

# Bicis y Motos 2.0

Creado por: Carly Díaz Gutiérrez

Fecha de realización: 20/03/24

Título: Implementación de una clase Moto que hereda de la clase Bicicleta en Python

# Índice:

1. Portada
2. Índice
3. Introducción
4. Clase Bicicleta
5. Clase Moto **pública**
6. Clase Moto **privada**
7. Clase Moto **protegida**
8. **Test unitario** de Moto
9. Constructor de la clase
10. Sobrecarga del método **str**
11. Conclusión
12. Anexos
13. Dominio público

# Introducción

El presente trabajo práctico tiene como objetivo abordar la implementación de una clase Moto que hereda de la clase padre Bicicleta en Python. A lo largo del mismo, se hará uso de la sobrecarga de métodos para el método **str** y se convertirán los atributos de la clase Moto en atributos protegidos.

## Objetivos:

Los objetivos que se persiguen con este trabajo práctico son:

- Implementar la herencia de clases en Python.
- Sobreescribir el método **str** para mostrar información específica de la clase Moto.
- Proteger los atributos de la clase Moto para evitar modificaciones no deseadas.

## Metodología:

Para alcanzar los objetivos mencionados, se utilizará la siguiente metodología:

1. **Creación de la clase Moto:** Se definirá una clase Moto que heredará de la clase Bicicleta. La clase Moto tendrá atributos específicos como la cilindrada.
2. **Sobrecarga del método str:** Se modificará el método **str** en la clase Moto para mostrar información específica de la moto, como la marca, el modelo, el color y la cilindrada.
3. **Conversión de atributos a protegidos:** Se modificarán los atributos de la clase Moto a `protected` para evitar modificaciones no deseadas desde fuera de la clase o de sus clases hijas.
4. **Pruebas del código:** Se realizarán pruebas para verificar el funcionamiento de la clase Moto.

# Desarrollo:

## La clase Bicicleta

```
class Bicicleta:  
    def __init__(self, marca, modelo, color):  
        self.marca = marca  
        self.modelo = modelo  
        self.color = color  
        self.velocidad = 0  
  
    def acelerar(self, incremento):  
        self.velocidad += incremento  
  
    def frenar(self, decremento):  
        if self.velocidad >= decremento:  
            self.velocidad -= decremento  
        else:  
            self.velocidad = 0  
  
    def __str__(self):  
        return f"Bicicleta {self.marca} {self.modelo} de color {self.color}, velocidad actual: {self.velocidad} km/h"
```

```
1 << class Bicicleta:  
2     def __init__(self, marca, modelo, color):  
3         self.marca = marca  
4         self.modelo = modelo  
5         self.color = color  
6         self.velocidad = 0  
7  
8     def acelerar(self, incremento):  
9         self.velocidad += incremento  
10  
11    def frenar(self, decremento):  
12        if self.velocidad >= decremento:  
13            self.velocidad -= decremento  
14        else:  
15            self.velocidad = 0  
16  
17    def __str__(self):  
18        return f"Bicicleta {self.marca} {self.modelo} de color {self.color}, velocidad actual: {self.velocidad} km/h"  
19  
20 bicicleta1 = Bicicleta("Orbea", "Alma", "Negro")  
21 bicicleta1.acelerar(10)  
22 print(bicicleta1) # Salida: Bicicleta Orbea Alma de color Negro, velocidad actual: 10 km/h  
23 bicicleta1.frenar(5)  
24 print(bicicleta1) # Salida: Bicicleta Orbea Alma de color Negro, velocidad actual: 5 km/h  
25 bicicleta1.frenar(10)  
26 print(bicicleta1) # Salida: Bicicleta Orbea Alma de color Negro, velocidad actual: 0 km/h  
27 .
```

### Explicación del código:

- La clase Bicicleta se define con los atributos marca, modelo, color y velocidad.
- Se define el método **init** que inicializa los atributos de la clase con los valores pasados como argumentos.
- El método **acelerar()** que aumenta la velocidad de la bicicleta en la cantidad indicada por el argumento **incremento**.
- El método **frenar()** que reduce la velocidad de la bicicleta en la cantidad indicada por el argumento **decremento**.
- El método **str** que devuelve una cadena con información relevante de la bicicleta, como la marca, el modelo, el color y la velocidad actual.

