1. PRUEBAS UNITARIAS

Una prueba unitaria es una técnica de pruebas de software que se utiliza para verificar el funcionamiento individual y aislado de una unidad de código, generalmente una función, método o clase. El objetivo principal de las pruebas unitarias es asegurarse de que cada unidad de código funcione correctamente de manera independiente antes de combinarla con otras partes del programa.

Algunas ventajas de realizar pruebas unitarias son:

- Detección temprana de errores: Las pruebas unitarias pueden ayudar a identificar errores en el código antes de que se integren con otras partes del programa. Esto permite corregir los errores de manera temprana y reduce el tiempo y el esfuerzo necesario para depurar el código más adelante.
- Mejora la calidad del código: Al escribir pruebas unitarias, se requiere pensar en diferentes casos y situaciones posibles, lo que puede llevar a un diseño de código más sólido y modular. Además, las pruebas unitarias actúan como documentación automatizada, lo que facilita la comprensión y el mantenimiento del código a largo plazo.
- Facilita la refactorización: Las pruebas unitarias proporcionan confianza al realizar cambios en el código existente. Si las pruebas unitarias pasan después de la refactorización, tienes una mayor seguridad de que no has introducido nuevos errores en el proceso.

Existen otros tipos de pruebas, como las pruebas de integración, pruebas de aceptación y pruebas de rendimiento, entre otras. Cada tipo de prueba tiene un propósito específico en el ciclo de desarrollo de software y se enfoca en diferentes aspectos del sistema.

2. TDD

Las pruebas son tan importantes que incluso existe una metodología denominada TDD, significa "Test-Driven Development" (Desarrollo Dirigido por Pruebas, en español). Es una práctica de desarrollo de software en la que las pruebas unitarias se escriben antes de escribir el código de producción.

El enfoque principal del TDD es seguir un ciclo de desarrollo iterativo y repetitivo, que generalmente se compone de los siguientes pasos:

- Escribir una prueba: Primero, se escribe una prueba unitaria que defina el comportamiento esperado de una funcionalidad específica. En esta etapa, la prueba generalmente fallará porque el código de producción aún no se ha implementado.
- Ejecutar la prueba: A continuación, se ejecuta la prueba recién escrita. Como se esperaba, la prueba fallará porque el código de producción necesario no existe o no está implementado correctamente.
- Escribir el código de producción: Ahora, se implementa el código de producción necesario para que la prueba pase correctamente. El objetivo es escribir la cantidad mínima de código necesaria para que la prueba pase.
- Ejecutar todas las pruebas: Después de escribir el código de producción, se ejecutan todas las pruebas, incluida la nueva prueba que se acaba de agregar. Todas las pruebas deben pasar correctamente en esta etapa.
- Refactorizar el código: Si todas las pruebas pasan, se puede proceder a mejorar la calidad del código sin cambiar su comportamiento externo. Esta etapa implica revisar y optimizar el código, asegurándose de que siga siendo legible, mantenible y eficiente.
- Repetir el ciclo: El ciclo se repite escribiendo una nueva prueba para la siguiente funcionalidad o escenario y luego repitiendo los pasos anteriores.

La principal idea detrás del TDD es que las pruebas actúan como especificaciones para el código. Al escribir las pruebas primero, se establece un objetivo claro y se asegura de que el código esté diseñado para cumplir con ese objetivo. Además, el TDD fomenta el diseño modular, la cobertura de pruebas y la refactorización continua del código.

El TDD puede ayudar a mejorar la calidad del código, acelerar el desarrollo al reducir la cantidad de errores y proporcionar una mayor confianza en el funcionamiento del software.

3. PyUNIT

PyUnit, también conocido como unittest, es un módulo de Python que proporciona un marco de pruebas unitarias. Forma parte de la biblioteca estándar de Python y se basa en el marco de pruebas unitarias de JUnit, que es utilizado en el lenguaje de programación Java.

