Programación con IA Generativa

Pruebas con IA

Índice

1 | Pruebas unitarias

Pruebas unitarias

Las pruebas unitarias están diseñadas para **verificar que un método o función se ejecuta según lo esperado**, de acuerdo con la lógica teórica del desarrollador.

Ejemplo: Pruebas unitarias para los métodos de la clase Cuenta

```python
class Cuenta:
    def __init__(self, saldo_inicial=0):
        if saldo_inicial < 0:
            raise ValueError("El saldo no puede ser negativo")
        self.saldo = saldo_inicial

    def depositar(self, monto):
        if monto <= 0:
            raise ValueError("El monto a depositar no puede ser negativo")
        self.saldo += monto

    def retirar(self, monto):
        if monto <= 0:
            raise ValueError("El monto a retirar no puede ser negativo")
        if monto > self.saldo:
            raise ValueError("El monto a retirar no puede ser mayor al saldo")
        self.saldo -= monto

    def get_saldo(self):
        return self.saldo
```

Consideraciones a tener en cuenta

Podemos solicitarle al agente que escriba pruebas que cubran una gran variedad de escenarios pero debemos especificar bien los requisitos de la prueba para obtener los mejores resultados.

A continuación, se muestra un prompt de ejemplo que puedes usar a fin de generar pruebas unitarias para la clase Cuenta:

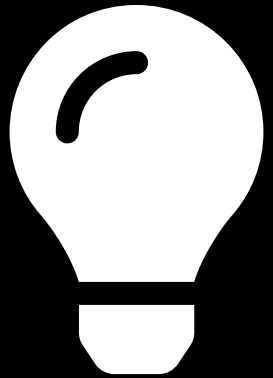
Desarrollar un conjunto completo de pruebas unitarias para la clase Cuenta en Python. Desarrollar varios métodos de prueba que cubran una amplia gama de escenarios e incluyan casos perimetrales, control de excepciones y validación de datos.

Analizando el promt

El prompt hace algunas cosas importantes:

- **Ámbito**: pide de forma explícita un conjunto integral de pruebas unitarias, por lo que el agente no se limita a generar solo pruebas básicas y abarca una variedad de posibles casos de uso.
- **Escenarios**: la solicitud especifica que las pruebas deben incluir:
    - **Casos perimetrales:** comprueba las condiciones de límite (por ejemplo, valores negativos para los depósitos o retiros, cuando se retira una cantidad superior al saldo disponible).
    - **Control de excepciones:** pruebas para errores que deben activarse cuando se producen determinadas acciones no válidas (por ejemplo, retirar una cantidad superior al saldo disponible, depositar cantidades negativas).
    - **Validación de datos:** garantiza que los métodos validen y controlen las entradas correctamente.

Podemos utilizar el chat utilizando **/tests** + nombre de la clase o método.



Ejemplo: Pruebas unitarias para los métodos de la clase Cuenta

```python
import unittest
from prueba.service import Cuenta  # Assuming Cuenta is in prueba.service

class TestCuenta(unittest.TestCase):

    def test_inicializacion_saldo_positivo(self):
        cuenta = Cuenta(100)
        self.assertEqual(cuenta.get_saldo(), 100)

    def test_inicializacion_saldo_cero(self):
        cuenta = Cuenta()
        self.assertEqual(cuenta.get_saldo(), 0)

    def test_inicializacion_saldo_negativo(self):
        with self.assertRaisesRegex(ValueError, "El saldo no puede ser negativo"):
            Cuenta(-50)
```

Ejemplo: Pruebas unitarias para los métodos de la clase Cuenta

```python
def test_deposito_monto_positivo(self):
    cuenta = Cuenta(100)
    cuenta.depositar(50)
    self.assertEqual(cuenta.get_saldo(), 150)

def test_deposito_monto_cero(self):
    cuenta = Cuenta(100)
    with self.assertRaisesRegex(ValueError, "El monto a depositar no puede ser negativo"):
        cuenta.depositar(0)
    self.assertEqual(cuenta.get_saldo(), 100) # Saldo no debe cambiar

def test_deposito_monto_negativo(self):
    cuenta = Cuenta(100)
    with self.assertRaisesRegex(ValueError, "El monto a depositar no puede ser negativo"):
        cuenta.depositar(-50)
    self.assertEqual(cuenta.get_saldo(), 100) # Saldo no debe cambiar
```

