import unittest

from testmoto import Moto

class TestMoto(unittest.TestCase):

    def setUp(self):

        self.moto = Moto()

    def test\_acelerar(self):

        self.moto.acelerar()

        self.assertEqual(self.moto.velocidad, 10)

    def test\_frenar(self):

        self.moto.acelerar()

        self.moto.frenar()

        self.assertEqual(self.moto.velocidad, 0)

    def test\_cambiar\_marcha(self):

        self.moto.cambiar\_marcha(2)

        self.assertEqual(self.moto.marcha, 2)

    def test\_marcha\_invalida(self):

        with self.assertRaises(ValueError):

            self.moto.cambiar\_marcha(6)

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

El código proporcionado es un conjunto de pruebas unitarias en Python que utiliza el módulo unittest para verificar el comportamiento de una clase llamada Moto. A continuación, se describe el código utilizando términos de programación:

1. **Importación de Módulos**: Se importa el módulo unittest, que proporciona herramientas para crear y ejecutar pruebas unitarias, y se importa la clase Moto desde el módulo testmoto.
2. **Definición de la Clase de Pruebas**: Se define una clase TestMoto que hereda de unittest.TestCase. Esta clase contendrá varios métodos de prueba para verificar el funcionamiento de la clase Moto.
3. **Método setUp**: Este método se ejecuta antes de cada prueba. Se crea una instancia de Moto y se asigna a self.moto, asegurando que cada prueba comience con un nuevo objeto Moto.
4. **Método de Prueba test\_acelerar**: Se prueba el método acelerar de la clase Moto. Se llama a self.moto.acelerar() y se verifica que la velocidad de la moto sea igual a 10 utilizando self.assertEqual.
5. **Método de Prueba test\_frenar**: Se prueba el método frenar. Primero, se acelera la moto y luego se llama a self.moto.frenar(). Se verifica que la velocidad sea 0 después de frenar.
6. **Método de Prueba test\_cambiar\_marcha**: Se prueba el método cambiar\_marcha de la clase Moto. Se cambia la marcha a 2 y se verifica que la marcha actual de la moto sea igual a 2.
7. **Método de Prueba test\_marcha\_invalida**: Se prueba el manejo de excepciones al intentar cambiar a una marcha inválida (6). Se espera que se lance un ValueError, que se verifica utilizando self.assertRaises.
8. **Ejecución de Pruebas**: Al final del archivo, se incluye un bloque que ejecuta las pruebas si el archivo se ejecuta como un script principal, llamando a unittest.main().

Este conjunto de pruebas asegura que los métodos de la clase Moto funcionen correctamente y manejen adecuadamente los casos de error.

El código proporcionado define una clase llamada Moto, que representa una motocicleta y contiene métodos para manipular su velocidad y marcha. A continuación, se describe el código utilizando términos de programación:

1. **Definición de la Clase Moto**: Se define la clase Moto, que encapsula el comportamiento y las propiedades de una motocicleta.
2. **Método Constructor \_\_init\_\_**: Este método se ejecuta al crear una nueva instancia de Moto. Inicializa dos atributos:
   * self.velocidad: Se establece en 0, representando la velocidad inicial de la moto.
   * self.marcha: Se establece en 1, asumiendo que la marcha inicial es la primera.
3. **Método acelerar**: Este método incrementa la velocidad de la moto en 10 unidades cada vez que se llama. No hay restricciones en la velocidad máxima, lo que significa que puede seguir aumentando indefinidamente.
4. **Método frenar**: Este método disminuye la velocidad de la moto en 10 unidades. Sin embargo, utiliza la función max para asegurarse de que la velocidad no sea negativa. Si la velocidad actual es menor que 10, se establece en 0.
5. **Método cambiar\_marcha**: Este método permite cambiar la marcha de la moto. Acepta un parámetro marcha y verifica si está dentro del rango válido (entre 1 y 5). Si el valor está fuera de este rango, se lanza una excepción ValueError con un mensaje que indica que la marcha debe estar entre 1 y 5. Si el valor es válido, se actualiza el atributo self.marcha con el nuevo valor.

En resumen, la clase Moto proporciona una representación básica de una motocicleta, permitiendo controlar su velocidad y marcha, y asegurando que las operaciones realizadas sean válidas y coherentes.

La parte de código proporcionada muestra una versión refactorizada de la clase Moto. A continuación, se describe el código utilizando términos de programación:

1. **Definición de la Clase Moto**: La clase Moto sigue representando una motocicleta, pero ahora incluye documentación para cada método, lo que mejora la claridad y la mantenibilidad del código.
2. **Método Constructor \_\_init\_\_**: Este método inicializa los atributos velocidad y marcha de la misma manera que en la versión anterior:
   * self.velocidad: Se establece en 0, representando la velocidad inicial de la moto.
   * self.marcha: Se establece en 1, asumiendo que la marcha inicial es la primera.
3. **Método acelerar**: Este método incrementa la velocidad de la moto en 10 unidades. Se ha añadido un comentario de documentación (docstring) que explica brevemente la funcionalidad del método, indicando que aumenta la velocidad de la moto.
4. **Método frenar**: Este método disminuye la velocidad de la moto en 10 unidades, asegurando que la velocidad no sea negativa mediante el uso de la función max. También se ha añadido un docstring que describe su funcionalidad, especificando que no permite que la velocidad sea negativa.
5. **Método cambiar\_marcha**: Este método permite cambiar la marcha de la moto. Se verifica que el valor de marcha esté dentro del rango válido (entre 1 y 5). Si el valor es inválido, se lanza una excepción ValueError. Se ha incluido un docstring que explica que este método cambia la marcha de la moto y que el valor debe estar entre 1 y 5.

