Pruebas y manejo de errores en Python

Writing unit tests in python

Para escribir pruebas unitarias en Python, puedes utilizar el módulo unittest. Las pruebas unitarias te permiten verificar el comportamiento de funciones y métodos individuales.

```
import unittest

# Función a probar
def add(a, b):
    return a + b

# Clase de prueba
class TestMathFunctions(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(1, 2), 3)
        self.assertEqual(add(-1, 1), 0)

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

Edge cases

Los casos límite (edge cases) son pruebas que evalúan el comportamiento de una función o método en los valores extremos o inusuales de su rango de entrada.

```
import unittest

# Función a probar
def divide(a, b):
```

```
    if b == 0:
        raise ValueError("No se puede dividir por cero.")
    return a / b

# Clase de prueba
class TestEdgeCases(unittest.TestCase):
    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(1, 0)

    def test_large_numbers(self):
        self.assertEqual(divide(10**10, 10**5), 10**5)

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

Additional test cases

Los casos de prueba adicionales son pruebas que cubren situaciones específicas o adicionales que no se cubren en los casos de prueba básicos.

```
import unittest

# Función a probar
def multiply(a, b):
    return a * b

# Clase de prueba
class TestAdditionalCases(unittest.TestCase):
    def test_multiply_negative(self):
        self.assertEqual(multiply(-1, 1), -1)
        self.assertEqual(multiply(-2, -3), 6)

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

Black box vs. white box

Las pruebas de caja negra y caja blanca son dos enfoques de prueba diferentes. La caja negra se enfoca en los resultados sin conocer la implementación interna, mientras que la caja blanca considera la lógica interna del código.

```
# Caja negra: Prueba de la funcionalidad sin conocer la implementación
def is_even(num):
    return num % 2 == 0

class TestBlackBox(unittest.TestCase):
    def test_is_even(self):
        self.assertTrue(is_even(2))
        self.assertFalse(is_even(3))

# Caja blanca: Prueba con conocimiento de la lógica interna
class TestWhiteBox(unittest.TestCase):
    def test_even_numbers(self):
        for i in range(0, 100, 2):
            self.assertTrue(is_even(i))

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

Other test types

Existen varios tipos de pruebas adicionales como pruebas de integración, pruebas de sistema y pruebas de aceptación que verifican diferentes aspectos del software.

```
# Ejemplo de prueba de integración
def concatenate_strings(a, b):
    return a + b

class TestIntegration(unittest.TestCase):
    def test_concatenate_strings(self):
```

```
        self.assertEqual(concatenate_strings("Hello", "
World"), "Hello World")

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

Test-driven development

El desarrollo guiado por pruebas (TDD) es una metodología en la que primero escribes pruebas antes de implementar la funcionalidad necesaria para pasarlas.

```
import unittest

# Función a desarrollar
def subtract(a, b):
    return a - b

# Clase de prueba
class TestTDD(unittest.TestCase):
    def test_subtract(self):
        self.assertEqual(subtract(5, 3), 2)
        self.assertEqual(subtract(10, 5), 5)

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

The try-except construct

El constructo try-except en Python se usa para manejar excepciones que pueden ocurrir durante la ejecución del código.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("No se puede dividir por cero.")
```

Raising errors

Para lanzar excepciones en Python, se utiliza la palabra clave raise. Esto es útil para indicar errores específicos en el flujo del programa.

```
def divide(a, b):
    if b == 0:
        raise ValueError("No se puede dividir por cero.")
    return a / b

try:
    divide(1, 0)
except ValueError as e:
    print(f"Error: {e}")
```

Testing for expected errors

Para probar si una función maneja errores como se espera, puedes usar assertRaises en tus pruebas unitarias.

```
import unittest

def divide(a, b):
    if b == 0:
        raise ValueError("No se puede dividir por cero.")
    return a / b

class TestErrorHandling(unittest.TestCase):
    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(1, 0)

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