El módulo unittest (pyunit) ofrece una serie de clases y métodos que facilitan la escritura y ejecución de pruebas unitarias en Python. Algunas características importantes de unittest incluyen:

- Clases de prueba: unittest proporciona la clase TestCase, que se utiliza como base para definir las pruebas unitarias. Puedes crear subclases de TestCase y agregar métodos de prueba a esas subclases.

- Métodos de prueba: Los métodos de prueba son funciones que verifican el comportamiento de una unidad de código específica. Estos métodos deben comenzar con el prefijo test_ para que unittest los identifique como pruebas y los ejecute automáticamente.

- Aserciones: unittest ofrece una variedad de métodos de aserción (como assertEqual, assertTrue, assertFalse, entre otros) que permiten verificar condiciones y comparar valores esperados con los valores obtenidos durante la ejecución de una prueba.

- Configuración y limpieza: unittest proporciona métodos especiales para realizar configuración y limpieza antes y después de la ejecución de cada prueba. Estos métodos, como setUp() y tearDown(), se utilizan para preparar el entorno de prueba antes de cada prueba y realizar limpieza posterior.

- Descubrimiento y ejecución de pruebas: unittest incluye herramientas para descubrir automáticamente las pruebas en un conjunto de archivos y directorios, y para ejecutarlas en un orden determinado. Puedes ejecutar las pruebas a través de la línea de comandos o mediante herramientas de ejecución específicas.

Ejemplo

```
import unittest

class MyTestCase(unittest.TestCase):
    def test_sum(self):
        result = 2 + 2
        self.assertEqual(result, 4)
```

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

En este ejemplo, creamos una subclase de unittest.TestCase llamada MyTestCase y definimos un método de prueba llamado test_sum. Dentro de este método, realizamos una suma y luego usamos self.assertEqual para verificar si el resultado es igual a 4.

Luego, ejecutamos las pruebas utilizando unittest.main(), que descubre automáticamente todas las pruebas en el archivo y las ejecuta.

4. Metodos assert

La biblioteca unittest proporciona una variedad de métodos assert que puedes utilizar para verificar diferentes condiciones en tus pruebas unitarias. Algunos de los métodos assert más comunes incluyen:

- assertEqual(a, b): Verifica que a sea igual a b.
- assertNotEqual(a, b): Verifica que a no sea igual a b.
- assertTrue(x): Verifica que x sea verdadero.
- assertFalse(x): Verifica que x sea falso.
- assertIs(a, b): Verifica que a sea el mismo objeto que b.
- assertIsNot(a, b): Verifica que a no sea el mismo objeto que b.
- assertIsNone(x): Verifica que x sea None.
- assertIsNotNone(x): Verifica que x no sea None.
- assertIn(a, b): Verifica que a esté presente en b.
- assertNotIn(a, b): Verifica que a no esté presente en b.
- assertIsInstance(a, b): Verifica que a sea una instancia de la clase b.
- assertNotIsInstance(a, b): Verifica que a no sea una instancia de la clase b.
- assertRaises(exception, callable, *args, **kwargs): Verifica que llamar a callable con los argumentos dados genere una excepción del tipo exception.
- assertRaisesRegex(exception, regex, callable, *args, **kwargs): Verifica que llamar a callable genere una excepción del tipo exception y que el mensaje de error coincida con el patrón de expresión regular regex.
- assertAlmostEqual(a, b): Verifica que a y b sean casi iguales (útil para comparar números de punto flotante).

- assertNotAlmostEqual(a, b): Verifica que a y b no sean casi iguales.

Para utilizar los métodos de aserción, simplemente llámalos dentro de los métodos de prueba de tu clase TestCase. Por ejemplo:

```
import unittest


class MyTestCase(unittest.TestCase):
    def test_sum(self):
        result = 2 + 2
        self.assertEqual(result, 4)


    def test_boolean(self):
        value = True
        self.assertTrue(value)


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

En este ejemplo, usamos self.assertEqual en el método test_sum para verificar si la suma de 2 y 2 es igual a 4. También utilizamos self.assertTrue en el método test_boolean para verificar si la variable value es verdadera.

Si alguna de las aserciones falla, unittest registrará el resultado como una falla en la prueba y proporcionará información detallada sobre el error.

Recuerda que puedes combinar múltiples aserciones en una sola prueba para verificar diferentes condiciones y escenarios.

Ejemplo:

Supongamos que tienes una clase llamada Calculator que contiene un método is_positive() que devuelve True si un número es positivo y False en caso contrario. Queremos asegurarnos de que el método is_positive() funcione correctamente.