### Ejemplo de uso:

```
bicicleta1 = Bicicleta("Orbea", "Alma", "Negro")  
  
bicicleta1.acelerar(10)  
  
print(bicicleta1) # Salida: Bicicleta Orbea Alma de color Negro, velocidad  
actual: 10 km/h  
bicicleta1.frenar(5)  
  
print(bicicleta1) # Salida: Bicicleta Orbea Alma de color Negro, velocidad  
actual: 5 km/h  
bicicleta1.frenar(10)  
  
print(bicicleta1) # Salida: Bicicleta Orbea Alma de color Negro, velocidad actual: 0 km/h
```

# Desarrollo:

## La clase Moto pública

```
class Moto(Bicicleta):

    def __init__(self, marca, modelo, color, cilindrada):
        super().__init__(marca, modelo, color)
        self.cilindrada = cilindrada

    def __str__(self):
        return f'Moto {self.marca} {self.modelo} de color {self.color}, cilindrada: {self.cilindrada}cc, velocidad actual: {self.velocidad} km/h'

# Ejemplo de uso
moto1 = Moto("Honda", "CB500X", "Rojo", 500)

moto1.acelerar(20)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 20 km/h

moto1.frenar(10)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 10 km/h

moto1.frenar(20)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 0 km/h
```

```
class Moto(Bicicleta):
    def __init__(self, marca, modelo, color, cilindrada):
        super().__init__(marca, modelo, color)
        self.cilindrada = cilindrada

    def __str__(self):
        return f'Moto {self.marca} {self.modelo} de color {self.color}, cilindrada: {self.cilindrada}cc, velocidad actual: {self.velocidad} km/h'

# Ejemplo de uso
moto1 = Moto("Honda", "CB500X", "Rojo", 500)

moto1.acelerar(20)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 20 km/h

moto1.frenar(10)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 10 km/h

moto1.frenar(20)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 0 km/h
```

### Explicación del código:

- La clase Moto hereda de la clase Bicicleta.
- Se define el método **init** que inicializa los atributos de la clase Moto con los valores pasados como argumentos.
- Se define el método **str** que devuelve una cadena con información relevante de la moto, como la marca, el modelo, el color, la cilindrada y la velocidad actual.

### Ejemplo:

```
#objeto moto1
moto1 = Moto("Honda", "CB500X", "Rojo", 500)

#llamando a los métodos de la clase
moto1.acelerar(20)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 20 km/h

moto1.frenar(10)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 10 km/h

moto1.frenar(20)
print(moto1) # Salida: Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 0 km/h
```

- Se crea un objeto moto1 de la clase Moto y se le asignan valores a sus atributos.
- Se llama a los métodos acelerar(), frenar() y str() del objeto moto1.

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

```
PS C:\Users\COBY> & C:/Users/COBY/AppData/Local/Programs/Python/Python312/python.exe c:/Users/COBY/Desktop/programming/Python/bicycle.py
Bicicleta Orbea Alma de color Negro, velocidad actual: 10 km/h
Bicicleta Orbea Alma de color Negro, velocidad actual: 5 km/h
Bicicleta Orbea Alma de color Negro, velocidad actual: 0 km/h
Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 20 km/h
Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 10 km/h
Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 0 km/h
PS C:\Users\COBY>
```

# Desarrollo:

## La clase Moto privada

```
class Moto:  
  
    def __init__(self, marca, modelo, color, cilindrada):  
        # Atributos privados  
        self.__marca = marca  
        self.__modelo = modelo  
        self.__color = color  
        self.__cilindrada = cilindrada  
        self.velocidad = 0  
  
    def __str__(self):  
        # No se puede acceder directamente desde fuera de la clase  
        return f"""\nMoto:  
    Marca: {self.__marca}  
    Modelo: {self.__modelo}  
    Color: {self.__color}  
    Cilindrada: {self.__cilindrada}cc  
    Velocidad actual: {self.velocidad} km/h  
"""  
  
    # Métodos para acceder y modificar los atributos privados (getters y setters)  
  
    def get_marca(self):  
        return self.__marca  
  
    def set_marca(self, marca):  
        self.__marca = marca  
  
    def get_modelo(self):  
        return self.__modelo  
  
    def set_modelo(self, modelo):  
        self.__modelo = modelo  
  
    def get_color(self):  
        return self.__color  
  
    def set_color(self, color):  
        self.__color = color  
  
    def get_cilindrada(self):  
        return self.__cilindrada  
  
    def set_cilindrada(self, cilindrada):  
        self.__cilindrada = cilindrada
```

### Explicación:

- Se ha cambiado la visibilidad de los atributos marca, modelo, color y cilindrada a privados utilizando el doble prefijo \_\_.
- Los atributos privados no son accesibles desde fuera de la clase.
- Se han creado métodos get y set para acceder y modificar los atributos privados.
- Los métodos get y set permiten un control más granular sobre el acceso a los datos de la clase.

