trabajable en **equipo**

+ **auditable**

+ **sólido** y de **calidad**





Estrategias de desarrollo

Test driven design

Primero escribir tests
Luego el **código**
Optimizar

Test driven design

1 Crear tests

Pasar los tests **2**

3 Refactorizar

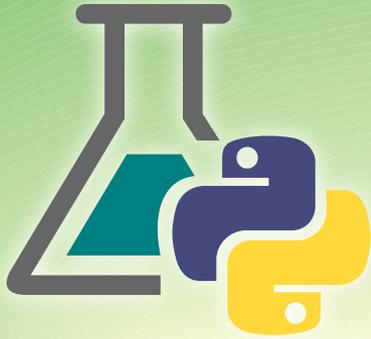




Describir la
funcionalidad esperada
de forma legible
para no técnicos
¡y usarla para tests!

Behavior driven design

Los tests se escriben
como **casos de uso**.
Sirven como
documentación.



Alcance de los tests

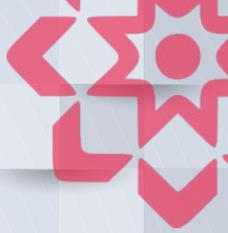


Test unitario

Prueban una única **unidad de código** (función, método) de forma **independiente**.

Ejemplo

Comprobar si una función para sumar números devuelve el resultado esperado.



Test funcional

Verifican si una **funcionalidad** del sistema anda bien.

Caja negra: No se conocen los detalles de implementación.

Ejemplo

Asegurarse de que un formulario de registro guarda los datos del usuario.



Test de integración

Verifican que los módulos o componentes se **coordinen** correctamente.

Ejemplo

Comprobar que la autenticación funciona contra la base de datos.

Alcance de tests

Test unitario

Comprobar piezas sueltas



Test funcionalidad

Verificar funciones como acelerar

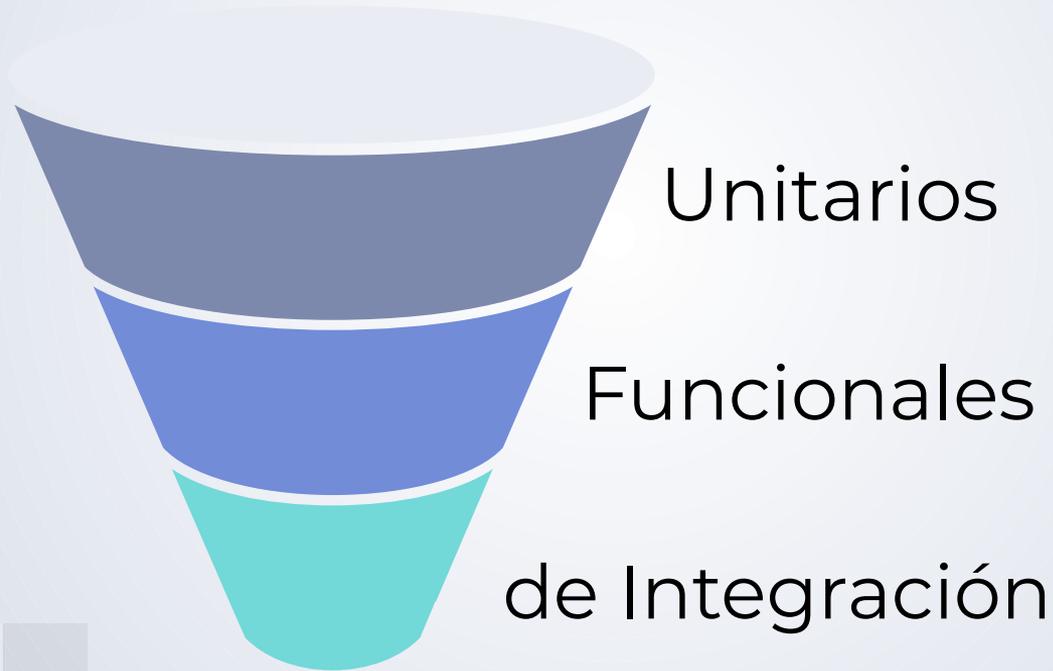


Test de integración

Asegurar que todo el auto trabaje junto



Tipos de test





Test de rendimiento

Evaluar la velocidad,
escalabilidad y
capacidad de
respuesta.

