Capítulo 12

Testing unitario

12.1. Introducción

El testing unitario es una práctica fundamental en el desarrollo de software que consiste en escribir y ejecutar pruebas automatizadas para verificar que las unidades individuales de código funcionen correctamente de manera aislada. Una unidad de código típicamente corresponde a un método, función o clase, y representa la porción más pequeña de código que puede ser probada de forma independiente.

El propósito principal del testing unitario es detectar errores en las primeras etapas del desarrollo, cuando son más fáciles y económicos de corregir. Al verificar que cada componente individual funciona según lo esperado, se puede tener mayor confianza en que el sistema completo funcionará correctamente cuando todas las partes se integren.

En la programación orientada a objetos, el testing unitario cobra especial relevancia debido a la naturaleza modular y encapsulada de las clases. Cada clase debe cumplir con un conjunto específico de responsabilidades y mantener su estado interno de manera coherente. Las pruebas unitarias permiten verificar que los métodos de una clase produzcan los resultados esperados, que las validaciones internas funcionen correctamente, y que el encapsulamiento se mantenga adecuadamente.

Los beneficios del testing unitario incluyen la detección temprana de errores, la facilitación del refactoring seguro del código, la documentación implícita del comportamiento esperado de cada componente, y la reducción del tiempo necesario para localizar y corregir defectos. Además, las pruebas unitarias actúan como una red de seguridad que permite a los desarrolladores modificar el código con confianza, sabiendo que cualquier regresión será detectada inmediatamente.

Para que el testing unitario sea efectivo, las pruebas deben ser rápidas de ejecutar, independientes entre sí, repetibles y fáciles de mantener. Cada prueba debe verificar un comportamiento específico y proporcionar información clara sobre qué aspecto del código falló cuando ocurre un error.

12.2. Pytest

Pytest es un framework de testing para Python que se ha convertido en el estándar de facto para escribir y ejecutar pruebas unitarias en este lenguaje. A diferencia de otros frameworks de testing, pytest se destaca por su sintaxis simple y natural, su capacidad de autodescubrimiento de pruebas, y sus potentes características avanzadas que facilitan la escritura de pruebas comprehensivas.

Una de las principales ventajas de pytest es que permite escribir pruebas utilizando la sintaxis estándar de Python con la instrucción `assert`, eliminando la necesidad de aprender métodos específicos del framework como ocurre con otros sistemas de testing. Esto hace que las pruebas sean más legibles y fáciles de escribir para desarrolladores que ya conocen Python.

Para instalar pytest, se utiliza el gestor de paquetes pip mediante el siguiente comando:

```
pip install pytest
```

Una vez instalado, pytest puede ejecutarse desde la línea de comandos. El framework automáticamente descubre y ejecuta todas las pruebas en el directorio actual y sus subdirectorios. Para ejecutar las pruebas, simplemente se utiliza:

```
pytest
```

Pytest busca automáticamente archivos que comiencen con `test_` o terminen con `_test.py`, y dentro de estos archivos ejecuta todas las funciones que comiencen con `test_`. Esta convención de nomenclatura facilita la organización y el descubrimiento automático de las pruebas.

Además del comando básico, pytest ofrece múltiples opciones de ejecución. Por ejemplo, se puede ejecutar un archivo específico de pruebas, mostrar información más detallada sobre la ejecución, o ejecutar solo las pruebas que contienen una palabra clave específica:

```
# Ejecutar un archivo específico
pytest test_ejemplo.py

# Mostrar información detallada
pytest -v

# Ejecutar pruebas que contengan una palabra clave
pytest -k "suma"
```

Pytest también genera reportes claros y detallados cuando las pruebas fallan, mostrando exactamente qué assertion falló, los valores esperados versus los obtenidos, y el contexto completo del error.

12.3. Pruebas

La estructura fundamental de las pruebas en pytest se basa en funciones que contienen assertions para verificar el comportamiento esperado del código. Una suite de pruebas es simplemente un conjunto de funciones de prueba organizadas en uno o más archivos, mientras que cada caso de prueba individual corresponde a una función específica que verifica un aspecto particular del código bajo prueba.

