Sesión 2: Introducción a pruebas con Pytest

Duración: 45 minutos

🎯 Objetivos de aprendizaje

Al finalizar esta sesión, el participante será capaz de:

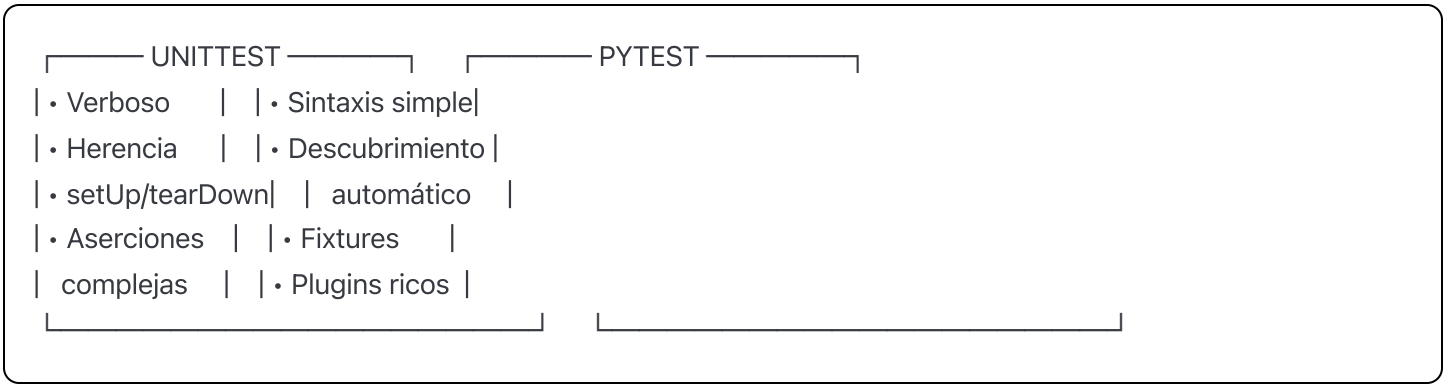
- Comprender la importancia de las pruebas automatizadas
- Instalar y configurar Pytest en un proyecto Python
- Escribir pruebas básicas y avanzadas con Pytest
- Utilizar fixtures para preparar datos de prueba
- Ejecutar y analizar resultados de pruebas
- Implementar pruebas parametrizadas y marcadores

📋 Contenido

1. ¿Por qué Pytest? (5 min)

Analogía: Las pruebas son como los cinturones de seguridad en un auto. No los necesitas hasta que los necesitas, pero cuando llega el momento, te salvan la vida.

Ventajas de Pytest sobre unittest:



¿Por qué elegir Pytest?

- Sintaxis más limpia y pythónica
- Mejor manejo de errores y debugging
- Sistema de fixtures potente
- Ejecución paralela nativa

- Ecosistema de plugins extenso

2. Instalación y Configuración (5 min)

Instalación básica:

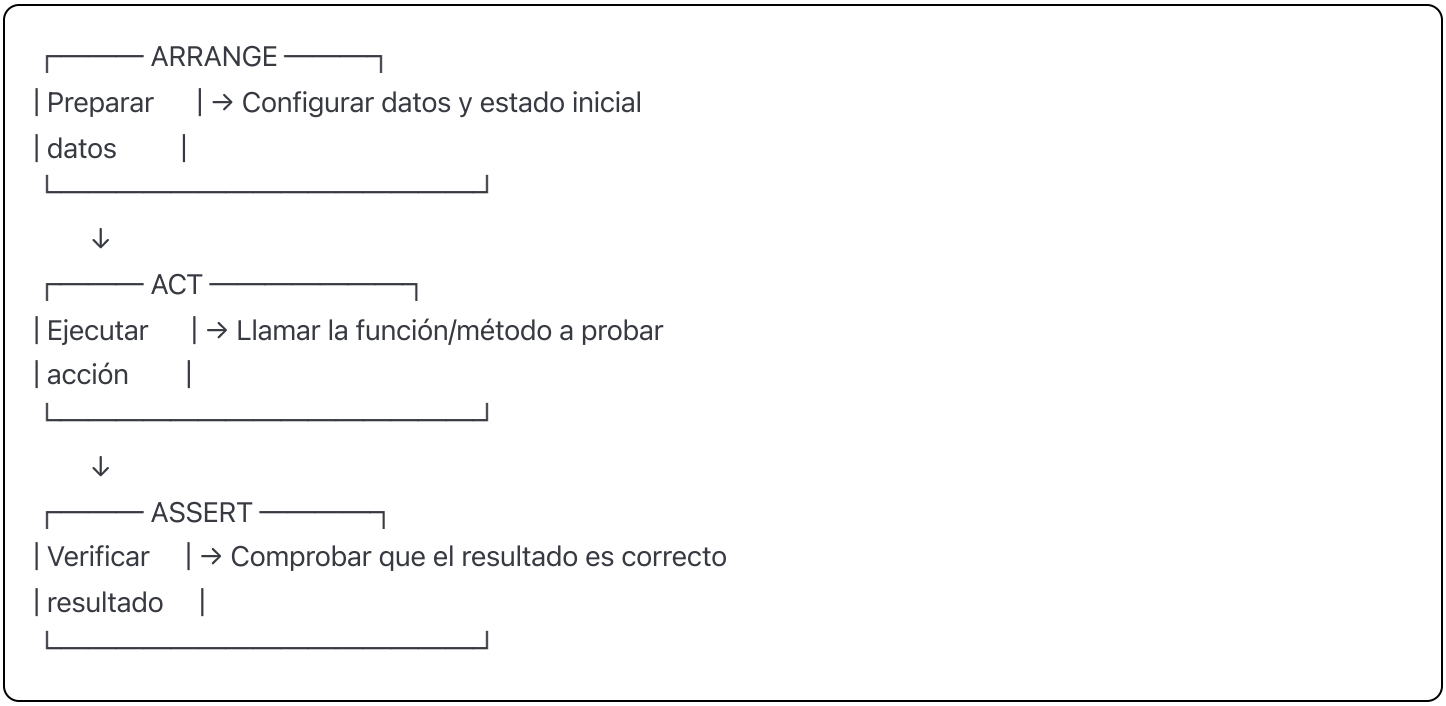
```bash
# Instalación simple
pip install pytest

# Con plugins esenciales
pip install pytest pytest-cov pytest-mock pytest-xdist
```

Estructura de proyecto recomendada:

```
mi_proyecto/
├── src/
│   └── calculadora.py
├── tests/
│   ├── __init__.py
│   ├── conftest.py
│   └── test_calculadora.py
├── pytest.ini
└── requirements.txt
```

3. Anatomía de una Prueba (10 min)

Los 3 pilares de una prueba (AAA Pattern):



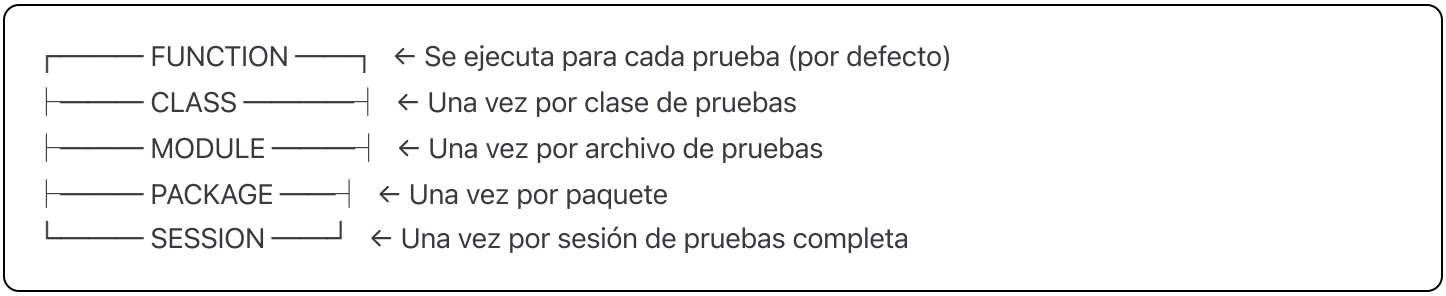
Ejemplo básico:

```python
def test_suma_basica():
    # Arrange
    a, b = 2, 3
    calculadora = Calculadora()

    # Act
    resultado = calculadora.sumar(a, b)

    # Assert
    assert resultado == 5
```

4. Fixtures - La Base de Datos de Pruebas (10 min)

¿Qué son las fixtures? Las fixtures son funciones que proporcionan datos o recursos para las pruebas de manera consistente y reutilizable.

