Paradigma de Objetos

Teoría y Aplicación Practica en python

Objetivos

- Entender los conceptos fundamentales del paradigma de objetos.
- Aprender cómo se implementan estos conceptos en Python.
- Ver ejemplos prácticos para ilustrar los conceptos.

Introducción al Paradigma de Objetos

¿Qué es el Paradigma de Objetos?

- El Paradigma Orientado a Objetos (POO) es un enfoque de programación que organiza el software en "objetos", que son entidades que encapsulan tanto datos (atributos) como comportamientos (métodos).
- Este paradigma se basa en conceptos clave como clases, herencia, encapsulamiento, polimorfismo y abstracción, permitiendo a los desarrolladores crear sistemas más modularizados, reutilizables y fáciles de mantener.

Breve historia de POO.

- La Programación Orientada a Objetos (POO) tiene sus raíces en la década de 1960 con la creación de Simula, desarrollado por Ole-Johan Dahl y Kristen Nygaard. Simula fue el primer lenguaje en introducir conceptos clave como clases y objetos, diseñados para simular sistemas complejos.
- En la década de 1980, Smalltalk, desarrollado en Xerox PARC, popularizó el uso de POO, destacando la importancia de la interacción entre objetos y el enfoque en la interfaz gráfica de usuario (GUI).
- El concepto ganó aún más tracción con la llegada de C++ en los años 80, que combinó la eficiencia de C con la flexibilidad de POO. Posteriormente, lenguajes como Java y Python continuaron impulsando la adopción de POO en la programación moderna, convirtiéndolo en un estándar para el desarrollo de software a gran escala.

Conceptos Fundamentales de POO

Clases y Objetos - ¿Qué es una clase?

- Una clase es una plantilla o modelo que define un conjunto de propiedades y comportamientos (métodos) comunes para un grupo de objetos.
- Es un plano abstracto que describe cómo deberían ser los objetos que se crean a partir de ella.
- Por ejemplo, una **clase** Coche puede tener **propiedades (atributos)** como color, marca, y modelo, y **comportamiento (métodos)** como acelerar y frenar.

Clases y Objetos - ¿Qué es un objeto?

- Un objeto es una instancia de una clase.
- Es una entidad concreta que se crea basándose en la definición de la clase y que contiene valores específicos para las propiedades definidas en la clase.
- Siguiendo con el ejemplo anterior, un **objeto** de la clase Coche podría ser un coche específico con color rojo, marca Toyota, y modelo Corolla.

Definición de una clase en Python

```python
# Definición de la clase
class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def acelerar(self):
        print(f"El {self.marca} {self.modelo} está acelerando.")

    def frenar(self):
        print(f"El {self.marca} {self.modelo} ha frenado.")
```

Creación de un Objeto en Python (Instanciación)

```python
# Creación de un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla", "rojo")

# Uso de los métodos del objeto
mi_coche.acelerar()
mi_coche.frenar()

# Acceso a los atributos del objeto
print(f"Mi coche es un {mi_coche.color} {mi_coche.marca} {mi_coche.modelo}.")
```

Encapsulamiento

- es uno de los principios fundamentales de la Programación Orientada a Objetos (POO) que se refiere a la práctica de agrupar los datos (atributos) y los métodos (comportamientos) que operan sobre esos datos dentro de una misma clase.
- Además, el encapsulamiento implica restringir el acceso directo a algunos de los componentes de un objeto, es decir, "ocultar" detalles internos y exponer solo lo necesario.

Importancia de Ocultar Detalles Internos

- **Protección de Datos**: Al ocultar los detalles internos, se evita que partes externas del código modifiquen directamente los atributos de un objeto, lo que podría llevar a un estado inconsistente o incorrecto. Por ejemplo, si un atributo solo puede tomar ciertos valores válidos, el encapsulamiento puede asegurar que este atributo solo se modifique a través de métodos que validen dichos valores.
- **Mantenimiento y Flexibilidad:** Ocultar los detalles internos permite que el código interno de una clase pueda cambiar sin afectar a las partes externas del código que dependen de esa clase. Esto facilita el mantenimiento del software, ya que los cambios en la implementación interna no requerirán modificaciones en todo el sistema.
- **Simplicidad y Abstracción:** El encapsulamiento ayuda a reducir la complejidad del sistema al permitir que los usuarios de una clase interactúen con ella a través de una interfaz clara y simple. Los detalles internos se mantienen ocultos, lo que facilita la comprensión y el uso de la clase sin necesidad de conocer su implementación interna.