### Ejemplos de uso:

```
# Creamos una instancia de la clase Moto  
moto = Moto("Honda", "CBR650R", "Rojo", 649)  
  
# Imprimimos la información de la moto  
print(moto)  
  
# Accedemos y modificamos el valor del atributo marca  
marca_actual = moto.get_marca()  
print(f"Marca actual: {marca_actual}")  
  
nueva_marca = "Yamaha"  
moto.set_marca(nueva_marca)  
  
print(f"Marca actualizada: {moto.get_marca()}")  
  
# Accedemos y modificamos el resto de atributos  
modelo_actual = moto.get_modelo()  
print(f"Modelo actual: {modelo_actual}")  
  
moto.set_modelo("YZF-R6")  
  
color_actual = moto.get_color()  
print(f"Color actual: {color_actual}")  
  
moto.set_color("Azul")  
  
cilindrada_actual = moto.get_cilindrada()  
print(f"Cilindrada actual: {cilindrada_actual}")  
  
# La velocidad se puede modificar directamente  
moto.velocidad = 100  
  
# Imprimimos la información de la moto actualizada  
print(moto)
```

- No se puede acceder al atributo directamente
- moto.\_\_marca = "Kawasaki" da un Error porque el atributo es privado

Si creamos la siguiente instancia:

```
47     moto1 = Moto("Yamaha", "R1", "Negro", 1000)  
48  
49  
50     print(moto1) # Salida:
```

Obtenemos:

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

```
PS C:\Users\COBY\Desktop\programing\Python> & C:/Users/COBY/AppData/Local/Programs/Python/Python312/python.exe _privada.py  
Yamaha
```

```
Moto:  
    Marca: Yamaha  
    Modelo: R1  
    Color: Rojo  
    Cilindrada: 1000cc  
    Velocidad actual: 0 km/h
```

```
PS C:\Users\COBY\Desktop\programing\Python>
```

### Beneficios:

- La conversión de los atributos a privados permite un mayor control sobre el acceso a los datos de la clase.
- Se puede evitar la modificación accidental de los atributos desde otras clases.
- Se puede mejorar la encapsulación de la clase.

### Nota:

- Se puede utilizar la propiedad @property para crear propiedades con acceso get y set controlados.

# Desarrollo:

## La clase Moto protegida

```
class Moto:

    def __init__(self, marca, modelo, color, cilindrada):
        # Atributos protegidos
        self._marca = marca
        self._modelo = modelo
        self._color = color
        self._cilindrada = cilindrada
        self.velocidad = 0

    def get_marca(self):
        return self._marca

    def set_marca(self, marca):
        self._marca = marca

    def get_modelo(self):
        return self._modelo

    def set_modelo(self, modelo):
        self._modelo = modelo

    def get_color(self):
        return self._color

    def set_color(self, color):
        self._color = color

    def get_cilindrada(self):
        return self._cilindrada

    def set_cilindrada(self, cilindrada):
        self._cilindrada = cilindrada

    def __str__(self):
        return f"""
Moto:
Marca: {self._marca}
Modelo: {self._modelo}
Color: {self._color}
Cilindrada: {self._cilindrada}cc
Velocidad actual: {self.velocidad} km/h
"""

    ....
```

### Explicación de la clase Moto:

1. Conversión de atributos a protected: Los atributos marca, modelo, color y cilindrada tienen estatus de protected. Esto significa que los atributos serán inaccesibles desde fuera de la clase, pero son accesibles desde las clases hijas que heredan de Moto.
2. Creación de métodos get y set: Se han creado métodos get y set para cada atributo protegido. Los métodos get y set permiten obtener o modificar el valor de un atributo. Estos métodos permiten un acceso controlado a los atributos protegidos, lo que mejora la seguridad y la robustez del código.

