1. HERENCIA

La herencia es un concepto fundamental en la programación orientada a objetos que permite la creación de nuevas clases basadas en clases existentes. En términos sencillos, la herencia implica la capacidad de una clase (llamada clase derivada o subclase) de heredar los atributos y métodos de otra clase (llamada clase base o superclase).

Las ventajas de utilizar la herencia en la programación orientada a objetos son:

- Reutilización de código: La herencia permite aprovechar el código existente de la clase base en la clase derivada. Los atributos y métodos definidos en la clase base pueden ser utilizados directamente en la clase derivada sin necesidad de volver a escribirlos.
- Modularidad y extensibilidad: La herencia permite crear una jerarquía de clases, donde cada clase derivada puede agregar o modificar el comportamiento de la clase base. Esto facilita la organización y estructuración del código, así como la incorporación de nuevas funcionalidades en las clases derivadas.
- Polimorfismo: La herencia permite el uso del polimorfismo, lo que significa que los objetos de las clases derivadas pueden ser tratados como objetos de la clase base. Esto brinda flexibilidad en la programación, ya que se puede escribir código genérico que funcione con diferentes tipos de objetos.

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

    def conducir(self):
        print("El vehículo está en movimiento.")

class Coche(Vehiculo):
    def __init__(self, marca, modelo, color):
```

```
        super().__init__(marca, modelo)
        self.color = color

    def conducir(self):
        print(f"El coche {self.marca} {self.modelo} de color {self.color} está en movimiento.")

class Moto(Vehiculo):
    def conducir(self):
        print("La moto está en movimiento.")

# Crear objetos de las clases derivadas
coche = Coche("Ford", "Mustang", "rojo")
moto = Moto("Honda", "CBR")

# Llamar a los métodos de las clases derivadas
coche.conducir()  # Salida: El coche Ford Mustang de color rojo está en movimiento.
moto.conducir()   # Salida: La moto está en movimiento.
```

En este ejemplo, la clase Vehiculo es la clase base que define un vehículo genérico con un método conducir(). La clase Coche y la clase Moto son clases derivadas que heredan de la clase Vehiculo. La clase Coche sobrescribe el método conducir() para imprimir información específica del coche, mientras que la clase Moto utiliza la implementación heredada del método conducir().

La herencia permite que las clases derivadas (Coche y Moto) compartan características y comportamiento comunes con la clase base (Vehiculo) y, al mismo tiempo, extiendan o modifiquen estos atributos y métodos según sea necesario. Esto promueve la reutilización del código y facilita la creación de una jerarquía de clases coherente y organizada.

2. POLIMORFISMO

El polimorfismo es otro concepto base en la programación orientada a objetos que se refiere a la capacidad de un objeto de tomar diferentes

formas o comportarse de diferentes maneras. En términos más generales, el polimorfismo permite que objetos de diferentes clases sean tratados de manera uniforme a través de una interfaz común.

El polimorfismo es conveniente de usar cuando se tiene un conjunto de clases relacionadas que comparten una interfaz común, pero implementan comportamientos específicos de manera diferente. Proporciona flexibilidad y extensibilidad al código, ya que permite que se utilice un único método o función para trabajar con diferentes tipos de objetos, sin necesidad de conocer el tipo específico en tiempo de compilación.

En Python, el polimorfismo se logra mediante la implementación de métodos con el mismo nombre en diferentes clases. Estos métodos deben tener la misma firma (es decir, el mismo nombre y la misma lista de parámetros), pero pueden tener implementaciones diferentes. Al llamar al método a través de una referencia de tipo genérico, el comportamiento adecuado se determina en tiempo de ejecución según el tipo real del objeto.

```
class Animal:
    def hablar(self):
        pass
class Perro(Animal):
    def hablar(self):
        return "Woof!"
class Gato(Animal):
    def hablar(self):
        return "¡Miau!"
def hacer_hablar(animal):
    print(animal.hablar())

# Crear instancias de diferentes clases
perro = Perro()
gato = Gato()

# Llamar al método hacer_hablar con diferentes objetos
hacer_hablar(perro)  # Salida: Woof!
```

```
hacer_hablar(gato)    # Salida: ¡Miau!
```

En este ejemplo, las clases Perro y Gato heredan de la clase base Animal y sobrescriben el método hablar(). La función hacer_hablar() acepta un objeto de tipo Animal y llama al método hablar() del objeto. Aunque el método hablar() se llama de la misma manera en ambas clases, cada clase implementa su propio comportamiento específico. El polimorfismo permite tratar a los objetos Perro y Gato de manera uniforme a través de la interfaz común proporcionada por la clase base Animal.

El polimorfismo es una característica poderosa que promueve la reutilización del código y la flexibilidad en el diseño de sistemas orientados a objetos. Permite escribir un código más genérico y extensible, ya que se puede agregar nuevas clases que cumplan con la interfaz común sin necesidad de modificar el código existente.

3. COMPOSICIÓN

La composición de objetos es un concepto en la programación orientada a objetos que implica combinar objetos más pequeños para construir objetos más complejos. En lugar de heredar comportamiento de otras clases, la composición se basa en la creación de relaciones entre objetos a través de la inclusión de una instancia de una clase dentro de otra.

En Python, la composición se puede lograr mediante la creación de atributos en una clase que sean instancias de otras clases. Los objetos de la clase contenedora pueden acceder y utilizar los atributos y métodos de la clase contenida para lograr cierto comportamiento.