```
import unittest


class Calculator:
```

```
    def is_positive(self, number):
        return number > 0


class CalculatorTestCase(unittest.TestCase):
    def test_is_positive(self):
        calculator = Calculator()
        self.assertTrue(calculator.is_positive(5))
        self.assertTrue(calculator.is_positive(10))
        self.assertFalse(calculator.is_positive(-5))
        self.assertFalse(calculator.is_positive(0))


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

En este ejemplo, creamos una clase Calculator que tiene un método is_positive() que verifica si un número es positivo. En nuestra clase CalculatorTestCase, definimos el método test_is_positive() donde realizamos varias aserciones utilizando assertTrue y assertFalse.

- self.assertTrue(calculator.is_positive(5)) verifica que el resultado de calculator.is_positive(5) sea verdadero, es decir, que el número 5 sea considerado positivo.
- self.assertTrue(calculator.is_positive(10)) verifica que el resultado de calculator.is_positive(10) sea verdadero, es decir, que el número 10 sea considerado positivo.
- self.assertFalse(calculator.is_positive(-5)) verifica que el resultado de calculator.is_positive(-5) sea falso, es decir, que el número -5 no sea considerado positivo.
- self.assertFalse(calculator.is_positive(0)) verifica que el resultado de calculator.is_positive(0) sea falso, es decir, que el número 0 no sea considerado positivo.

Si todas las aserciones se cumplen, las pruebas pasarán exitosamente. En caso de que alguna aserción falle, unittest mostrará información detallada sobre la falla y la línea de código asociada.

Recuerda que assertTrue se utiliza para verificar que una condición sea verdadera, mientras que assertFalse se utiliza para verificar que una condición sea falsa.

5. setUp

El método setUp() es un método especial que se utiliza en unittest para realizar configuraciones comunes antes de ejecutar cada prueba en una clase TestCase. Este método se ejecuta automáticamente antes de cada método de prueba y se utiliza para preparar el entorno necesario para las pruebas.

La función setUp() se define en una subclase de unittest.TestCase y se puede utilizar para realizar tareas como:

- Configurar objetos o recursos necesarios para las pruebas.
- Inicializar variables.
- Realizar acciones previas a la ejecución de cada prueba.

```
import unittest


class MyTestCase(unittest.TestCase):
    def setUp(self):
        # Configuración común para todas las pruebas
        self.my_object = MyObject()

    def test_something(self):
        # Acceso a self.my_object en la prueba
        result = self.my_object.do_something()
        self.assertEqual(result, expected_result)

    def test_another_thing(self):
        # Acceso a self.my_object en otra prueba
        result = self.my_object.do_another_thing()
        self.assertEqual(result, expected_result)
```

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

En este ejemplo, la clase MyTestCase hereda de unittest.TestCase y define el método setUp(). Dentro de setUp(), se realiza la configuración necesaria, que en este caso es crear una instancia de MyObject y asignarla a self.my_object.

Luego, en los métodos de prueba, como test_something() y test_another_thing(), puedes acceder a self.my_object y utilizarlo en las pruebas.

Cada vez que se ejecuta un método de prueba, setUp() se ejecutará primero para asegurarse de que el entorno esté configurado correctamente antes de ejecutar la prueba.

Es importante destacar que setUp() se ejecuta antes de cada método de prueba, por lo que cualquier cambio realizado en la configuración dentro de un método de prueba no afectará a los demás métodos de prueba.

El uso de setUp() ayuda a evitar la duplicación de código y garantiza una configuración consistente para todas las pruebas en una clase TestCase.