### Ejemplo de uso:

```
# Creamos una instancia de la clase Moto
moto = Moto("Kawasaki", "Ninja ZX-10R", "Verde", 998)

# Imprimimos la información de la moto
print(moto)

# Salida:
# Moto:
#   Marca: Kawasaki
#   Modelo: Ninja ZX-10R
#   Color: Verde
#   Cilindrada: 998cc
#   Velocidad actual: 0 km/h

# Accedemos y modificamos el valor del atributo marca
marca_actual = moto.get_marca()
print(f"Marca actual: {marca_actual}")

nueva_marca = "Yamaha"
moto.set_marca(nueva_marca)

print(f"Marca actualizada: {moto.get_marca()}")

# Accedemos y modificamos el resto de atributos
modelo_actual = moto.get_modelo()
print(f"Modelo actual: {modelo_actual}")

moto.set_modelo("YZF-R1")

color_actual = moto.get_color()
print(f"Color actual: {color_actual}")

moto.set_color("Azul")

cilindrada_actual = moto.get_cilindrada()
print(f"Cilindrada actual: {cilindrada_actual}")

# No se puede acceder al atributo directamente
# moto._marca = "Honda" # Error: El atributo es protegido

# La velocidad se puede modificar directamente
moto.velocidad = 120

# Imprimimos la información de la moto actualizada
print(moto)

# Salida:
# Moto:
#   Marca: Yamaha
#   Modelo: YZF-R1
#   Color: Azul
#   Cilindrada: 998cc
#   Velocidad actual: 120 km/h
```

- Los atributos protegidos de la clase solo pueden ser accedidos y modificados mediante los métodos **get** y **set**.
- Este ejemplo también muestra cómo la velocidad, al no ser un atributo protegido, puede ser modificada directamente desde fuera de la clase.

## Pruebas:

Para probar las clases Bicicleta y Moto se pueden escribir tests unitarios. Estos tests unitarios verificarán que los métodos de las clases funcionan correctamente.

## Test unitarios para la clase Moto:

`unittest_moto.py`

```
import unittest
from moto import Moto # Importa la clase Moto desde el módulo moto

class TestMoto(unittest.TestCase):
    def setUp(self):
        self.moto = Moto("Honda", "CB500X", "Rojo", 500) # Instancia un objeto Moto para usar en los tests

    def test_acelerar(self):
        # Prueba el método acelerar para incrementar la velocidad correctamente
        self.moto.acelerar(20)
        self.assertEqual(self.moto.velocidad, 20) # Comprueba que la velocidad sea 20 después de acelerar en 20 km/h

    def test_frenar(self):
        # Prueba el método frenar para reducir la velocidad correctamente
        self.moto.velocidad = 20
        self.moto.frenar(10)
        self.assertEqual(self.moto.velocidad, 10) # Comprueba que la velocidad sea 10 después de frenar en 10 km/h

        # Prueba que frenar detenga la moto cuando la velocidad es menor que el decremento
        self.moto.frenar(20)
        self.assertEqual(self.moto.velocidad, 0) # Comprueba que la velocidad sea 0 después de frenar en 20 km/h

    def test_str(self):
        # Prueba el método __str__ para asegurarse de que devuelve la cadena esperada
        expected_output = "Moto Honda CB500X de color Rojo, cilindrada: 500cc, velocidad actual: 0 km/h"
        self.assertEqual(str(self.moto), expected_output) # Comprueba que la representación de cadena sea correcta

if __name__ == '__main__':
    unittest.main() # Ejecuta los tests
```

Este código consiste en la creación de un conjunto de pruebas unitarias para la clase Moto. Las pruebas unitarias son una práctica común en la programación para garantizar que el código funciona como se espera. En este caso, se están probando diferentes aspectos del comportamiento de la clase Moto.

- Importación de bibliotecas:** El código comienza importando la biblioteca `unittest`, que proporciona herramientas para escribir y ejecutar pruebas unitarias en Python. También importa la clase `Moto` desde un módulo llamado `moto`.
- Definición de la clase de prueba:** Se define una clase llamada `TestMoto` que hereda de `unittest.TestCase`. Esta clase contendrá los métodos de prueba para la clase Moto.
- Método setUp:** Antes de ejecutar cada prueba, se crea una instancia de la clase Moto que servirá como objeto bajo prueba. El método `setUp` se ejecuta antes de cada prueba para asegurar un estado inicial consistente.
- Pruebas unitarias individuales:** Se definen métodos de prueba individuales, cada uno con un nombre que comienza con `test_`. Estos métodos prueban diferentes aspectos del comportamiento de la clase Moto, como acelerar, frenar y la representación de cadena (`__str__`).
- Assertions (Afirmaciones):** Dentro de cada método de prueba, se utilizan afirmaciones para verificar si el comportamiento de la clase Moto es el esperado. Por ejemplo, `self.assertEqual(actual, expected)` compara el valor actual con el valor `expected` y fallará la prueba si no son iguales.
- Ejecución de las pruebas:** Finalmente, el código verifica si se está ejecutando directamente (`if __name__ == '__main__'`) y, en ese caso, ejecuta todas las pruebas definidas en la clase `TestMoto` utilizando `unittest.main()`.