Ejemplo

Probar el sistema con el
más de usuarios que los
habituales para ver su
performance



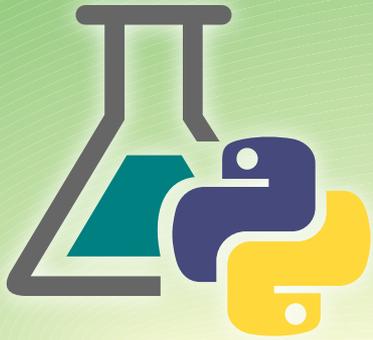
Test de usabilidad

Verificar si la aplicación es fácil de usar.

Ejemplo

Generar pruebas con usuarios finales.

Realizar A/B testing.



Bibliotecas

Bibliotecas

Unittest

Incluida en Python

Pytest

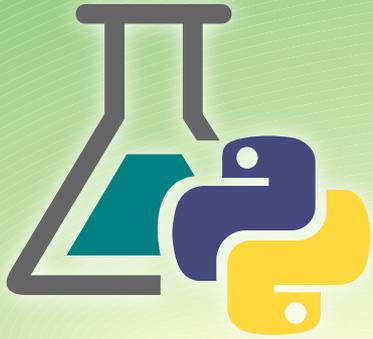
Muy flexible, expresiva y potente

Pytest-BDD Behave

Behavior Driven Design

Unittest.mock Pytest-mock

Crear objetos falsos de prueba



Unittest

Unittest: incluye

Pruebas **unitarias**.

Configuración y limpieza.

Verificación de
resultados esperados.



UnitTest

Hereda de

TestCase

comprueba con

assertNNN

```
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

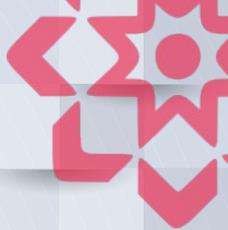
TestSuite

`unittest.TestCase`

`setUp`

`test_*`

`tearDown`



UnitTest

SetUp

Prepara entorno

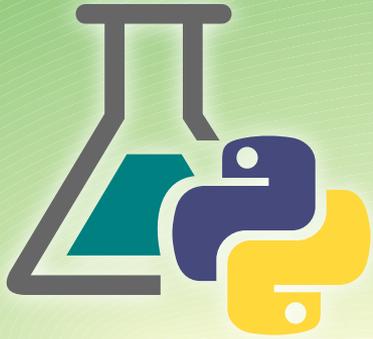
TearDown

Limpia el entorno

```
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**

Sintaxis más **limpia**

Usa **assert**
directamente

Soporte nativo para
fixtures y **parametrize**

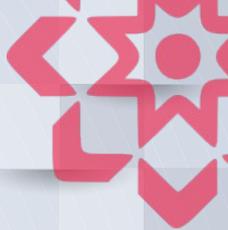
Extensible con plugins

Verbosidad en métodos

Métodos como
assertEqual

Necesita configuración
manual

No acepta funciones



PyTest

las funciones se llaman

`test_ ...`

comprueba con

`assert`

```
import pytest

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

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

PyTest

Las **fixtures** son muy **prácticas**

Automáticamente usadas con **autouse**

Diferentes **alcances**:

Del **módulo**, del **paquete**
o **globales**

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

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



PyTest

Parametrización:

Ejecuta una misma prueba con múltiples valores.

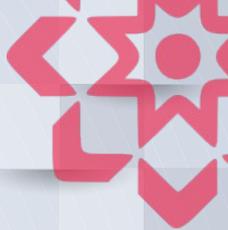
```
@pytest.mark.parametrize("x, y, esperado", [
    (1, 1, 2), (2, 3, 5),
])
def test_suma(x, y, esperado):
    assert x + y == esperado
```

PyTest

Manejo de Errores

Verificar **excepciones** con `pytest.raises`

