CLASE 17

Testing con pytest

"El codigo sin tests es codigo roto por diseno"

Hoy aprenderan:

Por que testear? Beneficios reales

Instalar y configurar pytest

Escribir tu primer test

Assertions: verificar que el codigo funciona

TIP: Testing = Seguridad de que tu codigo funciona

Por que testear tu codigo?

Sin tests:

"Funciona en mi maquina"

Miedo a cambiar codigo existente

Bugs aparecen en produccion

Horas debuggeando

Con tests:

Confianza al hacer cambios

Detectar bugs antes

Documentacion viva del codigo

Dormir tranquilo

Ejemplo real:

```
                        Cambias una funcion  -->  Corres tests  -->
PASAN  = Todo bien
FALLAN = Algo rompiste (pero lo sabes AHORA, no en produccion)
```

TIP: En entrevistas de trabajo, saber testing te diferencia de otros candidatos.

Que es pytest?

pytest = Framework de testing mas popular en Python

Por que pytest y no unittest?

pytest

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

Simple y directo

unittest (mas complicado)

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

Mas boilerplate

Otras ventajas de pytest:

- Mensajes de error mas claros y detallados
- Descubre tests automaticamente
- Miles de plugins disponibles
- Usado en la industria (Google, Mozilla, Dropbox...)

Instalacion y estructura

1. Instalar pytest:

```
pip install pytest
```

2. Verificar instalacion:

```
pytest --version
```

3. Estructura de proyecto:

```
mi_proyecto/
    calculadora.py        # Tu codigo
    test_calculadora.py   # Tus tests
              ^
     Archivo empieza con test_
```

4. Ejecutar tests:

```
  pytest                    # Todos los tests
pytest test_archivo.py    # Un archivo
pytest -v                 # Modo verbose
```

TIP: Las funciones de test tambien deben empezar con test_

IMPORTANTE: El archivo de tests DEBE empezar con test_ o pytest no lo encuentra.

Tu primer test

Archivo: calculadora.py

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

def resta(a, b):
    return a - b
```

assert = 'afirmo que esto es verdad'

```
   assert suma(2, 3) == 5   # Afirmo: 2+3 es 5

# Si es True  -> test PASA
# Si es False -> test FALLA
```

Archivo: test_calculadora.py

```
from calculadora import suma, resta

def test_suma_positivos():
    assert suma(2, 3) == 5

def test_suma_con_cero():
    assert suma(5, 0) == 5

def test_resta_basica():
    assert resta(10, 4) == 6
```

```
# En la terminal:
pytest test_calculadora.py -v
```

TIP: assert es una palabra clave de Python. Si la condición es False, lanza AssertionError.

Cuando un test falla

Que pasa si escribimos un test incorrecto?

```
                                        def test_suma_mal():
    assert suma(2, 3) == 6   # Incorrecto! 2+3=5
```

pytest nos muestra EXACTAMENTE que fallo:

```
                              FAILED test_calculadora.py::test_suma_mal

    def test_suma_mal():
>       assert suma(2, 3) == 6
E       assert 5 == 6
E        +  where 5 = suma(2, 3)

======= 1 failed, 3 passed in 0.03s =======
```

> Linea que fallo

E Que esperaba vs que obtuvo

TIP: pytest -v muestra cada test individual con su resultado

Anatomia de un test

```
def test_suma_positivos():        # Nombre descriptivo
    # ARRANGE (Preparar)
    a = 5
    b = 3

    # ACT (Actuar)
    resultado = suma(a, b)

    # ASSERT (Verificar)
    assert resultado == 8
```

Patron AAA:

ARRANGE - Preparar datos de entrada

ACT - Ejecutar la funcion

ASSERT - Verificar el resultado

Nombres descriptivos para tests:

```
                          test_suma_con_cero          # Que estoy testeando
test_division_por_cero    # Caso especifico
test_email_sin_arroba     # Condicion del test
```

TIP: Un buen nombre de test describe QUE se testea y BAJO QUE condicion.

Testear excepciones

A veces queremos verificar que una funcion LANCE un error

La funcion:

```
def dividir(a, b):
    if b == 0:
        raise ValueError("No dividir por 0")
    return a / b
```

El test:

```
import pytest

def test_division_por_cero():
    with pytest.raises(ValueError):
        dividir(10, 0)
```

Tambien puedes verificar el mensaje del error:

```
def test_division_por_cero_mensaje():
    with pytest.raises(ValueError) as exc_info:
        dividir(10, 0)
    assert "cero" in str(exc_info.value)
```

TIP: pytest.raises() verifica que SI se lance la excepcion esperada. Si NO se lanza, el test FALLA.

Multiples assertions

Opcion 1: Varios asserts (funciona, pero...)