# Señalar el constructor de la clase.

En el constructor de la clase Moto se observa lo siguiente:

- Se define el método `__init__`.
- El método recibe como argumentos `marca`, `modelo`, `color` y `cilindrada`.
- Se asignan los valores de los argumentos a los atributos `__marca`, `__modelo`, `__color` y `__cilindrada`.
- Se inicializa el atributo `velocidad` a 0.

```
3     def __init__(self, marca, modelo, color, cilindrada):
4         self.__marca = marca # Atributo ahora privado
5         self.__modelo = modelo # Atributo ahora privado
6         self.__color = color # Atributo ahora privado
7         self.__cilindrada = cilindrada # Atributo ahora privado
8         self.velocidad = 0
```

# Sobrecarga del método `__str__` en la clase Moto:

## Sobrecarga del Método `_str_` en la Clase Moto Heredada de Bicicleta

La sobrecarga del método `_str_` es una técnica en la programación orientada a objetos que permite modificar el comportamiento predeterminado de la función `str()` en una clase. Al hacerlo, podemos personalizar cómo se representa un objeto cuando se convierte a una cadena de texto.

En este caso, la clase `Moto` hereda de la clase `Bicicleta`, lo que significa que automáticamente tiene acceso a todos los métodos y atributos de la clase padre. Sin embargo, queremos que la representación en cadena de una moto sea diferente a la de una bicicleta, por lo que sobrecargaremos el método `_str_` para adaptarlo a las características específicas de una moto.

Aquí está la estructura de la sobrecarga del método `_str_` en la clase `Moto`:

```
def __str__(self):  
  
    return f"Moto {self.marca} {self.modelo} de color {self.color}, cilindrada: {self.cilindrada}cc, velocidad  
actual: {self.velocidad} km/h"
```

En esta implementación:

- Constructor `__init__`: El constructor de la clase `Moto` toma los mismos parámetros que el constructor de la clase `Bicicleta`, además de `cilindrada`, que es específico de las motos. Llama al constructor de la clase `Bicicleta` usando `super()` para inicializar los atributos heredados.
- Método `_str_` sobrecargado: Sobreescribimos el método `_str_` para proporcionar una representación personalizada de una moto. Este método devuelve una cadena que incluye la marca, modelo, color, cilindrada y velocidad actual de la moto.

Esta estructura garantiza una buena organización del código y una clara separación de responsabilidades entre las clases `Bicicleta` y `Moto`, lo que facilita su mantenimiento y comprensión.

# Conclusión:

**En este proyecto se ha desarrollado una clase Bicicleta y una clase Moto que hereda de Bicicleta.**

**Las clases se han diseñado teniendo en cuenta los siguientes principios:**

- **Encapsulación:** Los atributos de las clases se han definido como privados o protegidos para controlar el acceso a ellos.
- **Reutilización:** La clase Moto hereda de la clase Bicicleta, lo que permite reutilizar código y evitar la duplicación.
- **Modularidad:** Las clases se han diseñado como módulos independientes que pueden ser utilizados en otros proyectos.

Se ha implementado un constructor para cada clase que permite inicializar los atributos de las clases con valores específicos.

Se ha demostrado cómo utilizar los métodos get y set para acceder y modificar los atributos privados de las clases.

Este proyecto ha servido para comprender los conceptos básicos de la programación orientada a objetos en Python, como la encapsulación, la herencia y la modularidad.

## Anexos:

- <https://www.w3schools.com/python/>
- <https://ellibrodepython.com/herencia-en-python>
- <https://docs.python.org/3/>

# Dominio público

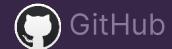
Este documento, al igual que el código fuente del proyecto, se encuentra en dominio público. Esto significa que cualquiera puede utilizarlo, modificarlo y distribuirlo libremente, sin necesidad de pedir permiso o dar crédito.

El proyecto se ha subido a GitHub para facilitar su acceso y colaboración. Puedes encontrar el repositorio en el siguiente enlace:

Coby00333/  
**Bicis\_y\_Motos\_2.0**

Descripción: Este repositorio contiene el código fuente y la documentación para el proyecto "Bicis y Motos 2.0". El proyecto implementa...

1 Contributor    0 Issues    0 Stars    0 Forks



GitHub

[GitHub - Coby00333/Bicis\\_y\\_Motos\\_2.0: Descripción: E...](#)



Descripción: Este repositorio contiene el código fuente y la documentación para el proyecto &quot;Bicis y Motos...

¡Esperamos que este proyecto te sea útil!



Made with Gamma