```
class Motor:
    def encender(self):
        print("El motor se ha encendido.")

    def apagar(self):
        print("El motor se ha apagado.")

class Coche:
    def __init__(self):
        self.motor = Motor()  # Instancia de la clase Motor
```

```
    def encender_coche(self):
        self.motor.encender()

    def apagar_coche(self):
        self.motor.apagar()

coche = Coche()
coche.encender_coche()  # Salida: El motor se ha encendido.
coche.apagar_coche()    # Salida: El motor se ha apagado.
```

En este ejemplo, la clase Coche tiene un atributo motor que es una instancia de la clase Motor. La clase Coche utiliza los métodos encender() y apagar() del objeto motor para encender y apagar el coche.

La composición permite construir objetos complejos al combinar objetos más pequeños y reutilizar su funcionalidad. En el ejemplo anterior, el coche se compone de un motor y utiliza sus métodos para lograr ciertos comportamientos. La relación entre el coche y el motor no es de herencia, sino de composición, lo que significa que el coche tiene un motor en lugar de ser un tipo de motor.

La composición ofrece ventajas como la modularidad, el bajo acoplamiento y la facilidad de mantenimiento. Permite crear objetos más complejos al combinar objetos más simples y mantener una estructura clara y organizada en el código.

4. DELEGACIÓN

La delegación es un concepto en la programación orientada a objetos que se refiere a la transferencia de responsabilidades de una clase a otra. En lugar de heredar directamente el comportamiento de una clase, una clase delega la ejecución de ciertas tareas a otra clase asociada.

En Python, la delegación se puede lograr utilizando la composición, donde una clase incluye una instancia de otra clase y utiliza sus métodos para realizar ciertas operaciones. La clase que delega la responsabilidad se conoce como la clase delegadora y la clase que realiza la tarea delegada se conoce como la clase delegada.

```
class Motor:
    def encender(self):
        print("El motor se ha encendido.")

    def apagar(self):
        print("El motor se ha apagado.")

class Coche:
    def __init__(self):
        self.motor = Motor()  # Instancia de la clase Motor

    def encender_coche(self):
        self.motor.encender()

    def apagar_coche(self):
        self.motor.apagar()

coche = Coche()
coche.encender_coche()  # Salida: El motor se ha encendido.
coche.apagar_coche()    # Salida: El motor se ha apagado.
```

En este ejemplo, la clase Coche tiene una instancia de la clase Motor y utiliza sus métodos encender() y apagar() para encender y apagar el coche. En lugar de heredar directamente el comportamiento del motor, la clase Coche delega estas tareas al objeto motor.

La delegación permite una mayor flexibilidad y modularidad en el diseño de clases, ya que las responsabilidades se pueden separar y asignar a diferentes clases según sea necesario. Además, si la implementación del motor cambia en el futuro, solo se necesita modificar la clase Motor sin afectar a la clase Coche.

5. COMPOSICIÓN VS DELEGACIÓN

La composición y la delegación son dos conceptos relacionados pero distintos en la programación orientada a objetos:

Composición: La composición implica la relación entre dos objetos donde uno de ellos contiene al otro como parte integral de su estructura. En la composición, un objeto se crea utilizando otros objetos como componentes o partes. El objeto contenedor tiene una referencia a los objetos contenidos y los utiliza para lograr su funcionalidad. La relación es de "tiene un" o "está compuesto por". La composición se logra mediante la creación de atributos en una clase que son instancias de otras clases. Por ejemplo, una clase Coche puede tener un atributo motor que es una instancia de la clase Motor.

Delegación: La delegación implica la transferencia de responsabilidades de una clase a otra. En lugar de heredar directamente el comportamiento de una clase, una clase delega la ejecución de ciertas tareas a otra clase asociada. En la delegación, un objeto utiliza otro objeto para llevar a cabo una operación específica en su nombre. La relación es de "utiliza" o "delega a". La delegación se logra a través de la inclusión de una instancia de una clase dentro de otra clase. Por ejemplo, una clase Coche puede tener un atributo motor que es una instancia de la clase Motor, y la clase Coche utiliza los métodos del objeto motor para encender y apagar el coche.

En resumen, la diferencia principal entre composición y delegación radica en el propósito de la relación entre objetos. La composición se centra en la creación de objetos complejos mediante la combinación de objetos más pequeños, donde el objeto contenedor es responsable de los objetos contenidos. Por otro lado, la delegación se centra en transferir responsabilidades a otros objetos, donde el objeto delegador utiliza otro objeto para realizar ciertas tareas en su nombre. Ambos conceptos ofrecen flexibilidad y reutilización de código, pero se aplican en contextos diferentes según las necesidades del diseño y la estructura del programa.