Ampliación: Pruebas en Python

- Ampliación: Pruebas en Python
  - Pruebas Unitarias y Dobles de Prueba en Python: Implementación Práctica
    - 1. Pruebas Unitarias con `unittest`
      - Estructura Básica de una Prueba Unitaria
    - 2. Dobles de Prueba con `unittest.mock`
      - `MagicMock` (El análogo a los Mocks de Mockito)
      - `patch` (Inyección de Mocks/Stubs)
    - Patrones Adicionales de `patch`
    - Consideraciones Finales

Pruebas Unitarias y Dobles de Prueba en Python: Implementación Práctica

Para desarrolladores familiarizados con frameworks de prueba como JUnit y bibliotecas de mocking como Mockito, la transición a Python es fluida. El ecosistema de pruebas de Python se centra principalmente en el módulo `unittest` de la biblioteca estándar, que ofrece una funcionalidad robusta para pruebas unitarias, y `unittest.mock`, fundamental para la implementación de dobles de prueba.

1. Pruebas Unitarias con `unittest`

El módulo `unittest` es la piedra angular para las pruebas unitarias en Python. Su diseño se inspira en JUnit y otros frameworks de pruebas de xUnit.

Estructura Básica de una Prueba Unitaria

Cada conjunto de pruebas se organiza en **clases de prueba** que heredan de `unittest.TestCase`. Los métodos dentro de estas clases que comienzan con `test_` son reconocidos automáticamente como **métodos de prueba**.

Ejemplo de Código bajo Prueba (CUT - Code Under Test):

Consideremos un módulo `calculadora.py` que contiene funciones aritméticas básicas:

```python
# calculadora.py

class Calculadora:
    def sumar(self, a, b):
        """Suma dos números."""
        return a + b

    def restar(self, a, b):
        """Resta dos números."""
        return a - b

    def dividir(self, a, b):
        """Divide dos números. Lanza ValueError si el divisor es cero."""
        if b == 0:
            raise ValueError("No se puede dividir por cero.")
        return a / b
```

Creando un Archivo de Pruebas:

Por convención, los archivos de prueba suelen nombrarse con el prefijo `test_` . Crearemos `test_calculadora.py` :

```python
# test_calculadora.py
import unittest
from calculadora import Calculadora

class TestCalculadora(unittest.TestCase):
    """Clase de pruebas para la clase Calculadora."""

    def setUp(self):
        """
        Método que se ejecuta ANTES de cada método de prueba.
        Inicializa una nueva instancia de Calculadora para cada prueba.
        """
        self.calc = Calculadora()

    def test_sumar(self):
        """Verifica la función sumar con diferentes escenarios."""
        self.assertEqual(self.calc.sumar(5, 3), 8)
        self.assertEqual(self.calc.sumar(-1, 1), 0)
        self.assertEqual(self.calc.sumar(0, 0), 0)

    def test_restar(self):
        """Verifica la función restar."""
        self.assertEqual(self.calc.restar(10, 4), 6)
        self.assertEqual(self.calc.restar(4, 10), -6)

    def test_dividir_numeros_enteros(self):
        """Verifica la función dividir con números enteros."""
        self.assertEqual(self.calc.dividir(10, 2), 5.0)

    def test_dividir_por_cero(self):
        """Verifica que dividir por cero levante un ValueError."""
        with self.assertRaises(ValueError):
            self.calc.dividir(10, 0)

    def tearDown(self):
        """
        Método que se ejecuta DESPUÉS de cada método de prueba.
        Utilizado para limpiar recursos si fuera necesario (ej. cerrar conexiones).
        """
        self.calc = None # Liberar referencia, aunque para este objeto simple no es crí

# Punto de entrada para ejecutar las pruebas
```

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

Métodos Clave de `unittest.TestCase`:

- **`setUp()`**: Se ejecuta antes de cada método de prueba. Ideal para inicializar objetos o preparar el entorno. Equivalente a `@BeforeEach` en JUnit 5 o `@Before` en JUnit 4.
- **`tearDown()`**: Se ejecuta después de cada método de prueba. Ideal para limpiar recursos. Equivalente a `@AfterEach` o `@After`.
- **`setUpClass(cls)`**: (Método de clase) Se ejecuta una vez antes de todas las pruebas en la clase. Equivalente a `@BeforeAll`.
- **`tearDownClass(cls)`**: (Método de clase) Se ejecuta una vez después de todas las pruebas en la clase. Equivalente a `@AfterAll`.