La sintaxis básica para escribir una prueba en pytest es crear una función cuyo nombre comience con `test_` y utilizar la instrucción **`assert`** para verificar condiciones:

```
def test_ejemplo_basico():
    resultado = 2 + 2
    assert resultado == 4
```

Para organizar las pruebas relacionadas con una clase específica, se recomienda crear un archivo de pruebas separado, típicamente con el nombre `test_nombre_clase.py`. Dentro de este archivo, se pueden agrupar las pruebas utilizando clases de prueba, aunque esto no es obligatorio:

```
class TestCalculadora:
    def test_suma(self):
        calc = Calculadora()
        resultado = calc.sumar(2, 3)
        assert resultado == 5

    def test_division_por_cero(self):
        calc = Calculadora()
        with pytest.raises(ZeroDivisionError):
            calc.dividir(10, 0)
```

12.3.1. Aserciones de igualdad

Las aserciones de igualdad son las más comunes en las pruebas unitarias y verifican que un valor obtenido sea exactamente igual al valor esperado. Pytest utiliza la instrucción **`assert`** estándar de Python junto con el operador de igualdad:

```
def test_igualdad_numerica():
    assert 5 == 5
    assert 2.5 == 2.5
```

```
def test_igualdad_cadenas():
    assert "hola" == "hola"
    assert "Python".lower() == "python"

def test_igualdad_listas():
    assert [1, 2, 3] == [1, 2, 3]
    assert sorted([3, 1, 2]) == [1, 2, 3]
```

Para la desigualdad, se utiliza el operador $\neq$:

```
def test_desigualdad():
    assert 5 ≠ 3
    assert "hola" ≠ "adiós"
```

12.3.2. Aserciones de comparación

Las aserciones de comparación permiten verificar relaciones de orden entre valores utilizando los operadores de comparación estándar de Python:

```
def test_comparaciones_numericas():
    assert 5 > 3
    assert 2 < 10
    assert 5 ≥ 5
    assert 3 ≤ 7

def test_comparaciones_cadenas():
    assert "abc" < "def"  # Orden alfabético
    assert "z" > "a"
```

12.3.3. Aserciones de pertenencia

Las aserciones de pertenencia verifican si un elemento está contenido en una colección utilizando el operador `in`:

```
def test_pertenencia_listas():
    numeros = [1, 2, 3, 4, 5]
    assert 3 in numeros
    assert 6 not in numeros

def test_pertenencia_cadenas():
    texto = "Hola mundo"
    assert "mundo" in texto
    assert "Python" not in texto

def test_pertenencia_diccionarios():
    datos = {"nombre": "Juan", "edad": 25}
    assert "nombre" in datos
    assert "apellido" not in datos
```

12.3.4. Aserciones de tipo

Las aserciones de tipo verifican que un objeto sea de un tipo específico utilizando la función `isinstance()`:

```
def test_tipos():
    assert isinstance(5, int)
    assert isinstance(3.14, float)
    assert isinstance("texto", str)
    assert isinstance([1, 2, 3], list)
    assert isinstance({"a": 1}, dict)
```

12.3.5. Aserciones de excepciones

Para verificar que el código lance excepciones bajo ciertas condiciones, pytest proporciona el context manager `pytest.raises()`:

```python
import pytest

def test_excepcion_division_cero():
    with pytest.raises(ZeroDivisionError):
        resultado = 10 / 0

def test_excepcion_valor_incorrecto():
    with pytest.raises(ValueError):
        int("texto_invalido")

def test_excepcion_con_mensaje():
    with pytest.raises(ValueError, match="invalid literal"):
        int("abc")
```

12.3.6. Aserciones de aproximación

Para valores de punto flotante, donde la comparación exacta puede fallar debido a imprecisiones de representación, pytest ofrece `pytest.approx()`:

```python
import pytest

def test_aproximacion():
    assert 0.1 + 0.2 == pytest.approx(0.3)
    assert 3.14159 == pytest.approx(3.14, rel=1e-2)
```

12.4. Ejemplo

Para ilustrar la aplicación práctica del testing unitario con pytest, se presenta una clase `Numeros` que encapsula una colección de números y proporciona métodos para realizar operaciones estadísticas básicas. Esta clase implementa el principio de composición al contener una lista de números como atributo interno.

```python
class Numeros:
    def __init__(self):
        self._numeros = []

    def agregar(self, numero):
        if not isinstance(numero, (int, float)):
            raise TypeError("Solo se pueden agregar números")
        self._numeros.append(numero)

    def cantidad(self):
        return len(self._numeros)

    def suma(self):
        return sum(self._numeros)

    def promedio(self):
        if self.cantidad() == 0:
            raise ValueError("No se puede calcular el promedio de una colección vacía")
        return self.suma() / self.cantidad()

    def maximo(self):
        if self.cantidad() == 0:
            raise ValueError("No se puede obtener el máximo de una colección vacía")
        return max(self._numeros)

    def minimo(self):
        if self.cantidad() == 0:
            raise ValueError("No se puede obtener el mínimo de una colección vacía")
        return min(self._numeros)

    def contiene(self, numero):
        return numero in self._numeros

    def limpiar(self):
        self._numeros.clear()
```

