Manejo de Clases en Python

En Python, una clase es una plantilla para crear objetos (instancias). Define un conjunto de atributos y métodos que los objetos creados a partir de la clase tendrán.

Terminología clave:

- **Clase**: Es un molde para crear objetos. Define los atributos (variables) y métodos (funciones) que un objeto puede tener.
- **Instancia**: Es un objeto creado a partir de una clase.
- **Herencia**: Es un mecanismo que permite crear una nueva clase a partir de una existente. La nueva clase hereda los atributos y métodos de la clase original, pero puede modificar o agregar nuevos.

Ejemplo: Clases `Punto` y `Vector`

Vamos a utilizar dos clases `Punto` y `Vector` para ilustrar estos conceptos.

Clase `Punto`

La clase `Punto` representa un punto en el espacio 2D con coordenadas `x` y `y`.

```
class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def mover(self, dx, dy):
        """Desplaza el punto en la dirección indicada por dx y dy."""
        self.x += dx
        self.y += dy

    def __str__(self):
        return "Punto({}, {})".format(self.x, self.y)
```

- `__init__`: Es el constructor de la clase. Se llama cuando se crea una nueva instancia de `Punto`. Inicializa las coordenadas `x` y `y`.
- `mover`: Es un método que permite mover el punto.
- `__str__`: Es un método especial que define la representación en cadena del objeto.

Crear una instancia de `Punto`

```
p1 = Punto(2, 3)
print(p1)  # Salida: Punto(2, 3)
p1.mover(1, -1)
print(p1)  # Salida: Punto(3, 2)
```

En este caso, `p1` es una **instancia** de la clase `Punto`. Tiene sus propias coordenadas `x` y `y`.

Clase `Vector`

Ahora, vamos a crear una clase `Vector` que hereda de `Punto`. Un `Vector` es similar a un `Punto`, pero tiene un punto de inicio y un punto de fin.

```
class Vector(Punto):
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        super().__init__(x1, y1)  # Punto inicial
        self.fin = Punto(x2, y2)  # Punto final

    def longitud(self):
        """Calcula la longitud del vector."""
        return ((self.fin.x - self.x) ** 2 + (self.fin.y - self.y) ** 2) ** 0.5

    def __str__(self):
        return "Vector desde " + super().__str__() + " hasta " + str(self.fin)
```

- **Herencia**: `Vector` hereda de `Punto`, lo que significa que hereda todos sus atributos y métodos.
- **`super().__init__`**: Se usa para llamar al constructor de la clase base (`Punto`) y así inicializar el punto de inicio.
- **`longitud`**: Calcula la longitud del vector usando la fórmula de distancia entre dos puntos.

Crear una instancia de `Vector`

```
v1 = Vector(0, 0, 3, 4)
print(v1)  # Salida: Vector desde Punto(0, 0) hasta Punto(3, 4)
print(v1.longitud())  # Salida: 5.0
```

En este ejemplo, `v1` es una **instancia** de la clase `Vector`. Puede acceder a los métodos y atributos de `Punto`, además de sus propios métodos como `longitud`.

A continuación, se muestra la clase `Fraccion` modificada para que almacene las fracciones en una lista de la forma `[numerador, denominador]`:

Clase `Fraccion` con Almacenamiento en Lista

```
import math

class Fraccion:
    def __init__(self, numerador, denominador):
        if denominador == 0:
            raise ValueError("El denominador no puede ser cero")
        self.fraccion = [numerador, denominador]
```

```
        self.simplificar()

    def simplificar(self):
        """Simplifica la fracción utilizando el máximo común divisor."""
        gcd = math.gcd(self.fraccion[0], self.fraccion[1])
        self.fraccion[0] //= gcd
        self.fraccion[1] //= gcd

    def __add__(self, otra):
        """Sobrecarga del operador + para sumar fracciones."""
        nuevo_numerador = self.fraccion[0] * otra.fraccion[1] +
otra.fraccion[0] * self.fraccion[1]
        nuevo_denominador = self.fraccion[1] * otra.fraccion[1]
        return Fraccion(nuevo_numerador, nuevo_denominador)

    def __sub__(self, otra):
        """Sobrecarga del operador - para restar fracciones."""
        nuevo_numerador = self.fraccion[0] * otra.fraccion[1] -
otra.fraccion[0] * self.fraccion[1]
        nuevo_denominador = self.fraccion[1] * otra.fraccion[1]
        return Fraccion(nuevo_numerador, nuevo_denominador)

    def __mul__(self, otra):
        """Sobrecarga del operador * para multiplicar fracciones."""
        nuevo_numerador = self.fraccion[0] * otra.fraccion[0]
        nuevo_denominador = self.fraccion[1] * otra.fraccion[1]
        return Fraccion(nuevo_numerador, nuevo_denominador)

    def __truediv__(self, otra):
        """Sobrecarga del operador / para dividir fracciones."""
        nuevo_numerador = self.fraccion[0] * otra.fraccion[1]
        nuevo_denominador = self.fraccion[1] * otra.fraccion[0]
        return Fraccion(nuevo_numerador, nuevo_denominador)

    def __str__(self):
        return str(self.fraccion[0]) + '/' + str(self.fraccion[1])


def main():
    # Crear dos fracciones
    f1 = Fraccion(1, 2)
    f2 = Fraccion(3, 4)

    # Mostrar las fracciones
    print("Fracción 1: " + str(f1))
    print("Fracción 2: " + str(f2))

    # Realizar y mostrar las operaciones
    suma = f1 + f2
    resta = f1 - f2
    multiplicacion = f1 * f2
    division = f1 / f2
```

```
    print("Suma: " + str(f1) + " + " + str(f2) + " = " + str(suma))
    print("Resta: " + str(f1) + " - " + str(f2) + " = " + str(resta))
    print("Multiplicación: " + str(f1) + " * " + str(f2) + " = " +
str(multiplicacion))
    print("División: " + str(f1) + " / " + str(f2) + " = " + str(division))

if __name__ == "__main__":
    main()
```