Métodos de Aserción (Ejemplos):

Los métodos de aserción de `unittest.TestCase` son cruciales para verificar los resultados de las pruebas.

- `self.assertEqual(a, b)`: `a == b`
- `self.assertNotEqual(a, b)`: `a != b`
- `self.assertTrue(x)`: `bool(x) is True`
- `self.assertFalse(x)`: `bool(x) is False`
- `self.assertIs(a, b)`: `a is b` (identidad de objeto)
- `self.assertIsNot(a, b)`: `a is not b`
- `self.assertIsNone(x)`: `x is None`
- `self.assertIsNotNone(x)`: `x is not None`
- `self.assertIn(member, container)`: `member in container`
- `self.assertNotIn(member, container)`: `member not in container`
- `self.assertRaises(exception, callable, *args, **kwargs)`: Verifica que `callable(*args, **kwargs)` levante la `exception` especificada. También puede usarse como un gestor de contexto (`with self.assertRaises(Exception): ...`).

Ejecución de Pruebas:

Para ejecutar las pruebas, navega al directorio que contiene `test_calculadora.py` en tu terminal y ejecuta:

```
python -m unittest test_calculadora.py
```

O simplemente:

```
python test_calculadora.py
```

Si tus pruebas están en un directorio y quieres descubrirlas automáticamente, puedes usar:

```
python -m unittest discover
```

> **Actividad**
> Realiza las actividades propuestas en el documento de pruebas de unidad de JUnit, traducidas a python.

2. Dobles de Prueba con `unittest.mock`

`unittest.mock` es la herramienta estándar de Python para crear dobles de prueba (Stubs, Mocks, Spies, Fakes). Su funcionalidad es comparable a la de Mockito en Java, permitiendo reemplazar objetos reales con objetos controlados para aislar la unidad bajo prueba y verificar interacciones.

Los elementos clave son `MagicMock` y la función `patch`.

`MagicMock` (El análogo a los Mocks de Mockito)

`MagicMock` es una clase versátil que puede simular cualquier objeto o método, y registra todas las interacciones.

Conceptos clave:

- **Comportamiento predefinido**: Puedes configurar valores de retorno (`.return_value`), excepciones (`.side_effect`), o incluso implementar lógicas personalizadas.
- **Verificación de interacciones**: Registra llamadas a métodos, argumentos pasados y cuántas veces se llamaron.

`patch` (Inyección de Mocks/Stubs)

`patch` es una función (o decorador) que reemplaza temporalmente un objeto por un mock durante la ejecución de una prueba. Es tu principal mecanismo para la **inversión de control** en pruebas.

Ejemplo de Código con Dependencia Externa:

Consideremos una clase `GestorDeUsuarios` que interactúa con un `ServicioDeBaseDeDatos` para obtener y guardar usuarios.

```python
# gestor_usuarios.py

class ServicioDeBaseDeDatos:
    def obtener_usuario(self, user_id):
        """Simula una consulta a base de datos real."""
        # En una aplicación real, esto interactuaría con una DB.
        print(f"DEBUG: Obteniendo usuario {user_id} de la DB real...")
        if user_id == 101:
            return {"id": 101, "nombre": "Alice", "email": "alice@example.com"}
        return None

    def guardar_usuario(self, user_data):
        """Simula guardar un usuario en la base de datos."""
        print(f"DEBUG: Guardando usuario {user_data['id']} en la DB real...")
        # Lógica real para guardar
        return True

class GestorDeUsuarios:
    def __init__(self):
        self.db_servicio = ServicioDeBaseDeDatos()

    def obtener_info_usuario(self, user_id):
        """Obtiene información de usuario y la formatea."""
        usuario = self.db_servicio.obtener_usuario(user_id)
        if usuario:
            return f"ID: {usuario['id']}, Nombre: {usuario['nombre']}, Email: {usuario[
        return "Usuario no encontrado."

    def registrar_usuario(self, user_id, nombre, email):
        """Registra un nuevo usuario en la base de datos."""
        nuevo_usuario = {"id": user_id, "nombre": nombre, "email": email}
        if self.db_servicio.guardar_usuario(nuevo_usuario):
            return "Usuario registrado exitosamente."
        return "Fallo al registrar el usuario."
```