```
                                    def test_calculadora_completa():
    assert suma(2, 2) == 4
    assert suma(0, 0) == 0      # No se ejecuta
    assert suma(-1, 1) == 0     # si el de arriba falla
```

IMPORTANTE: Si un assert falla, los siguientes NO se ejecutan.

Opcion 2: Un test por caso (MEJOR)

```
                                    def test_suma_positivos():
    assert suma(2, 2) == 4

def test_suma_ceros():
    assert suma(0, 0) == 0

def test_suma_mixtos():
    assert suma(-1, 1) == 0
```

Mejor practica: un test = un caso especifico. Asi sabes EXACTAMENTE que fallo.

EJERCICIO 1: Escribe tus primeros tests

Dado este modulo validadores.py:

```
def es_par(n):
    return n % 2 == 0

def es_positivo(n):
    return n > 0

def es_adulto(edad):
    return edad >= 18
```

```
pytest test_validadores.py -v
```

Tu mision - Crear test_validadores.py:

1. test_es_par_con_par

   es_par(4) debe ser True

2. test_es_par_con_impar

   es_par(7) debe ser False

3. test_es_positivo_positivo

   es_positivo(5) es True

4. test_es_positivo_negativo

   es_positivo(-3) es False

5. test_es_adulto_mayor

   es_adulto(25) es True

Pistas:

```
from validadores import es_par, es_positivo, es_adulto
Cada test es: assert funcion(valor) == resultado_esperado
```

Equipos de 2-3 | Tiempo: 10 minutos | BONUS: Agrega test para es_par(0)

BREAK

10 MINUTOS

Volvemos con assertions avanzados y casos edge

Tipos de assertions

Igualdad

```
assert resultado == esperado
assert resultado != otro_valor
```

Booleanos

```
assert es_valido          # True
assert not es_invalido    # False
```

Contenido

```
assert "hola" in texto
assert "error" not in mensaje
assert item in lista
```

Comparaciones

```
assert edad >= 18
assert precio < 100
assert len(lista) > 0
```

Tipo

```
assert isinstance(resultado, list)
assert isinstance(edad, int)
```

Excepciones

```
with pytest.raises(ValueError):
    funcion_que_falla()

with pytest.raises(TypeError):
    funcion_tipo_malo()
```

TIP: assert es simplemente: 'si esta condicion es True, todo bien. Si es False, falla.'

Casos edge (borde)

Que son? Valores limite donde el codigo puede fallar

Para una funcion suma(a, b):

```
                    Casos normales:
  suma(2, 3)              suma(10, 20)

Casos edge:
  suma(0, 0)              # Ceros
  suma(-5, -3)            # Negativos
  suma(999999, 1)         # Muy grandes
```

Para validar_edad(edad):

```
                    Casos edge:
  edad = 0        # Limite inferior
  edad = 17       # Justo ANTES de adulto
  edad = 18       # EXACTAMENTE adulto
  edad = -1       # Invalido
  edad = 150      # Limite superior?
```

Checklist de casos edge:

```
                  Cero (0)                  - Siempre testear con cero
  Negativos (-1, -100)      - Numeros negativos rompen muchas funciones
  Vacios ('', [], {})       - Strings vacios, listas vacias
  Limites (17, 18, 19)      - Valores JUSTO en el borde
  None                      - El valor nulo de Python
  Tipos inesperados         - Pasar string donde esperan int
```

TIP: Los bugs casi siempre estan en los casos edge, no en los normales.

Organizar tests con clases

Sin clase (funciona bien para pocos tests):

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

def test_suma_negativos():
    assert suma(-1, -1) == -2
```

Con clase (agrupa tests relacionados):

```
class TestSuma:
    def test_positivos(self):
        assert suma(2, 3) == 5

    def test_negativos(self):
        assert suma(-1, -1) == -2

class TestResta:
    def test_basica(self):
        assert resta(5, 3) == 2
```

Reglas para clases de test:

- La clase empieza con Test (TestSuma, TestResta)
- Los metodos empiezan con test_ (test_positivos)
- Cada metodo recibe self como primer parametro
- No necesitan __init__ (pytest maneja todo)

Cuando usar cual?