A continuación se presenta una suite completa de pruebas para esta clase, organizada de manera sistemática para cubrir todos los casos de uso posibles, incluyendo casos límite y situaciones de error:

```python
import pytest
from numeros import Numeros

class TestNumeros:

    def test_coleccion_vacia_inicial(self):
        """Verifica que una nueva instancia esté vacía"""
        coleccion = Numeros()
        assert coleccion.cantidad() == 0

    def test_agregar_numero_entero(self):
        """Verifica que se puedan agregar números enteros"""
        coleccion = Numeros()
        coleccion.agregar(5)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(5)

    def test_agregar_numero_flotante(self):
        """Verifica que se puedan agregar números flotantes"""
        coleccion = Numeros()
        coleccion.agregar(3.14)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(3.14)

    def test_agregar_numero_negativo(self):
        """Verifica que se puedan agregar números negativos"""
        coleccion = Numeros()
        coleccion.agregar(-10)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(-10)

    def test_agregar_cero(self):
        """Verifica que se pueda agregar el número cero"""
        coleccion = Numeros()
        coleccion.agregar(0)
        assert coleccion.cantidad() == 1
        assert coleccion.contiene(0)

    def test_agregar_tipo_invalido_cadena(self):
        """Verifica que se lance excepción al agregar una cadena"""
        coleccion = Numeros()
        with pytest.raises(TypeError, match="Solo se pueden agregar números"):
            coleccion.agregar("cinco")

    def test_agregar_tipo_invalido_lista(self):
        """Verifica que se lance excepción al agregar una lista"""
        coleccion = Numeros()
        with pytest.raises(TypeError):
            coleccion.agregar([1, 2, 3])

    def test_agregar_multiple_numeros(self):
        """Verifica que se puedan agregar múltiples números"""
        coleccion = Numeros()
        numeros = [1, 2, 3, 4, 5]
        for numero in numeros:
            coleccion.agregar(numero)

        assert coleccion.cantidad() == 5
        for numero in numeros:
            assert coleccion.contiene(numero)

    def test_suma_un_numero(self):
        """Verifica la suma con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(7)
        assert coleccion.suma() == 7

    def test_suma_multiples_numeros(self):
        """Verifica la suma con múltiples números"""
        coleccion = Numeros()
        coleccion.agregar(2)
        coleccion.agregar(3)
        coleccion.agregar(5)
        assert coleccion.suma() == 10

    def test_suma_numeros_negativos(self):
        """Verifica la suma con números negativos"""
        coleccion = Numeros()
```

```
        coleccion.agregar(-2)
        coleccion.agregar(5)
        coleccion.agregar(-1)
        assert coleccion.suma() == 2

    def test_suma_coleccion_vacia(self):
        """Verifica que la suma de una colección vacía sea cero"""
        coleccion = Numeros()
        assert coleccion.suma() == 0

    def test_promedio_un_numero(self):
        """Verifica el promedio con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(8)
        assert coleccion.promedio() == 8

    def test_promedio_multiples_numeros(self):
        """Verifica el promedio con múltiples números"""
        coleccion = Numeros()
        coleccion.agregar(2)
        coleccion.agregar(4)
        coleccion.agregar(6)
        assert coleccion.promedio() == 4

    def test_promedio_con_flotantes(self):
        """Verifica el promedio con números flotantes"""
        coleccion = Numeros()
        coleccion.agregar(1.5)
        coleccion.agregar(2.5)
        assert coleccion.promedio() == pytest.approx(2.0)

    def test_promedio_coleccion_vacia(self):
        """Verifica que se lance excepción al calcular promedio de colección vacía"""
        coleccion = Numeros()
        with pytest.raises(ValueError, match="No se puede calcular el promedio de una colección vacía"):
            coleccion.promedio()

    def test_maximo_un_numero(self):
        """Verifica el máximo con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(42)
        assert coleccion.maximo() == 42

    def test_maximo_multiples_numeros(self):
        """Verifica el máximo con múltiples números"""
        coleccion = Numeros()
        numeros = [3, 1, 4, 1, 5, 9, 2, 6]
        for numero in numeros:
            coleccion.agregar(numero)
        assert coleccion.maximo() == 9

    def test_maximo_numeros_negativos(self):