Creando Pruebas con `unittest.mock.patch`:

```python
# test_gestor_usuarios.py
import unittest
from unittest.mock import patch, MagicMock
from gestor_usuarios import GestorDeUsuarios

class TestGestorDeUsuarios(unittest.TestCase):

    # Usamos @patch para reemplazar ServicioDeBaseDeDatos con un mock
    # El string 'gestor_usuarios.ServicioDeBaseDeDatos' es la ruta al objeto que se va
    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
    def test_obtener_info_usuario_existente(self, MockServicioDeBaseDeDatos):
        # MockServicioDeBaseDeDatos es la clase Mockeada.
        # Cuando GestorDeUsuarios() crea una instancia de ServicioDeBaseDeDatos(),
        # recibirá una instancia de MagicMock. Necesitamos configurar el comportamiento
        mock_instancia_db = MockServicioDeBaseDeDatos.return_value

        # Configuramos el stub para el método obtener_usuario
        mock_instancia_db.obtener_usuario.return_value = {
            "id": 200, "nombre": "Bob", "email": "bob@example.com"
        }

        gestor = GestorDeUsuarios()
        resultado = gestor.obtener_info_usuario(200)

        # Verificaciones (Assertions)
        self.assertEqual(resultado, "ID: 200, Nombre: Bob, Email: bob@example.com")
        # Verificamos que el método obtener_usuario fue llamado con los argumentos corr
        mock_instancia_db.obtener_usuario.assert_called_once_with(200)
        # Verificamos que no se llamó a guardar_usuario
        mock_instancia_db.guardar_usuario.assert_not_called()

    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
    def test_obtener_info_usuario_no_existente(self, MockServicioDeBaseDeDatos):
        mock_instancia_db = MockServicioDeBaseDeDatos.return_value
        mock_instancia_db.obtener_usuario.return_value = None # Simula que el usuario n

        gestor = GestorDeUsuarios()
        resultado = gestor.obtener_info_usuario(999)

        self.assertEqual(resultado, "Usuario no encontrado.")
        mock_instancia_db.obtener_usuario.assert_called_once_with(999)

    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
```

```
    def test_registrar_usuario_exitoso(self, MockServicioDeBaseDeDatos):
        mock_instancia_db = MockServicioDeBaseDeDatos.return_value
        mock_instancia_db.guardar_usuario.return_value = True # Simula éxito al guardar

        gestor = GestorDeUsuarios()
        resultado = gestor.registrar_usuario(300, "Charlie", "charlie@example.com")

        self.assertEqual(resultado, "Usuario registrado exitosamente.")
        # Verificamos que guardar_usuario fue llamado con los datos correctos
        mock_instancia_db.guardar_usuario.assert_called_once_with(
            {"id": 300, "nombre": "Charlie", "email": "charlie@example.com"}
        )
        mock_instancia_db.obtener_usuario.assert_not_called() # No debería haber llamad

    @patch('gestor_usuarios.ServicioDeBaseDeDatos')
    def test_registrar_usuario_fallido(self, MockServicioDeBaseDeDatos):
        mock_instancia_db = MockServicioDeBaseDeDatos.return_value
        mock_instancia_db.guardar_usuario.return_value = False # Simula fallo al guarda

        gestor = GestorDeUsuarios()
        resultado = gestor.registrar_usuario(400, "Diana", "diana@example.com")

        self.assertEqual(resultado, "Fallo al registrar el usuario.")
        mock_instancia_db.guardar_usuario.assert_called_once_with(
            {"id": 400, "nombre": "Diana", "email": "diana@example.com"}
        )

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

Puntos Clave sobre `patch` y `MagicMock`:

1. **Ruta de `patch`**: El argumento a `patch` es una cadena de texto que indica la ruta *donde el objeto se busca* en el momento de la ejecución. Por ejemplo, si `GestorDeUsuarios` importa `ServicioDeBaseDeDatos` como `from gestor_usuarios import ServicioDeBaseDeDatos`, entonces la ruta para mockear es `gestor_usuarios.ServicioDeBaseDeDatos`.
2. **Mock de la Clase vs. Mock de la Instancia**: Cuando usas `@patch('modulo.Clase')`, `MockServicioDeBaseDeDatos` (el argumento que recibe el método de prueba) es un mock de la *clase* `ServicioDeBaseDeDatos`. Para interactuar con los métodos que serían llamados en una *instancia* de esa clase (ej. `obtener_usuario`), debes acceder a `MockServicioDeBaseDeDatos.return_value`. Esto simula lo que ocurre cuando `GestorDeUsuarios` hace `self.db_servicio = ServicioDeBaseDeDatos()`.

3. **Configuración de Retornos y Efectos Secundarios**:
    - `mock_objeto.metodo.return_value = valor` : Configura el valor que devolverá el método mockeado.
    - `mock_objeto.metodo.side_effect = Excepcion` : Hace que el método mockeado levante una excepción.
    - `mock_objeto.metodo.side_effect = [valor1, valor2, ...]` o `mock_objeto.metodo.side_effect = funcion` : Permite definir una secuencia de retornos o una función personalizada que se ejecutará.
4. **Métodos de Verificación de Mocks (Análogos a `Mockito.verify()`):**
    - `mock_objeto.metodo.assert_called()` : Verifica que el método fue llamado al menos una vez.
    - `mock_objeto.metodo.assert_called_once()` : Verifica que el método fue llamado exactamente una vez.
    - `mock_objeto.metodo.assert_called_with(*args, **kwargs)` : Verifica que el método fue llamado con los argumentos especificados.
    - `mock_objeto.metodo.assert_called_once_with(*args, **kwargs)` : Verifica que fue llamado exactamente una vez con los argumentos especificados.
    - `mock_objeto.metodo.assert_not_called()` : Verifica que el método no fue llamado.

Patrones Adicionales de `patch`

- **`@patch.object()`** : Permite mockear un atributo o método específico de un objeto ya existente, en lugar de una clase completa o un objeto a nivel de módulo.