Alcance

- En la programación orientada a objetos (POO), los atributos de una clase pueden ser públicos o privados, y esta distinción afecta cómo se accede a esos atributos desde fuera de la clase.

Atributos Públicos

- **Definición:** Los atributos públicos son aquellos que pueden ser accedidos y modificados desde fuera de la clase.
- En Python, los atributos públicos no tienen un prefijo especial en su nombre.
- **Acceso:** Se accede directamente usando el nombre del atributo.

```python
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre  # Atributo público
        self.edad = edad      # Atributo público

    def mostrar_info(self):
        print(f"Nombre: {self.nombre}, Edad: {self.edad}")

# Crear una instancia de Persona
persona = Persona("Juan", 30)

# Acceder al atributo público
print(persona.nombre)  # Output: Juan

# Modificar el atributo público
persona.edad = 31
print(persona.edad)    # Output: 31
```

Atributos Privados

- **Definición:** Los atributos privados son aquellos que no deberían ser accesibles directamente desde fuera de la clase.
- En Python, se indican con un prefijo doble de guiones bajos (‘__’), lo que ayuda a evitar el acceso accidental o no autorizado.
- **Acceso:** Se accede a través de métodos dentro de la misma clase (getters y setters) en lugar de directamente

```python
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre  # Atributo privado
        self.__edad = edad      # Atributo privado

    def mostrar_info(self):
        print(f"Nombre: {self.__nombre}, Edad: {self.__edad}")

    # Métodos para acceder a atributos privados
    def get_nombre(self):
        return self.__nombre

    def set_nombre(self, nombre):
        self.__nombre = nombre

    def get_edad(self):
        return self.__edad

    def set_edad(self, edad):
        self.__edad = edad
```

```python
# Crear una instancia de Persona
persona = Persona("Juan", 30)

# Acceso y modificación a través de métodos
print(persona.get_nombre())  # Output: Juan
persona.set_edad(31)
print(persona.get_edad())    # Output: 31

# Intentar acceder directamente a un atributo privado (no recomendado)
# print(persona.__nombre)  # Esto causará un error
```

Métodos de Clase y de Instancia

- Los **métodos de instancia** son aquellos que operan sobre una instancia de la clase y pueden acceder y modificar los atributos de la instancia.
- Los **métodos de clase (@classmethod)** reciben la clase como primer argumento en lugar de una instancia, y se utilizan para operar en datos que son compartidos por todas las instancias de la clase.

Métodos de Clase y de Instancia

```python
class Producto:
    iva = 0.21  # Atributo de clase

    def __init__(self, nombre, precio):
        self.nombre = nombre  # Atributo de instancia
        self.precio = precio

    def precio_con_iva(self):
        return self.precio * (1 + Producto.iva)

    @classmethod
    def actualizar_iva(cls, nuevo_iva):
        cls.iva = nuevo_iva
```

```python
# Crear una instancia de Producto
producto1 = Producto("Laptop", 1000)

# Método de instancia
print(producto1.precio_con_iva())  # Muestra: 1210.0

# Método de clase
Producto.actualizar_iva(0.18)
print(producto1.precio_con_iva())  # Muestra: 1180.0
```

Herencia

- El concepto de herencia es uno de los pilares fundamentales de la programación orientada a objetos (POO).
- La herencia permite crear una nueva clase (llamada clase hija o derivada) a partir de una clase existente (llamada clase padre o base), heredando así sus atributos y métodos.
- Esta característica es clave para promover la reutilización de código y para establecer relaciones jerárquicas entre clases.

Herencia (Ventajas)

- **Reutilización de código:** La herencia permite reutilizar el código de una clase existente en nuevas clases, evitando la duplicación de código y facilitando el mantenimiento del software.
- **Jerarquía y organización:** La herencia establece una relación jerárquica entre clases, lo que ayuda a organizar el código de manera más clara y lógica.
- **Polimorfismo:** La herencia es la base para el polimorfismo, otro pilar de la POO, que permite tratar a los objetos de distintas clases derivadas como si fueran instancias de la superclase.