```
def dividir(a, b):  
    return a / b  
  
def test_division_por_cero():  
    with pytest.raises(ZeroDivisionError):  
        dividir(1, 0)
```



PyTest: Plugins

pytest-cov: Medir cobertura de código

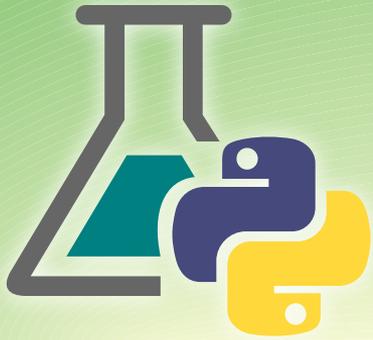
pytest-django: Pruebas en proyectos Django.

pytest-flask: Pruebas en proyectos Flask.

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

Demo 

PyTest



Pytest-BDD

Behavior driven design

Feature

Scenario

Given

When

Then

Behavior driven design: Lenguaje Gherkin

Palabras clave:

Feature: Describe una **funcionalidad**.

Scenario: Un **caso** específico de uso.

Given: Estado **inicial**.

When: **Acción**.

Then: **Resultado** esperado.

Behavior driven design

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

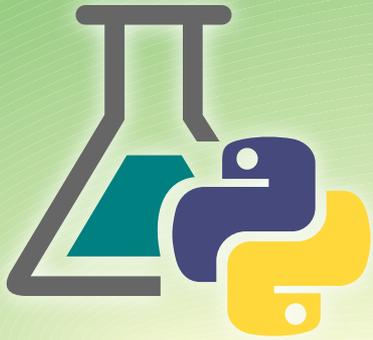
@given("tengo los números 2 y 3", target_fixture="numeros")
def numeros():
    return 2, 3

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

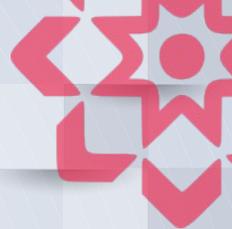
@then("el resultado debe ser 5")
def verificar(sumar):
    assert sumar == 5
```

Demo 

PyTest-BDD



Patch y mock

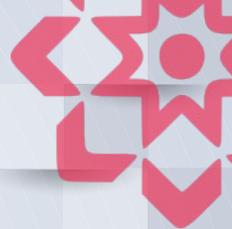


Mock

Copia simulada de un objeto para **aislar una parte del sistema** en las pruebas.

Ejemplo

Crear un sensor simulado que pueda dar todo tipo de valores



Patch

Reemplazar

temporalmente
un **objeto real por**
un mock para
simular

Ejemplo

Usar un sensor
simulado para
probar el velocímetro

Patch y Mock

```
from unittest.mock import patch
from modulo import obtener_datos

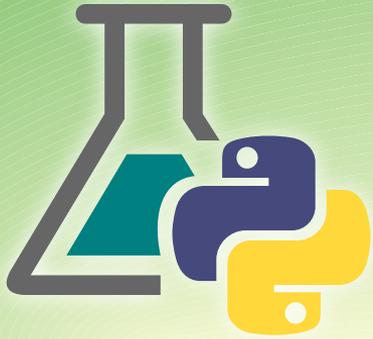
@patch("modulo.requests.get")
def test_obtener_datos(mock_get):
    mock_get.return_value.json.return_value = {"clave": "valor"}

    resultado = obtener_datos()

    assert resultado == {"clave": "valor"}
    mock_get.assert_called_once_with("https://api.example.com/data")
```

Demo

Patch y Mock



Mejores prácticas

Mejores prácticas



Pruebas
Simples
y Claras

Pruebas **simples** son fáciles de mantener y depurar.

- Prueba una sola cosa por test (principio Arrange-Act-Assert).
- Usa nombres **descriptivos** para los tests
- Evita lógica **compleja** dentro de los tests.

Mejores prácticas



Automatiza
Pruebas
Repetitivas

Automatizar reduce errores humanos y asegura consistencia.

- Utiliza herramientas como pytest o unittest.
- Configura un pipeline de **integración continua** (CI/CD) para ejecutar pruebas automáticamente con cada cambio de código.

Mejores prácticas



Las **pruebas aisladas** son más rápidas, estables y fáciles de depurar.

- Usa **fixtures** para preparar y limpiar datos o configuraciones.
- Utiliza **mocking** para simular interacciones con APIs externas, bases de datos u otros sistemas externos.

Mejores prácticas



Pruebas en Capas

Diferentes **tipos de pruebas** aseguran calidad en múltiples niveles.

- Pruebas **unitarias**: funciones individuales
- Pruebas de **funcionalidad**: interacciones entre módulos.
- Pruebas **integración**: flujo completo de la aplicación.

Mejores prácticas



Los tests **obsoletos** generan ruido y confusión.

- **Refactoriza** las pruebas cuando cambie el código.
- **Elimina** pruebas redundantes o innecesarias.
- **Revisa la cobertura de código** para identificar áreas no probadas.

Mejores prácticas



Los bugs suelen aparecer en **escenarios extremos o inesperados.**

- **Prueba casos** como entradas nulas, números negativos o valores fuera de rango.
- **Simula errores** de red, fallos de bases de datos o recursos no disponibles.

Mejores prácticas



Testing

Importancia

Estrategias

TDD

BDD

Alcance

Unitarios

Funcionales

de Integración

de Rendimiento

de Usabilidad

Patch y Mock

Bibliotecas

UnitTest

Pytest

Pytest-BDD

Unitest.mock Pytest-Mock

Mejores prácticas