

Programación Orientada a Objetos (POO)

Polimorfismo de operadores

El polimorfismo de operadores en programación orientada a objetos se refiere a la capacidad de las clases para definir o sobrescribir cómo los operadores estándar (como +, -, *, etc.) se comportan cuando se aplican a instancias de esas clases. En otras palabras, permite que las instancias de una clase respondan a operadores de una manera personalizada y específica para esa clase. Esto se logra mediante la implementación de métodos especiales en una clase. Estos métodos se llaman “métodos mágicos” o “métodos especiales”.

Métodos Mágicos para Operadores Aritméticos

- `__add__(self, other)`: Suma (+)
- `__sub__(self, other)`: Resta (-)
- `__mul__(self, other)`: Multiplicación (*)
- `__truediv__(self, other)`: División (/)
- `__floordiv__(self, other)`: División entera (//)
- `__mod__(self, other)`: Módulo (%)
- `__pow__(self, other)`: Potencia (**)

Métodos Mágicos para Operadores de Comparación

- `__eq__(self, other)`: Igual a (==)
- `__ne__(self, other)`: Diferente de (!=)
- `__lt__(self, other)`: Menor que (<)
- `__le__(self, other)`: Menor o igual que (<=)
- `__gt__(self, other)`: Mayor que (>)
- `__ge__(self, other)`: Mayor o igual que (>=)

Y muchos más...

`**other*` es un parámetro que representa el segundo operando en una operación binaria*

El siguiente código muestra cómo implementar el polimorfismo de operadores en Python mediante la sobrescritura de los métodos especiales `__add__`, `__sub__`, y

`__mul__` en la clase `Vector`. Este enfoque permite que los operadores matemáticos estándar se utilicen con instancias de la clase `Vector`, proporcionando una forma intuitiva de manipular vectores.

Clase `Vector`

- `def __init__(self, x, y)`
 - **Propósito:** Inicializa una instancia de la clase `Vector` con las coordenadas `x` y `y`.
 - **Parámetros:**
 - * `x` (float/int): Coordenada en el eje X.
 - * `y` (float/int): Coordenada en el eje Y.
- `def __add__(self, other)`
 - **Propósito:** Sobrescribe el operador `+` para permitir la suma de dos vectores.
 - **Parámetros:**
 - * `other` (`Vector`): Otro objeto de la clase `Vector` con el que se realiza la suma.
 - **Implementación:**
 - * Retorna un nuevo objeto `Vector` cuya coordenada `x` es la suma de las coordenadas `x` de los dos vectores, y cuya coordenada `y` es la suma de las coordenadas `y` de los dos vectores.
- `def __sub__(self, other)`
 - **Propósito:** Sobrescribe el operador `-` para permitir la resta de dos vectores.
 - **Parámetros:**
 - * `other` (`Vector`): Otro objeto de la clase `Vector` con el que se realiza la resta.
 - **Implementación:**
 - * Retorna un nuevo objeto `Vector` cuya coordenada `x` es la diferencia entre las coordenadas `x` de los dos vectores, y cuya coordenada `y` es la diferencia entre las coordenadas `y` de los dos vectores.
- `def __mul__(self, scalar)`
 - **Propósito:** Sobrescribe el operador `*` para permitir la multiplicación de un vector por un escalar.
 - **Parámetros:**
 - * `scalar` (float/int): Un número que se multiplica con las coordenadas del vector.
 - **Implementación:**
 - * Retorna un nuevo objeto `Vector` cuya coordenada `x` es el producto de la coordenada `x` del vector por el escalar, y cuya coordenada `y` es el producto de la coordenada `y` del vector por el escalar.
- `def __repr__(self)`

- **Propósito:** Sobrescribe el método `__repr__` para proporcionar una representación detallada del vector.
- **Implementación:**
 - * Devuelve una cadena que muestra las coordenadas del vector en el formato `(x, y)`. Este método es útil para la depuración y para mostrar una representación clara del objeto en la consola.

```
# Definir una clase Vector que usa polimorfismo de operadores
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Sobrescribir el operador +
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # Sobrescribir el operador -
    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    # Sobrescribir el operador *
    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    # Sobrescribir el método __repr__ para imprimir el Vector
    def __repr__(self): # __repr__(self): Representación detallada (repr(obj))
        return f"({self.x}, {self.y})"

# Crear instancias de la clase Vector
v1 = Vector(1, 2)
v2 = Vector(3, 4)

# Usar los operadores definidos en la clase Vector
v3 = v1 + v2    # Utiliza __add__
v4 = v1 - v2    # Utiliza __sub__
v5 = v1 * 3     # Utiliza __mul__

# Imprimir los resultados
print(f"v1: {v1}")        # Imprime: Vector(1, 2)
print(f"v2: {v2}")        # Imprime: Vector(1, 2)
print(f"{v1} + {v2} = {v3}")    # Imprime: Vector(1, 2)
```

```
print(f"{v1} - {v2} = {v4}")      # Imprime: Vector(1, 2)
print(f"{v1} * 3 = {v5}")        # Imprime: Vector(1, 2)
```

```
v1: (1, 2)
v2: (3, 4)
(1, 2) + (3, 4) = (4, 6)
(1, 2) - (3, 4) = (-2, -2)
(1, 2) * 3 = (3, 6)
```