Herencia (Como Funciona)

- Cuando una clase deriva de otra, hereda todos los atributos y métodos de la clase base.
- Esto significa que la subclase puede utilizar los mismos métodos y atributos que la clase base sin necesidad de redefinirlos.
- Sin embargo, la subclase puede:
  - **Añadir nuevos atributos y métodos:** Para extender la funcionalidad de la clase base.
  - **Sobrescribir métodos existentes:** Para cambiar el comportamiento de métodos heredados de la clase base.

Herencia

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass  # Método a ser sobrescrito en clases derivadas

class Perro(Animal):
    def hacer_sonido(self):
        return "Guau"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau"
```

```python
# Crear instancias de las clases derivadas
perro = Perro("Fido")
gato = Gato("Luna")

print(perro.hacer_sonido())  # Muestra: Guau
print(gato.hacer_sonido())   # Muestra: Miau
```

Polimorfismo

- El polimorfismo permite que diferentes clases implementen el mismo método de manera diferente.
- Esto permite usar el mismo método en distintas clases y obtener resultados distintos, dependiendo del contexto.

```python
class Vehiculo:
    def mover(self):
        pass

class Coche(Vehiculo):
    def mover(self):
        return "El coche se mueve sobre ruedas"

class Barco(Vehiculo):
    def mover(self):
        return "El barco navega en el agua"
```

```python
def mover_vehiculo(vehiculo):
    print(vehiculo.mover())

coche = Coche()
barco = Barco()

mover_vehiculo(coche)  # Muestra: El coche se mueve sobre ruedas
mover_vehiculo(barco)  # Muestra: El barco navega en el agua
```

Encapsulamiento y Métodos Especiales

- Encapsulamiento:
  - Los atributos y métodos pueden ser privados (__nombre) para proteger su acceso directo desde fuera de la clase.
  - Se accede a estos atributos a través de métodos especiales como getters y setters.
- Métodos Especiales (__str__, __repr__, etc.):
  - Python permite definir métodos especiales para controlar el comportamiento de las clases en contextos específicos, como al convertir un objeto a una cadena o al realizar comparaciones.

Encapsulamiento y Métodos Especiales

```python
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.__titular = titular  # Atributo privado
        self.__saldo = saldo      # Atributo privado

    def depositar(self, monto):
        self.__saldo += monto

    def retirar(self, monto):
        if monto <= self.__saldo:
            self.__saldo -= monto
        else:
            print("Fondos insuficientes")

    def __str__(self):
        return f"Cuenta de {self.__titular}, Saldo: {self.__saldo}"
```

```python
# Crear una instancia de CuentaBancaria
cuenta = CuentaBancaria("Juan", 5000)
cuenta.depositar(2000)
cuenta.retirar(1000)
print(cuenta)  # Muestra: Cuenta de Juan, Saldo: 6000
```

Hernecia (Método Super())

- La función **super()** se utiliza para llamar a métodos de la superclase desde una subclase.
- Esto es útil cuando quieres extender el comportamiento de un método en lugar de reemplazarlo por completo.

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

class Perro(Animal):
    def __init__(self, nombre, raza):
        super().__init__(nombre)
        self.raza = raza

mi_perro = Perro("Fido", "Labrador")
print(mi_perro.nombre)  # "Fido"
print(mi_perro.raza)    # "Labrador"
```

Composición

- La composición es una técnica para crear clases complejas a partir de objetos de otras clases.
- A diferencia de la herencia, la composición permite combinar múltiples objetos dentro de una nueva clase.
- La composición es una técnica de diseño en la que una clase se construye utilizando objetos de otras clases, en lugar de heredar de una clase base.
- Se dice que una clase "tiene" (has-a) objetos de otra clase como parte de su estructura interna.

Composición (Ventajas)

- **Flexibilidad:** La composición permite crear objetos complejos mediante la combinación de objetos más simples, lo que facilita la reutilización y modificación de componentes individuales sin afectar la clase que los contiene.
- **Mantenimiento:** Los cambios en una clase compuesta por otras clases son menos propensos a afectar a todo el sistema, ya que cada clase encapsula su propio comportamiento.
- **Desacoplamiento:** La composición promueve un diseño de software en el que las clases están menos acopladas entre sí, lo que mejora la modularidad y facilita la prueba y mantenimiento del código.