Explicación

1. **Atributo `fraccion`**:
   - La fracción se almacena en una lista `[numerador, denominador]` en lugar de utilizar variables separadas.
2. **Operaciones**:
   - Los métodos de sobrecarga (`__add__`, `__sub__`, `__mul__`, `__truediv__`) realizan las operaciones correspondientes utilizando los valores almacenados en la lista `fraccion`.
3. **Representación en cadena (`__str__`)**:
   - Convierte los elementos de la lista en cadenas para mostrar la fracción en el formato `numerador/denominador`.

Ejemplo de Uso

El programa sigue funcionando igual, pero con las fracciones almacenadas en listas. La salida esperada sería:

```
Fracción 1: 1/2
Fracción 2: 3/4
Suma: 1/2 + 3/4 = 5/4
Resta: 1/2 - 3/4 = -1/4
Multiplicación: 1/2 * 3/4 = 3/8
División: 1/2 / 3/4 = 2/3
```

Este diseño sigue siendo eficiente y permite operar con fracciones usando operadores sobrecargados, manteniendo la fracción almacenada en una lista para una mayor consistencia con la solicitud.

Ejemplo de Uso de Clases con Objetos Cotidianos

Vamos a crear dos conjuntos de clases: uno relacionado con personas (donde `Estudiante` y `Profesor` heredan de `Persona`), y otro relacionado con vehículos (donde `Auto` y `Helicoptero` heredan de `Vehiculo`).

Clases `Persona`, `Estudiante`, y `Profesor`

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def describir(self):
        return "Nombre: " + self.nombre + ", Edad: " + str(self.edad)


class Estudiante(Persona):
    def __init__(self, nombre, edad, carrera):
        super().__init__(nombre, edad)
        self.carrera = carrera

    def describir(self):
        return super().describir() + ", Carrera: " + self.carrera


class Profesor(Persona):
    def __init__(self, nombre, edad, asignatura):
        super().__init__(nombre, edad)
        self.asignatura = asignatura

    def describir(self):
        return super().describir() + ", Asignatura: " + self.asignatura
```

Explicación:

- **`Persona`**: Es la clase base que contiene atributos comunes como `nombre` y `edad`, y un método `describir` que devuelve una cadena con estos datos.
- **`Estudiante`**: Hereda de `Persona` y añade un atributo `carrera`. Sobrescribe el método `describir` para incluir la carrera en la descripción.
- **`Profesor`**: También hereda de `Persona` y añade un atributo `asignatura`. Al igual que `Estudiante`, sobrescribe el método `describir` para incluir la asignatura.

Ejemplo de Uso:

```
estudiante = Estudiante("Ana", 20, "Ingeniería Informática")
profesor = Profesor("Dr. Pérez", 45, "Matemáticas")

print(estudiante.describir())
print(profesor.describir())
```

Salida:

```
Nombre: Ana, Edad: 20, Carrera: Ingeniería Informática
Nombre: Dr. Pérez, Edad: 45, Asignatura: Matemáticas
```

Clases `Vehiculo`, `Auto`, y `Helicoptero`

```python
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def describir(self):
        return "Marca: " + self.marca + ", Modelo: " + self.modelo

    def mover(self):
        return "El vehículo se está moviendo"


class Auto(Vehiculo):
    def __init__(self, marca, modelo, puertas):
        super().__init__(marca, modelo)
        self.puertas = puertas

    def describir(self):
        return super().describir() + ", Puertas: " + str(self.puertas)

    def mover(self):
        return "El auto está rodando sobre el asfalto"


class Helicoptero(Vehiculo):
    def __init__(self, marca, modelo, rotores):
        super().__init__(marca, modelo)
        self.rotores = rotores

    def describir(self):
        return super().describir() + ", Rotores: " + str(self.rotores)

    def mover(self):
        return "El helicóptero está volando"
```

Explicación:

- **`Vehiculo`**: Es la clase base con atributos `marca` y `modelo`. Tiene un método `describir` y un método `mover` que indica que el vehículo se está moviendo.
- **`Auto`**: Hereda de `Vehiculo` y añade un atributo `puertas`. Sobrescribe `describir` para incluir el número de puertas y `mover` para indicar que el auto se mueve sobre el asfalto.

- `Helicoptero`: Hereda de `Vehiculo` y añade un atributo `rotores`. Sobrescribe `describir` para incluir el número de rotores y `mover` para indicar que el helicóptero está volando.

Ejemplo de Uso:

```
auto = Auto("Toyota", "Corolla", 4)
helicoptero = Helicoptero("Bell", "206", 2)

print(auto.describir())
print(auto.mover())

print(helicoptero.describir())
print(helicoptero.mover())
```

Salida:

```
Marca: Toyota, Modelo: Corolla, Puertas: 4
El auto está rodando sobre el asfalto
Marca: Bell, Modelo: 206, Rotores: 2
El helicóptero está volando
```

Resumen

En este ejemplo, se demuestra cómo utilizar la herencia en Python para modelar tanto personas como vehículos. Las clases base (`Persona` y `Vehiculo`) definen atributos y comportamientos comunes, mientras que las clases derivadas (`Estudiante`, `Profesor`, `Auto`, `Helicoptero`) extienden y personalizan estos comportamientos según sus características específicas.