En resumen, la refactorización de la clase Moto mejora la documentación del código, lo que facilita su comprensión y uso. La funcionalidad de la clase se mantiene igual, pero ahora es más accesible para otros desarrolladores que puedan trabajar con el código en el futuro

Las tres etapas del código que has proporcionado (la clase de pruebas unitarias, la clase Moto y la refactorización de la clase Moto) son necesarias y tienen un orden lógico para funcionar correctamente. A continuación, se explica la importancia de cada etapa y su relación:

1. **Definición de la Clase Moto**:
   * **Importancia**: Esta etapa es fundamental porque define la lógica y el comportamiento de la motocicleta. Sin esta clase, no habría nada que probar. La clase Moto contiene los métodos que permiten manipular la velocidad y la marcha de la moto.
   * **Funcionalidad**: La clase debe estar bien implementada para que las pruebas unitarias puedan verificar su correcto funcionamiento. Si hay errores en esta clase, las pruebas fallarán.
2. **Refactorización de la Clase Moto**:
   * **Importancia**: La refactorización mejora la calidad del código, haciéndolo más legible y mantenible. Al agregar documentación (docstrings), se facilita la comprensión del propósito de cada método, lo que es crucial para el trabajo en equipo y el mantenimiento a largo plazo.
   * **Funcionalidad**: Aunque la refactorización no cambia la lógica del código, asegura que cualquier desarrollador que lea el código en el futuro pueda entender rápidamente cómo funciona la clase Moto y cómo interactuar con ella.
3. **Definición de la Clase de Pruebas TestMoto**:
   * **Importancia**: Esta etapa es esencial para garantizar que la clase Moto funcione como se espera. Las pruebas unitarias permiten detectar errores y asegurar que los métodos de la clase se comporten correctamente bajo diferentes condiciones.
   * **Funcionalidad**: Las pruebas dependen de la implementación de la clase Moto. Si la clase no está definida o tiene errores, las pruebas no podrán ejecutarse correctamente. Además, si la clase se refactoriza, las pruebas deben seguir siendo válidas para asegurar que la funcionalidad no se haya visto afectada.

**Orden de Funcionamiento**

El orden de estas etapas es crucial:

1. **Primero** se debe definir la clase Moto, ya que es la base sobre la cual se construirán las pruebas.
2. **Luego**, se puede refactorizar la clase Moto para mejorar su calidad y legibilidad, asegurando que la lógica siga siendo correcta.
3. **Finalmente**, se define la clase de pruebas TestMoto, que se basa en la implementación de la clase Moto para verificar su correcto funcionamiento.

Si se intenta ejecutar las pruebas antes de definir la clase Moto, o si la clase tiene errores, las pruebas fallarán. Por lo tanto, seguir este orden es esencial para el desarrollo efectivo y la validación del código.

a abstracción y el encapsulamiento son dos conceptos fundamentales de la programación orientada a objetos que se aplican en el código de la clase Moto. A continuación, se explica cómo se manifiestan estos conceptos en el código proporcionado:

**Abstracción**

**Definición**: La abstracción se refiere a la capacidad de representar conceptos complejos mediante modelos más simples, ocultando los detalles innecesarios y mostrando solo la funcionalidad esencial.

**Aplicación en el Código**:

* La clase Moto abstrae el comportamiento de una motocicleta al proporcionar métodos como acelerar, frenar y cambiar\_marcha.
* Los usuarios de la clase no necesitan conocer los detalles internos de cómo se implementan estos métodos; solo necesitan saber que pueden llamar a estos métodos para interactuar con la moto.
* Por ejemplo, al llamar a self.moto.acelerar(), el usuario no necesita saber cómo se incrementa la velocidad internamente; solo necesita saber que la moto acelerará.

**Encapsulamiento**

**Definición**: El encapsulamiento es el principio de ocultar el estado interno de un objeto y requerir que toda interacción se realice a través de métodos definidos. Esto ayuda a proteger el estado del objeto y a mantener la integridad de los datos.

**Aplicación en el Código**:

* En la clase Moto, los atributos velocidad y marcha son encapsulados dentro de la clase. No se accede directamente a estos atributos desde fuera de la clase; en su lugar, se utilizan métodos para interactuar con ellos.
* Por ejemplo, la velocidad solo puede ser modificada a través de los métodos acelerar y frenar, lo que asegura que la velocidad nunca se establezca en un valor negativo (gracias a la lógica en el método frenar).
* Además, el método cambiar\_marcha incluye validaciones para asegurarse de que la marcha esté dentro de un rango válido, protegiendo así el estado interno de la moto.

**Resumen**

* **Abstracción**: Se observa en la forma en que la clase Moto proporciona métodos que permiten a los usuarios interactuar con la moto sin necesidad de conocer los detalles de su implementación.
* **Encapsulamiento**: Se observa en la forma en que los atributos velocidad y marcha están protegidos dentro de la clase y solo pueden ser modificados a través de métodos específicos, asegurando la integridad del estado del objeto.

Estos principios ayudan a crear un código más limpio, mantenible y menos propenso a errores, facilitando la comprensión y el uso de la clase Moto.