Composición (Como funciona)

- una clase se compone de uno o más objetos de otras clases,
- lo que significa que una clase puede utilizar los métodos y atributos de los objetos que contiene para definir su propio comportamiento.
- Esta relación se representa generalmente como un atributo que es una instancia de otra clase.

Composición

```python
class Motor:
    def __init__(self, tipo):
        self.tipo = tipo

class Rueda:
    def __init__(self, tamano):
        self.tamano = tamano

class Coche:
    def __init__(self, motor, rueda):
        self.motor = motor
        self.rueda = rueda

    def descripcion(self):
        return f"Coche con motor {self.motor.tipo} y ruedas de tamaño {self.rueda.tama
```

```python
motor = Motor("V8")
rueda = Rueda("17 pulgadas")
coche = Coche(motor, rueda)

print(coche.descripcion())  # Muestra: Coche con motor V8 y ruedas de tamaño 17
```

Composición (Conceptos Claves)

- **Relación "has-a":**
  - En composición, una clase "tiene" objetos de otra clase, a diferencia de la herencia donde una clase "es un" tipo de otra clase.
  - Por ejemplo, un Coche "tiene un" Motor y "tiene" Ruedas, pero un Coche no "es un" Motor.
- **Encapsulamiento:**
  - La composición promueve el encapsulamiento al mantener las clases separadas y encapsuladas dentro de una clase mayor.
  - Esto facilita la modificación y el mantenimiento, ya que los cambios en una clase no afectan directamente a otras clases.

Composición (Conceptos Claves)

- **Delegación:**
  - A menudo, una clase que utiliza composición delega ciertas operaciones a los objetos que contiene.
  - Por ejemplo, en el ejemplo anterior, el coche delega el inflado de las ruedas y el encendido del motor a sus componentes Rueda y Motor, respectivamente.
- **Composición vs Herencia:**
  - **Herencia:** Se utiliza cuando existe una clara relación jerárquica y es necesaria una especialización de la clase base. Es útil cuando quieres aprovechar o modificar un comportamiento existente.
  - **Composición:** Se prefiere cuando quieres construir objetos complejos mediante la combinación de objetos más simples y cuando quieres evitar el acoplamiento fuerte que puede venir con la herencia.

POO – Conceptos Avanzados

Resumen

- Concepto de Clases (Concretas) y Objetos
- Instanciación
- Encapsulamiento
- Atributos Públicos y Privados
- Métodos de Instancia y de clase
- Herencia
- Polimorfismo
- Composición

Métodos y Clases Abstractas

- las **clases abstractas** son plantillas que no se pueden instanciar directamente, sino que sirven como base para otras clases.
- Estas clases contienen **métodos abstractos**, que son métodos declarados pero no implementados.
- Las clases derivadas que heredan de una clase abstracta están obligadas a implementar los métodos abstractos.
- **Clase Abstracta:** No se puede instanciar y está diseñada para ser heredada por otras clases.
- **Método Abstracto:** Declarado sin una implementación. Las subclases deben sobrescribir estos métodos.

Métodos y Clases Abstractas - Características

- **Proporciona una estructura común:** Las clases abstractas permiten definir una estructura que debe ser seguida por las subclases, asegurando que ciertas funcionalidades sean implementadas.
- **No se puede instanciar:** Una clase abstracta no puede crear instancias de sí misma, es decir, no se pueden crear objetos directamente de ella.
- **Obligatoriedad en la implementación:** Si una clase hereda de una clase abstracta, debe implementar todos los métodos abstractos, a menos que también sea abstracta.

