Guía Completa de Programación Orientada a Objetos (POO) en Python

1. ¿Qué es la Programación Orientada a Objetos (POO)?

La POO es un paradigma de programación que utiliza **objetos** para representar datos y comportamientos, de la misma manera en que lo hacemos en el mundo real. En lugar de escribir una serie de instrucciones para procesar datos, la POO nos permite agrupar los datos y las funciones que operan sobre ellos en unidades lógicas llamadas **objetos**.

Clase vs. Objeto:

- Una **clase** es el plano, la plantilla o el molde para crear objetos. Define un conjunto de atributos (características) y métodos (comportamientos).
- Un **objeto** es una instancia de una clase. Es una entidad concreta que puedes crear a partir del plano de la clase.

Ejemplo:

- La **clase** podría ser "Coche". Define que todos los coches tienen marca, modelo y año.
- El **objeto** podría ser "mi_coche", una instancia específica de la clase Coche, con los valores: marca="Toyota", modelo="Corolla", año=2020.

2. Fundamentos de las Clases en Python

Atributos y Métodos

- **Atributos:** Son las variables que pertenecen a una clase o a un objeto. Representan las características o el estado del objeto.
- **Métodos:** Son las funciones que pertenecen a una clase y definen el comportamiento de los objetos.

El método especial __init__ es el **constructor**. Se llama automáticamente cuando creas un nuevo objeto y se usa para inicializar sus atributos. El parámetro self se refiere a la instancia del objeto y es necesario en todos los métodos de instancia.

```python
# Ejemplo de clase, atributos y métodos
class Coche:
    # Método constructor para inicializar el objeto
    def __init__(self, marca, modelo, anio):
        self.marca = marca
        self.modelo = modelo
        self.anio = anio


    # Método de instancia para mostrar los datos del coche
    def mostrar_datos(self):
        print(f"Marca: {self.marca}, Modelo: {self.modelo}, Año: {self.anio}")


# Creación de un objeto (instancia de la clase Coche)
mi_coche = Coche("Toyota", "Corolla", 2020)


# Acceder a un atributo del objeto
print(f"Mi coche es un {mi_coche.marca} {mi_coche.modelo}.")


# Llamar a un método del objeto
mi_coche.mostrar_datos()
```

imprime

```
Mi coche es un Toyota Corolla.
Marca: Toyota, Modelo: Corolla, Año: 2020
```

resumen

Elemento	¿Qué hace?
`class Coche:`	Define la clase
`__init__`	Inicializa el objeto con atributos
`self`	Se refiere al objeto actual
`mostrar_datos()`	Método para imprimir info del coche
`mi_coche = Coche(...)`	Crea un objeto de la clase
`mi_coche.marca`	Accede al atributo
`mi_coche.mostrar_datos()`	Llama al método del objeto

3. Los Cuatro Pilares de la POO

A. Herencia

La herencia permite que una clase (clase hija o subclase) herede atributos y métodos de otra clase (clase padre o superclase). Esto promueve la reutilización de código.

```python
# La clase Coche será la clase padre
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo


    def mostrar_datos(self):
        print(f"Marca: {self.marca}, Modelo: {self.modelo}")


# CocheElectrico es la clase hija que hereda de Coche
class CocheElectrico(Coche):
    def __init__(self, marca, modelo, capacidad_bateria):
        # Llama al constructor de la clase padre
        super().__init__(marca, modelo)
        self.capacidad_bateria = capacidad_bateria


    # Sobreescritura del método (Overriding)
    def mostrar_datos(self):
        # Llama al método de la clase padre y añade su propio comportamiento
        super().mostrar_datos()
        print(f"Capacidad de batería: {self.capacidad_bateria} kWh")


mi_coche_electrico = CocheElectrico("Tesla", "Model 3", 75)
mi_coche_electrico.mostrar_datos()
```

imprime

```
Marca: Tesla, Modelo: Model 3
Capacidad de batería: 75 kWh
```

resumen

Elemento	¿Qué representa?	¿Qué hace?
`Coche`	Clase padre	Define marca y modelo
`CocheElectrico(Coche)`	Clase hija	Hereda de Coche
`super().__init__()`	Llama al constructor del padre	Evita repetir código
`super().mostrar_datos()`	Usa el método del padre	Luego añade más
`capacidad_bateria`	Nuevo atributo	Solo lo tiene `CocheElectrico`

B. Encapsulamiento

El encapsulamiento consiste en ocultar el estado interno de un objeto y exponer solo una interfaz controlada y segura para interactuar con él. En Python, esto se logra con la convención de prefijos:

- _atributo: Indicador de que el atributo es "protegido" (solo para uso interno).
- __atributo: Indicador de que el atributo es "privado". Python lo renombra (name mangling) para evitar colisiones.

```python
# Ejemplo de encapsulamiento con getters y setters
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial):
        self.__titular = titular  # Atributo "privado"
        self.__saldo = saldo_inicial  # Atributo "privado"


    # Getter para obtener el saldo (interfaz controlada)
    def get_saldo(self):
        return self.__saldo


    # Setter para modificar el saldo (interfaz controlada)
    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad
            print(f"Depósito de {cantidad} realizado. Nuevo saldo: {self.__saldo}")
        else:
            print("El monto a depositar debe ser positivo.")


mi_cuenta = CuentaBancaria("Juan Pérez", 1000)
# Intentar acceder directamente al atributo privado (no es la forma correcta)
# print(mi_cuenta.__saldo) # Esto causaría un error
print(f"Saldo actual: {mi_cuenta.get_saldo()}")
mi_cuenta.depositar(500)

```

imprime

```
Saldo actual: 1000
Depósito de 500 realizado. Nuevo saldo: 1500
```

resumen

Elemento	¿Qué hace?	¿Por qué es importante?
`__atributo`	Atributo privado	Protege los datos del objeto
`get_saldo()`	Getter	Permite leer el valor de forma segura
`depositar()`	Setter	Permite modificar el valor con validación
Encapsulamiento	Oculta los detalles internos	Da seguridad y control

C. Polimorfismo

El polimorfismo (muchas formas) permite que objetos de diferentes clases respondan a la misma llamada de método de manera diferente. En Python, esto es natural, ya que no se requiere declarar el tipo de objeto.

```python
# Ejemplo de polimorfismo
class Gato:
    def hablar(self):
        return "Miau"


class Perro:
    def hablar(self):
        return "Guau"


class Vaca:
    def hablar(self):
        return "Muu"


# Una función que acepta cualquier objeto con un método 'hablar'
def sonido_animal(animal):
    print(animal.hablar())


# Creando instancias de las diferentes clases
gato = Gato()
perro = Perro()
vaca = Vaca()


# La misma función 'sonido_animal' funciona con cada objeto
sonido_animal(gato)
sonido_animal(perro)
sonido_animal(vaca)
```

imprime

```
Miau
Guau
Muu
```

resumen

Clase	Método `hablar()` devuelve
`Gato`	`"Miau"`
`Perro`	`"Guau"`
`Vaca`	`"Muu"`

D. Abstracción

La abstracción se enfoca en mostrar solo los detalles esenciales de un objeto y ocultar la complejidad interna. En Python, las clases abstractas, que no pueden ser instanciadas directamente, nos ayudan a definir una interfaz común para las clases hijas.

```python
from abc import ABC, abstractmethod

# La clase Vehiculo es una clase abstracta
class Vehiculo(ABC):
    @abstractmethod
    def arrancar(self):
        pass


    @abstractmethod
    def detener(self):
        pass

# La clase Coche_Abstracto debe implementar los métodos abstractos
class Coche_Abstracto(Vehiculo):
    def arrancar(self):
        print("El coche ha arrancado.")


    def detener(self):
        print("El coche se ha detenido.")


# La clase Moto_Abstracta también debe implementarlos
class Moto_Abstracta(Vehiculo):
    def arrancar(self):
        print("La moto ha arrancado.")


    def detener(self):
        print("La moto se ha detenido.")


# No se puede crear un objeto de la clase abstracta Vehiculo
# vehiculo = Vehiculo() # Esto causaría un error


coche = Coche_Abstracto()
moto = Moto_Abstracta()


coche.arrancar()
moto.detener()
```

imprime

```
El coche ha arrancado.
La moto se ha detenido.
```

4. Métodos Mágicos (Dunder Methods)

Los métodos con doble guion bajo, como __init__ o __str__, son métodos especiales que Python invoca en momentos específicos.

El método __str__ se usa para definir la representación en cadena de texto de un objeto.

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

    # Método dunder para representar el objeto como una cadena legible
    def __str__(self):
        return f"Persona(nombre='{self.nombre}', edad={self.edad})"


p = Persona("Ana", 30)
# Cuando se imprime el objeto, se llama automáticamente a __str__
print(p)

```

imprime

```
Persona(nombre='Ana', edad=30)
```

resumen

Parte	¿Qué hace?
`__init__`	Constructor: inicializa los datos del objeto
`self.nombre` / `self.edad`	Atributos del objeto
`__str__`	Método especial para mostrar el objeto como texto
`print(p)`	Llama automáticamente a `p.__str__()`