Ejemplo: Pruebas unitarias para los métodos de la clase Cuenta

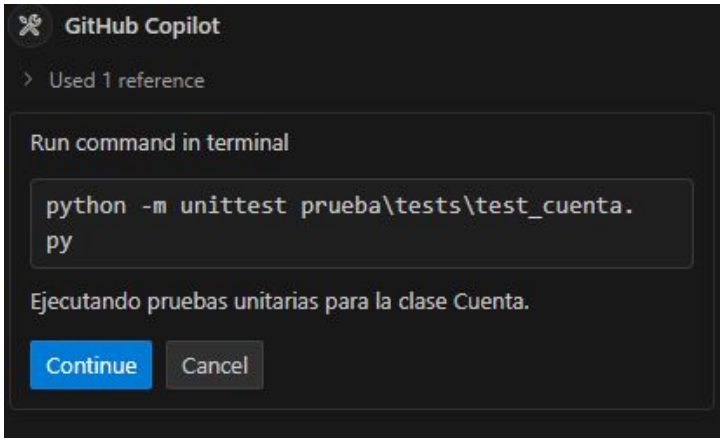
```python
    def test_retiro_monto_cero(self):
        cuenta = Cuenta(100)
        with self.assertRaisesRegex(ValueError, "El monto a retirar no puede ser negativo"):
            cuenta.retirar(0)
        self.assertEqual(cuenta.get_saldo(), 100) # Saldo no debe cambiar

    def test_retiro_monto_negativo(self):
        cuenta = Cuenta(100)
        with self.assertRaisesRegex(ValueError, "El monto a retirar no puede ser negativo"):
            cuenta.retirar(-20)
        self.assertEqual(cuenta.get_saldo(), 100) # Saldo no debe cambiar

    def test_retiro_monto_mayor_al_saldo(self):
        cuenta = Cuenta(50)
        with self.assertRaisesRegex(ValueError, "El monto a retirar no puede ser mayor al saldo"):
            cuenta.retirar(100)
        self.assertEqual(cuenta.get_saldo(), 50) # Saldo no debe cambiar
```

Ejemplo: Pruebas unitarias para los métodos de la clase Cuenta

```
    def test_get_saldo_despues_de_inicializacion(self):
        cuenta = Cuenta(200)
        self.assertEqual(cuenta.get_saldo(), 200)

    def test_get_saldo_despues_de_deposito(self):
        cuenta = Cuenta()
        cuenta.depositar(75)
        self.assertEqual(cuenta.get_saldo(), 75)

    def test_get_saldo_despues_de_retiro(self):
        cuenta = Cuenta(100)
        cuenta.retirar(25)
        self.assertEqual(cuenta.get_saldo(), 75)


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

Ejecutar las pruebas

Una vez desarrolladas las pruebas podemos también solicitarle al agente con un prompt que nos ejecute las mismas:



2 | Pruebas de integración

Pruebas de Integración

Pruebas integrales o pruebas de integración son aquellas que se realizan en el ámbito del desarrollo de software una vez que se han aprobado las pruebas unitarias y **prueban que todos los elementos unitarios que componen el software, funcionan juntos correctamente probándolos en grupo.**

Ejemplo: Pruebas de integración entre una clase Cuenta y una clase Notificación

```python
class Cuenta:
    def __init__(self, saldo_inicial=0, notificador=None):
        if saldo_inicial < 0:
            raise ValueError("El saldo no puede ser negativo")
        self.saldo = saldo_inicial
        self.notificador = notificador

    def depositar(self, monto):
        if monto <= 0:
            raise ValueError("El monto a depositar no puede ser negativo")
        self.saldo += monto
        if self.notificador:
            self.notificador.notificar(f"Se depositó {monto}, y el nuevo saldo es: {self.saldo}")

    def retirar(self, monto):
        if monto <= 0:
            raise ValueError("El monto a retirar no puede ser negativo")
        if monto > self.saldo:
            raise ValueError("El monto a retirar no puede ser mayor al saldo")
        self.saldo -= monto
        if self.notificador:
            self.notificador.notificar(f"Se retiró {monto}, y el nuevo saldo es: {self.saldo}")

    def get_saldo(self):
        return self.saldo
```

Consideraciones a tener en cuenta

A continuación, se muestra un prompt de ejemplo que puedes usar a fin de generar pruebas integrales para la clase Cuenta y Notificador:

Desarrollar pruebas de integración para la función depositar en la clase Cuenta. Usar mocks para simular el Notificador y comprobar que se invoca correctamente después de un depósito.

Analizando el promt

El prompt hace algunas cosas importantes:

- **Ámbito:** especifica pruebas de integración, para lo que se centra en la interacción entre el método deposito y notificar, en lugar de solo pruebas unitarias.
- **Mocks:** solicita de forma explícita el uso de Moks (simular) la clase Notificador con la garantía de que se prueba la interacción con los sistemas externos sin depender de su implementación real.
- **Verificación**: la solicitud enfatiza que hay que comprobar que se llama correctamente al Notificador después de un depósito, lo que garantiza que la integración entre los componentes funciona según lo previsto.
- **Especificidad**: la solicitud indica claramente el método (depositar) y la clase (Cuenta) que se van a probar.

Mocks

En la programación orientada a objetos se llaman objetos simulados a los objetos que imitan el comportamiento de objetos reales de una forma controlada.

El método **assert_called_once_with** es un método que pertenece a la clase Mock del módulo unittest.mock en Python. Se utiliza en las pruebas unitarias para afirmar tres cosas sobre un método simulado::

- Que el método fue llamado.
- Que fue llamado exactamente una vez.
- Que fue llamado con los argumentos específicos que se indican en la aserción.

Ejemplo: Pruebas de integración entre una clase Cuenta y una clase Notificación

```python
import unittest
from unittest.mock import Mock
from bank_account import BankAccount

class TestCuentaIntegration(unittest.TestCase):
    def setUp(self):
        self.notificador = Mock()

    def test_depositar_con_notificador(self):
        cuenta= Cuenta(saldo_inicial=100, notificador=self.notificador)
        cuenta.depositar(50)
        self.assertEqual(cuenta.get_saldo(), 150)
        self.notificador.notificar.assert_called_once_with("Se depositó 50, y el nuevo saldo es: 150")

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

Actividad en grupo

Armar un ejemplo en el que el asistente de IA nos construya las pruebas unitarias.

20:40 a 21:00 Actividad en equipo

21:00 sala principal

? Dudas