Métodos y Clases Abstractas - Ejemplo

- En Python, las clases abstractas se definen utilizando el módulo abc (Abstract Base Classes)

```python
from abc import ABC, abstractmethod

# Definición de una clase abstracta
class FormaGeometrica(ABC):

    @abstractmethod
    def calcular_area(self):
        pass

    @abstractmethod
    def calcular_perimetro(self):
        pass
```

Métodos y Clases Abstractas - Ejemplo

```python
# Implementación de una subclase concreta
class Circulo(FormaGeometrica):

    def __init__(self, radio):
        self.radio = radio

    def calcular_area(self):
        return 3.14 * self.radio ** 2

    def calcular_perimetro(self):
        return 2 * 3.14 * self.radio
```

```python
# Implementación de otra subclase concreta
class Rectangulo(FormaGeometrica):

    def __init__(self, largo, ancho):
        self.largo = largo
        self.ancho = ancho

    def calcular_area(self):
        return self.largo * self.ancho

    def calcular_perimetro(self):
        return 2 * (self.largo + self.ancho)
```

```python
# Uso de las clases concretas
circulo = Circulo(5)
print("Área del círculo:", circulo.calcular_area())
print("Perímetro del círculo:", circulo.calcular_perimetro())

rectangulo = Rectangulo(4, 6)
print("Área del rectángulo:", rectangulo.calcular_area())
print("Perímetro del rectángulo:", rectangulo.calcular_perimetro())
```

Métodos y Clases Abstractas - Ejemplo

- Resumen:
  - FormaGeometrica:
    - Es una clase abstracta que define la estructura que deben seguir todas las formas geométricas.
    - Declara dos métodos abstractos: calcular_area y calcular_perimetro.
  - Circulo y Rectangulo:
    - Son clases concretas que heredan de FormaGeometrica y deben implementar los métodos abstractos.
    - Cada clase tiene su propia implementación de cómo se calculan el área y el perímetro.

Ventajas de Usar Clases Abstractas

- **Consistencia:** Garantizan que todas las subclases implementen los métodos necesarios, proporcionando un diseño consistente y estructurado.
- **Reutilización de Código:** Permiten la reutilización de código al definir funcionalidades comunes en la clase base, y dejando que las subclases manejen los detalles específicos.
- **Flexibilidad y Extensibilidad:** Facilitan la creación de nuevas subclases que pueden ofrecer diferentes implementaciones para los métodos abstractos, sin modificar el código existente.

Interfaces

- una **interfaz** define un contrato que las clases deben cumplir.
- Este contrato especifica qué métodos deben implementarse, pero no cómo se implementan.
- A diferencia de las clases abstractas, una interfaz **no puede contener ningún código de implementación**, solo la firma de los métodos.
- **Interfaz:** Define un conjunto de métodos que una clase debe implementar.
- **Clase que implementa una interfaz:** Se compromete a proporcionar una implementación concreta para todos los métodos definidos en la interfaz.

Interfaces – Características Principales

- **Definen contratos claros:** Las interfaces garantizan que las clases que las implementan cumplan con un conjunto específico de métodos, independientemente de cómo estén implementados.
- **No contienen implementación:** A diferencia de las clases abstractas que pueden tener algunos métodos implementados, las interfaces solo declaran métodos sin proporcionar ninguna implementación.
- **Polimorfismo:** Las interfaces permiten que diferentes clases implementen el mismo conjunto de métodos, permitiendo a los objetos ser tratados de manera uniforme en cuanto a su comportamiento.

Interfaces – Ventajas

- **Definición clara de responsabilidades:** Las interfaces permiten definir contratos claros, asegurando que las clases implementen ciertas funcionalidades.
- **Flexibilidad:** Permiten que diferentes clases implementen los mismos métodos de maneras diferentes, facilitando el polimorfismo.
- **Mantenimiento y escalabilidad:** Al utilizar interfaces, se pueden cambiar las implementaciones de las clases sin afectar otras partes del sistema que dependen de esas interfaces, lo que facilita el mantenimiento y la escalabilidad.

Interfaces - Ejemplo

- Python no tiene interfaces formales como otros lenguajes, podemos simular este comportamiento utilizando clases abstractas puras (sin implementación) y el módulo ‘abc’

```python
from abc import ABC, abstractmethod

# Definición de una interfaz
class Operaciones(ABC):

    @abstractmethod
    def sumar(self, a, b):
        pass

    @abstractmethod
    def restar(self, a, b):
        pass
```

```python
# Clase que implementa la interfaz
class CalculadoraBasica(Operaciones):

    def sumar(self, a, b):
        return a + b

    def restar(self, a, b):
        return a - b

# Otra clase que implementa la interfaz con diferentes métodos
class CalculadoraAvanzada(Operaciones):

    def sumar(self, a, b):
        return a + b + 10  # Un comportamiento diferente

    def restar(self, a, b):
        return a - b - 5   # Un comportamiento diferente
```