6. tearDown

El método tearDown() es otro método especial que se utiliza en unittest para realizar tareas de limpieza después de ejecutar cada prueba en una clase TestCase. Este método se ejecuta automáticamente después de cada método de prueba y se utiliza para limpiar cualquier configuración o recursos que se hayan utilizado durante las pruebas.

La función tearDown() se define en una subclase de unittest.TestCase y se puede utilizar para realizar tareas como:

- Liberar recursos utilizados durante las pruebas.
- Restaurar configuraciones a su estado original.
- Realizar acciones posteriores a la ejecución de cada prueba.

```
import unittest

class MyTestCase(unittest.TestCase):
    def setUp(self):
        # Configuración común para todas las pruebas
        self.my_object = MyObject()
```

```
    def tearDown(self):
        # Tareas de limpieza después de cada prueba
        self.my_object.cleanup()

    def test_something(self):
        # Acceso a self.my_object en la prueba
        result = self.my_object.do_something()
        self.assertEqual(result, expected_result)

    def test_another_thing(self):
        # Acceso a self.my_object en otra prueba
        result = self.my_object.do_another_thing()
        self.assertEqual(result, expected_result)

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

En este ejemplo, la clase MyTestCase define tanto el método setUp() como el método tearDown(). Dentro de tearDown(), se realizan las tareas de limpieza necesarias, que en este caso implican llamar al método cleanup() en self.my_object para liberar cualquier recurso o restaurar cualquier configuración necesaria.

Cada vez que se ejecuta un método de prueba, tearDown() se ejecutará automáticamente después de ese método de prueba, incluso si el método de prueba generó una excepción o un error. Esto garantiza que las tareas de limpieza se realicen independientemente del resultado de la prueba.

Es importante destacar que tearDown() se ejecuta después de cada método de prueba, por lo que cualquier cambio realizado en la limpieza dentro de un método de prueba no afectará a los demás métodos de prueba.

El uso de tearDown() es opcional y solo se utiliza cuando necesitas realizar tareas de limpieza específicas después de cada prueba. Si no necesitas

realizar ninguna tarea de limpieza, no es necesario definir el método tearDown() en tu clase TestCase.

7. Fixture

Lo que fuimos construyendo hasta el momento, en unittest se denomina fixture. Podemos decir entonces que es un conjunto de datos, configuraciones o recursos necesarios para llevar a cabo una o más pruebas. Estos fixtures se utilizan para preparar el entorno de prueba antes de ejecutar las pruebas y también para realizar tareas de limpieza después de que las pruebas se hayan completado.

Los fixtures en unittest se implementan como vimos anteriormente, utilizando los métodos setUp() y tearDown() en una clase TestCase. El método setUp() se utiliza para configurar el entorno de prueba antes de que se ejecute cada prueba, mientras que el método tearDown() se utiliza para limpiar y restablecer el entorno después de cada prueba.

Ejemplo de un fixture completo

```
import unittest


class MyTestCase(unittest.TestCase):
    def setUp(self):
        # Configurar el entorno de prueba
        # Crear objetos, abrir conexiones, inicializar variables, etc.

    def tearDown(self):
        # Limpiar y restablecer el entorno de prueba
        # Cerrar conexiones, liberar recursos, revertir cambios, etc.

    def test_something(self):
        # Prueba que utiliza el entorno de prueba configurado en setUp()
        # Realizar aserciones y verificar resultados

    def test_another_thing(self):
```

```
    # Otra prueba que también utiliza el entorno de prueba configurado en setUp()

    # Realizar aserciones y verificar resultados


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

En este ejemplo, el método setUp() se utiliza para configurar el entorno de prueba común para todas las pruebas. Esto puede incluir la creación de objetos, la inicialización de variables, la apertura de conexiones a bases de datos, entre otros.

El método tearDown() se utiliza para realizar tareas de limpieza y restablecer el entorno de prueba después de cada prueba. Esto puede incluir cerrar conexiones a bases de datos, liberar recursos, revertir cambios realizados durante la prueba, entre otros.