```
from unittest.mock import patch
# ... dentro de una clase de prueba
def test_con_patch_object(self):
    gestor = GestorDeUsuarios() # Instancia real
    with patch.object(gestor.db_servicio, 'obtener_usuario') as mock_obtener:
        mock_obtener.return_value = {"id": 1, "nombre": "Juan"}
        resultado = gestor.obtener_info_usuario(1)
        self.assertEqual(resultado, "ID: 1, Nombre: Juan, Email: None") # Email es I
        mock_obtener.assert_called_once_with(1)
```

- **`patch` como gestor de contexto (`with patch(...) as mock_obj:`)**: Útil para mockear objetos que no son clases, o para un control más granular del alcance del mock. El mock se revierte automáticamente al salir del bloque `with` .

Consideraciones Finales

- **Alcance de los Mocks**: Los mocks creados con `patch` se activan y desactivan automáticamente dentro del alcance del decorador o del bloque `with`. Esto asegura que las pruebas son aisladas y no interfieren entre sí.
- **Velocidad de Ejecución**: Python `unittest` es generalmente rápido. Los dobles de prueba contribuyen significativamente a mantener esta velocidad al evitar interacciones con sistemas lentos.
- **Mantenimiento**: Escribir pruebas claras y bien estructuradas, junto con un uso juicioso de los dobles de prueba, reduce el acoplamiento y facilita el mantenimiento del código a largo plazo.
- **Pytest**: Aunque `unittest` es el estándar, **Pytest** es un framework de pruebas de terceros muy popular en la comunidad de Python, conocido por su sintaxis más concisa, sus "fixtures" (que son una forma elegante de configurar el entorno de prueba) y su poderosa integración con `unittest.mock`. Si buscas una experiencia aún más simplificada, Pytest es una excelente opción a explorar después de dominar `unittest`.

> **Actividad**
> Realiza las actividades propuestas en el documento de pruebas de integración de mockito, traducidas a python.