Interfaces - Ejemplo

```
# Uso de las clases que implementan la interfaz
calc_basica = CalculadoraBasica()
print("Suma básica:", calc_basica.sumar(5, 3))
print("Resta básica:", calc_basica.restar(5, 3))

calc_avanzada = CalculadoraAvanzada()
print("Suma avanzada:", calc_avanzada.sumar(5, 3))
print("Resta avanzada:", calc_avanzada.restar(5, 3))
```

- **Operaciones:** Actúa como una interfaz que declara los métodos sumar y restar. Cualquier clase que implemente esta interfaz debe proporcionar su propia implementación de estos métodos.
- **CalculadoraBasica y CalculadoraAvanzada:** Son clases que implementan la interfaz Operaciones. A pesar de tener los mismos métodos, ofrecen diferentes comportamientos.

Clase Abstracta vs Interfaz

```python
from abc import ABC, abstractmethod

# Clase abstracta
class Animal(ABC):

    def __init__(self, nombre):
        self.nombre = nombre

    @abstractmethod
    def sonido(self):
        pass

    def mover(self):
        print(f"{self.nombre} se está moviendo")

# Clase concreta que hereda de la clase abstracta
class Perro(Animal):

    def sonido(self):
        return "Guau guau"
```

```python
# Uso de la clase abstracta y concreta
perro = Perro("Rex")
print(perro.sonido())  # "Guau guau"
perro.mover()          # "Rex se está moviendo"
```

```python
# Uso de las clases que implementan la interfaz
auto = Auto()
bicicleta = Bicicleta()

print(auto.acelerar())      # "El auto acelera"
print(auto.frenar())        # "El auto frena"
print(bicicleta.acelerar()) # "La bicicleta acelera"
print(bicicleta.frenar())   # "La bicicleta frena"
```

```python
from abc import ABC, abstractmethod

# Simulación de una interfaz
class Vehiculo(ABC):

    @abstractmethod
    def acelerar(self):
        pass

    @abstractmethod
    def frenar(self):
        pass

# Clase concreta que implementa la interfaz
class Auto(Vehiculo):

    def acelerar(self):
        return "El auto acelera"

    enar(self):
    turn "El auto frena"

ase concreta que implementa la interfaz
cleta(Vehiculo):

elerar(self):
    return "La bicicleta acelera"

    def frenar(self):
        return "La bicicleta frena"
```

Diferencias Clave

- **Clase Abstracta:**
  - Puede tener tanto métodos abstractos como métodos con implementación.
  - Puede tener atributos.
  - Se utiliza cuando se quiere compartir alguna funcionalidad común entre las clases derivadas, pero también forzar a las clases hijas a implementar ciertos métodos.
- **Interfaz:**
  - Solo declara métodos abstractos, sin implementación.
  - No tiene atributos ni métodos implementados.
  - Se utiliza para definir un contrato que las clases deben seguir, sin proporcionar ningún código base o funcionalidad común.

Delegación

- La **delegación** es un **patrón de diseño** en la POO en el que un objeto delega la responsabilidad de una tarea a otro objeto.
- En lugar de implementar una funcionalidad directamente, un objeto **puede delegar esa responsabilidad** a un "ayudante" o colaborador, que es otro objeto que realiza la tarea en su nombre.
- Este patrón es útil para **evitar duplicación de código** y para adherirse al **principio de responsabilidad única**, permitiendo que un objeto se enfoque en su propósito principal y delegue tareas específicas a otros objetos más adecuados para manejarlas.

Delegación – Características Principales

- **Reutilización de Código:** La delegación permite reutilizar código existente en lugar de reimplementarlo, promoviendo la modularidad.
- **Desacoplamiento:** Mediante la delegación, un objeto puede delegar una tarea sin tener que preocuparse por cómo se realiza, lo que desacopla la lógica de implementación.
- **Flexibilidad:** Los objetos pueden cambiar su comportamiento en tiempo de ejecución al delegar tareas a diferentes objetos.