Pocos tests (5-10) -> Funciones sueltas esta bien
Muchos tests (10+) -> Clases para organizar por funcionalidad
Ambos son validos! Usen lo que les resulte mas comodo.

EJERCICIO 2: Tests completos con casos edge

Funcion a testear (descuento.py):

```
def calcular_descuento(precio, porcentaje):
    """Porcentaje debe ser 0-100. Precio >= 0."""
    if precio < 0:
        raise ValueError("Precio no puede ser negativo")
    if porcentaje < 0 or porcentaje > 100:
        raise ValueError("Porcentaje debe ser 0-100")
    return precio * (1 - porcentaje / 100)
```

Tests a crear:

Caso normal (100, 20) -> 80
Sin descuento (porcentaje=0)
Descuento total (porcentaje=100) -> 0
Precio negativo -> ValueError
Porcentaje >100 -> ValueError
Porcentaje <0 -> ValueError

Pistas:

```
import pytest
from descuento import calcular_descuento
```

Para excepciones:

```
with pytest.raises(ValueError):
    calcular_descuento(-50, 10)
```

Equipos de 2-3 | Tiempo: 15 minutos | BONUS: Agrega test para precio=0

Revisemos la solucion

[El instructor muestra el codigo en VS Code]

Puntos clave:

Caso normal

```
assert calcular_descuento(100, 20) == 80
```

Casos limite (edge)

```
calcular_descuento(100, 0)   == 100  # Sin desc
calcular_descuento(100, 100) == 0    # Gratis
```

Excepciones con pytest.raises

```
with pytest.raises(ValueError):
    calcular_descuento(-50, 10)
```

Nombres descriptivos

```
test_descuento_normal
test_descuento_cero
test_precio_negativo_lanza_error
```

Output esperado:

```
                                    $ pytest test_descuento.py -v
test_descuento.py::TestCalcularDescuento::test_descuento_normal    PASSED
test_descuento.py::TestCalcularDescuento::test_sin_descuento       PASSED
test_descuento.py::TestCalcularDescuento::test_descuento_total     PASSED
test_descuento.py::TestCalcularDescuento::test_precio_negativo     PASSED
========= 8 passed in 0.02s =========
```

Buenas practicas de testing

HACER:

Un test = un caso especifico
Nombres descriptivos (test_suma_con_negativos)
Testear casos edge primero
Ejecutar tests frecuentemente
Tests independientes (no dependen entre si)

EVITAR:

Tests que dependen de otros tests
Tests que modifican archivos reales
Tests con print() para verificar
Ignorar tests que fallan
Tests sin assert (no verifican nada)

Cobertura recomendada:

Funciones criticas (pagos, auth): 100%
Funciones normales: 80%+
Codigo UI/visual: opcional

TIP: Escribe el test ANTES de arreglar un bug. Asi verificas que el fix funciona.

Resumen - Testing con pytest

Lo que aprendieron hoy:

Concepto	Comando / Codigo
Instalar pytest	`pip install pytest`
Ejecutar tests	`pytest -v`
Test basico	`assert suma(2,3) == 5`
Testear excepciones	`with pytest.raises(ValueError):`
Patron de test	`Arrange -> Act -> Assert`
Casos edge	`Cero, negativos, limites, vacios`
Buena practica	`Un test = un caso especifico`

Proxima clase: Clase 18 - Intro a OOP (Clases y Objetos)

Van a crear sus propias clases y objetos. Testing les va a servir para verificarlos!

Tarea: Escribir 5 tests para una funcion que crearon en clases anteriores