        """Verifica el máximo con solo números negativos"""
        coleccion = Numeros()
        coleccion.agregar(-5)
        coleccion.agregar(-2)
        coleccion.agregar(-8)
        assert coleccion.maximo() == -2

    def test_maximo_coleccion_vacia(self):
        """Verifica que se lance excepción al buscar máximo de colección vacía"""
        coleccion = Numeros()
        with pytest.raises(ValueError, match="No se puede obtener el máximo de una colección vacía"):
            coleccion.maximo()

    def test_minimo_un_numero(self):
        """Verifica el mínimo con un solo número"""
        coleccion = Numeros()
        coleccion.agregar(17)
        assert coleccion.minimo() == 17

    def test_minimo_multiples_numeros(self):
        """Verifica el mínimo con múltiples números"""
        coleccion = Numeros()
        numeros = [3, 1, 4, 1, 5, 9, 2, 6]
        for numero in numeros:
            coleccion.agregar(numero)
        assert coleccion.minimo() == 1
```

```
    def test_minimo_numeros_positivos(self):
        """Verifica el mínimo con solo números positivos"""
        coleccion = Numeros()
        coleccion.agregar(5)
        coleccion.agregar(2)
        coleccion.agregar(8)
        assert coleccion.minimo() == 2

    def test_minimo_coleccion_vacia(self):
        """Verifica que se lance excepción al buscar mínimo de colección vacía"""
        coleccion = Numeros()
        with pytest.raises(ValueError, match="No se puede obtener el mínimo de una colección vacía"):
            coleccion.minimo()

    def test_contiene_numero_presente(self):
        """Verifica que contiene devuelva True para números presentes"""
        coleccion = Numeros()
        coleccion.agregar(10)
        coleccion.agregar(20)
        coleccion.agregar(30)
        assert coleccion.contiene(20)

    def test_contiene_numero_ausente(self):
        """Verifica que contiene devuelva False para números ausentes"""
        coleccion = Numeros()
        coleccion.agregar(10)
        coleccion.agregar(20)
        assert not coleccion.contiene(15)

    def test_contiene_coleccion_vacia(self):
        """Verifica que contiene devuelva False en colección vacía"""
        coleccion = Numeros()
        assert not coleccion.contiene(5)

    def test_limpiar_coleccion(self):
        """Verifica que limpiar vacíe completamente la colección"""
        coleccion = Numeros()
        coleccion.agregar(1)
        coleccion.agregar(2)
        coleccion.agregar(3)

        assert coleccion.cantidad() == 3

        coleccion.limpiar()
        assert coleccion.cantidad() == 0
        assert not coleccion.contiene(1)
        assert not coleccion.contiene(2)
        assert not coleccion.contiene(3)

    def test_operaciones_despues_limpiar(self):
        """Verifica que las operaciones fallen después de limpiar"""
        coleccion = Numeros()
        coleccion.agregar(5)
        coleccion.limpiar()

        with pytest.raises(ValueError):
            coleccion.promedio()

        with pytest.raises(ValueError):
            coleccion.maximo()

        with pytest.raises(ValueError):
            coleccion.minimo()

    def test_flujo_completo(self):
        """Prueba de integración que verifica un flujo completo de operaciones"""
        coleccion = Numeros()

        # Agregar números
        numeros = [2, 4, 6, 8, 10]
        for numero in numeros:
            coleccion.agregar(numero)

        # Verificar todas las operaciones
        assert coleccion.cantidad() == 5
        assert coleccion.suma() == 30
        assert coleccion.promedio() == 6
```

```
        assert coleccion.maximo() == 10
        assert coleccion.minimo() == 2

        # Verificar pertenencia
        for numero in numeros:
            assert coleccion.contiene(numero)

        assert not coleccion.contiene(7)
```

Esta suite de pruebas cubre exhaustivamente todos los métodos de la clase `Numeros`, incluyendo casos normales, casos límite y situaciones de error. Las pruebas están organizadas de manera lógica, cada una verifica un comportamiento específico, y utilizan nombres descriptivos que facilitan la comprensión de qué está siendo probado.

El conjunto de pruebas incluye verificaciones para el manejo correcto de excepciones, validaciones de tipos, operaciones con colecciones vacías, y un caso de prueba de integración que verifica el flujo completo de operaciones. Esta aproximación sistemática al testing garantiza que la clase `Numeros` funcione correctamente en todas las situaciones previstas.