Delegación - ejemplo

```python
class Motor:
    def arrancar(self):
        return "El motor está arrancado"

    def detener(self):
        return "El motor está detenido"

class Coche:
    def __init__(self):
        self.motor = Motor()  # Delegación

    def encender(self):
        return self.motor.arrancar()  # Delegación al método arrancar del motor

    def apagar(self):
        return self.motor.detener()  # Delegación al método detener del motor

# Uso de la delegación
coche = Coche()
print(coche.encender())  # "El motor está arrancado"
print(coche.apagar())    # "El motor está detenido"
```

Delegación - Comparación con Otros Conceptos

- **Composición:** Mientras que en la delegación un objeto se apoya en otro para realizar una tarea específica, en la composición, los objetos más pequeños forman parte de un objeto más grande, que combina sus comportamientos.
- **Herencia:** En lugar de heredar comportamientos, en la delegación un objeto simplemente utiliza otro objeto para realizar algunas de sus tareas. La delegación es una alternativa flexible a la herencia, ya que no establece una relación tan estrecha entre las clases.
- **Polimorfismo:** La delegación puede utilizarse junto con el polimorfismo, permitiendo que diferentes objetos que implementan la misma interfaz (o métodos similares) se utilicen indistintamente, pero delegando tareas específicas.

Clases y Métodos Finales

- los conceptos de **clases finales** y **métodos finales** se utilizan para restringir la herencia y la sobrescritura (override) en las clases derivadas.
- Esencialmente, una clase o un método marcado como "final" no puede ser extendido o modificado en clases hijas.
- Estos conceptos se aplican para:
  - **Evitar que una clase sea heredada**.
  - **Prevenir que un método sea sobrescrito** en una clase derivada.
  - **Proteger la integridad del código** cuando un comportamiento no debe ser alterado.

Clases y Métodos Finales

- **Clases Finales:**
  - Una clase final es una clase que no puede ser heredada. Esto significa que ninguna otra clase puede derivarse de una clase final.
  - En Python, no existe una palabra clave **final**, pero se puede simular el comportamiento mediante la creación de una metaclase.
- **Métodos Finales:**
  - Un método final es un método que no puede ser sobrescrito (overriden) en una clase derivada.
  - Python no tiene una palabra clave final, pero se puede implementar esta restricción mediante una convención en la implementación.

Clases y Métodos Finales - Ventajas

- **Protección del Comportamiento Crítico:** Si un método o clase tiene un comportamiento que es fundamental para el funcionamiento del programa, marcarlo como final asegura que este comportamiento no se vea alterado accidentalmente en futuras extensiones de la clase.
- **Simplicidad:** Restringir la herencia y sobrescritura puede simplificar la arquitectura del código, asegurando que ciertas partes de la lógica permanezcan inalteradas.
- **Contratos Fuertes:** Al definir métodos finales, se establece un contrato fuerte entre la clase base y sus derivados, obligando a las clases hijas a adherirse a ciertas implementaciones.

Clases y Métodos Estáticos

- **Método estático:** Es un método que pertenece a la clase en sí, en lugar de a una instancia específica de la clase. No puede acceder a los atributos de instancia ni a otros métodos de instancia.
- **Clase estática:** Aunque Python no tiene un concepto de "clase estática" como tal (como en otros lenguajes como C#), se puede lograr un comportamiento similar utilizando módulos o definiendo clases que contengan únicamente métodos y atributos estáticos.

Clases y Métodos Estáticos

- Un **método estático** en Python se define utilizando el decorador **@staticmethod**.
- Estos métodos no requieren acceso a los datos de instancia (es decir, no tienen un self como primer argumento).
- Son útiles cuando una función dentro de una clase no necesita acceder a los atributos o métodos de la clase.

Ventajas

- **Eficiencia:** Los métodos estáticos son más eficientes cuando no se necesita acceder a los datos de instancia o de clase, ya que no tienen la sobrecarga de la vinculación a una instancia específica.
- **Organización:** Agrupar funciones relacionadas dentro de una clase estática o en un módulo mejora la organización del código.
- **Reutilización:** Los métodos estáticos pueden ser reutilizados sin la necesidad de instanciar la clase, lo que es útil para operaciones comunes o utilitarias.
- **Simplicidad:** Para funciones que no dependen de instancias, el uso de métodos estáticos simplifica la estructura de la clase y el código.