Scopes de fixtures:



Ejemplo de fixtures:

```python
@pytest.fixture
def calculadora():
    """Fixture que proporciona una instancia de calculadora"""
    return Calculadora()

@pytest.fixture
def datos_complejos():
    """Fixture con datos de prueba complejos"""
    return {
        'numeros_positivos': [1, 2, 3, 4, 5],
        'numeros_negativos': [-1, -2, -3],
        'decimales': [1.5, 2.7, 3.14159]
    }
```

5. Aserciones Avanzadas (5 min)

Pytest vs Assert tradicional:

```python
# ❌ Assert tradicional (poco informativo)
assert resultado == esperado

# ✅ Pytest (información detallada automática)
assert resultado == esperado
# Si falla: AssertionError: assert 8 == 5
#        -5
#        +8
```

Tipos de aserciones comunes:

```python
```

```python
# Igualdad y comparaciones
assert a == b
assert a != b
assert a > b

# Contenido
assert item in lista
assert "substring" in texto

# Excepciones
with pytest.raises(ValueError):
    funcion_que_debe_fallar()

# Aproximaciones (para floats)
assert abs(resultado - esperado) < 0.001
# O mejor:
assert resultado == pytest.approx(esperado)
```

6. Pruebas Parametrizadas (5 min)

¿Por qué parametrizar? Evita duplicación de código cuando necesitas probar la misma lógica con diferentes datos.

Estructura de parametrización:

```python
@pytest.mark.parametrize("entrada,esperado", [
    (2, 4),     # 2² = 4
    (3, 9),     # 3² = 9
    (4, 16),    # 4² = 16
    (0, 0),     # 0² = 0
])
def test_cuadrado(calculadora, entrada, esperado):
    assert calculadora.cuadrado(entrada) == esperado
```

7. Marcadores (Markers) (3 min)

Organización de pruebas:

```python
```

```
@pytest.mark.slow
def test_operacion_lenta():
    pass

@pytest.mark.unit
def test_suma_unitaria():
    pass

@pytest.mark.integration
def test_integracion_completa():
    pass
```

Ejecución selectiva:

```bash
# Solo pruebas rápidas
pytest -m "not slow"

# Solo pruebas unitarias
pytest -m unit

# Combinaciones
pytest -m "unit and not slow"
```

8. Comandos Esenciales (2 min)

```bash
```

```
# Ejecutar todas las pruebas
pytest

# Verbosidad aumentada
pytest -v

# Mostrar print() en pruebas
pytest -s

# Parar en primer fallo
pytest -x

# Ejecutar pruebas específicas
pytest tests/test_calculadora.py::test_suma

# Cobertura de código
pytest --cov=src

# Reporte HTML de cobertura
pytest --cov=src --cov-report=html
```

⚠️ Buenas Prácticas

🎯 NAMING (Nomenclatura):

- Archivos: `test_*.py` o `*_test.py`
- Funciones: `test_*`
- Clases: `Test*`
- Nombres descriptivos: `test_suma_numeros_positivos()`

🏗️ ORGANIZACIÓN:

- Un archivo de prueba por módulo
- Agrupar pruebas relacionadas en clases
- Usar `conftest.py` para fixtures compartidas
- Mantener pruebas independientes entre sí

📊 COBERTURA:

- Objetivo: >80% de cobertura

- Priorizar código crítico
- No obsesionarse con 100%
- Cobertura de calidad > cantidad

🚫 Errores Comunes

❌ NO hagas esto:

- Pruebas que dependen del orden de ejecución
- Fixtures que modifican estado global
- Aserciones múltiples sin relación en una prueba
- Pruebas que duran más de unos segundos (sin marcar como `@slow`)

✅ SÍ haz esto:

- Una prueba, un concepto
- Nombres descriptivos y específicos
- Fixtures con scope apropiado
- Cleanup automático con `yield`

📊 Configuración Avanzada

pytest.ini:

```ini
```

```
[tool:pytest]
testpaths = tests
python_files = test_*.py *_test.py
python_functions = test_*
python_classes = Test*
addopts =
    -v
    --strict-markers
    --disable-warnings
    --cov=src
    --cov-report=term-missing
markers =
    slow: marks tests as slow
    unit: marks tests as unit tests
    integration: marks tests as integration tests
```

📝 Resumen Final

1. **Pytest simplifica** la escritura y ejecución de pruebas
2. **Fixtures** proporcionan datos consistentes y reutilizables
3. **Parametrización** evita duplicación de código de pruebas
4. **Marcadores** organizan y permiten ejecución selectiva
5. **Configuración adecuada** mejora la experiencia de desarrollo
6. **Cobertura de código** guía pero no obsesiona
7. **Pruebas independientes** son pruebas confiables

🏃‍♂️ Actividad Práctica Sugerida

Ejercicio: Toma una clase simple (ej: Calculadora) y crea una suite completa de pruebas que incluya:

- Pruebas básicas de cada método
- Fixtures para datos de prueba
- Pruebas parametrizadas para múltiples casos
- Pruebas de excepciones
- Configuración de markers

Tiempo estimado: 20-25 minutos adicionales