Cada prueba en la clase MyTestCase utilizará el entorno de prueba configurado en setUp() y se ejecutará de forma aislada, sin afectar a las demás pruebas. Después de que se haya ejecutado cada prueba, se llamará automáticamente al método tearDown() para limpiar y restablecer el entorno antes de la siguiente prueba.

El uso de fixtures en unittest ayuda a garantizar que las pruebas se ejecuten en un entorno controlado y consistente, permitiendo la reutilización de configuraciones y la limpieza adecuada después de las pruebas.

8. Ejecución de pruebas

Puedes ejecutar todas tus pruebas en un archivo o módulo utilizando la función unittest.main() como hicimos hasta el momento, ese tipo de ejecución lo denominamos básicas, pero hay otros modos de ejecutar la prueba.

Ejecución selectiva: Si deseas ejecutar solo una prueba específica o un conjunto de pruebas, puedes utilizar la línea de comandos junto con el módulo unittest. Aquí tienes algunos ejemplos:

```
python -m unittest nombre_del_modulo
```

Ejecutar todas las pruebas en una clase TestCase específica:

```
python -m unittest nombre_del_modulo.NombreDeLaClase
```

Ejecutar una prueba específica dentro de una clase TestCase:

python -m unittest nombre_del_modulo.NombreDeLaClase.nombre_del_metodo_de_prueba

Utiliza estos comandos en tu línea de comandos o terminal, reemplazando nombre_del_modulo, NombreDeLaClase y nombre_del_metodo_de_prueba con los nombres reales de tu archivo, clase y método de prueba.

Informes de ejecución: unittest proporciona informes detallados sobre la ejecución de las pruebas. Dependiendo de la configuración, puedes obtener informes en forma de texto en la consola, informes en formato XML, HTML, entre otros.

Para obtener informes en formato XML, puedes usar la opción -xml al ejecutar las pruebas:

python -m unittest nombre_del_modulo -xml

Esto generará un archivo XML con información detallada sobre la ejecución de las pruebas.

9. Colección de casos de prueba: TestSuite

En unittest, una test suite (suite de pruebas) es una colección de casos de prueba que se agrupan lógicamente para ejecutarse juntos. Permite ejecutar conjuntos específicos de pruebas en lugar de ejecutar todas las pruebas en un archivo o módulo.

Una test suite se crea utilizando la clase TestSuite de unittest. Puedes agregar instancias de clases TestCase o incluso otras test suites a la suite de pruebas utilizando los métodos addTest() o addTests(). Luego, puedes ejecutar la test suite para ejecutar todas las pruebas que contiene.

```
import unittest

# Importa las clases de prueba que deseas agregar a la test suite
from test_module1 import TestClass1
from test_module2 import TestClass2


# Crea una instancia de TestSuite
test_suite = unittest.TestSuite()
```

```
# Agrega las clases de prueba a la test suite
test_suite.addTest(unittest.makeSuite(TestClass1))
test_suite.addTest(unittest.makeSuite(TestClass2))

# Ejecuta la test suite
unittest.TextTestRunner().run(test_suite)
```

En este ejemplo, se importan las clases TestClass1 y TestClass2, que contienen las pruebas que deseas agrupar. Luego, se crea una instancia de TestSuite llamada test_suite. Luego, se utilizan los métodos addTest() o addTests() para agregar las clases de prueba a la suite de pruebas.

Finalmente, se utiliza unittest.TextTestRunner().run(test_suite) para ejecutar la test suite y obtener los resultados de las pruebas en la consola.

Utilizar una test suite es útil cuando deseas ejecutar un conjunto específico de pruebas en lugar de todas las pruebas en un archivo o módulo. También puedes anidar test suites dentro de otras test suites para organizar y ejecutar pruebas en grupos lógicos.

Recuerda que la creación y ejecución de una test suite es opcional. Puedes ejecutar pruebas individuales o todas las pruebas sin utilizar una test suite. La test suite proporciona una forma adicional de organizar y ejecutar pruebas según tus necesidades.