Programación Orientada a Objetos (POO) en Python

La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el código en torno a objetos, en lugar de funciones o lógica. Python es un lenguaje que permite implementar la POO de manera flexible y sencilla.

1. Conceptos Clave de la Programación Orientada a Objetos

1. **Clase**: Una clase es una plantilla o blueprint que define la estructura y el comportamiento de los objetos.
2. **Objeto**: Un objeto es una instancia de una clase; es la entidad concreta que se crea a partir de la clase.
3. **Atributo**: Un atributo es una variable que pertenece a una clase u objeto.
4. **Método**: Un método es una función que pertenece a una clase y opera sobre los objetos de esa clase.
5. **Encapsulamiento**: Proceso de agrupar datos y métodos que operan sobre esos datos en una sola unidad (clase). Ayuda a ocultar la implementación interna de los datos.
6. **Herencia**: Permite crear una nueva clase a partir de una clase existente, reutilizando el código y extendiéndolo.
7. **Polimorfismo**: Permite que diferentes clases puedan ser tratadas como si fueran del mismo tipo a través de métodos con el mismo nombre.
8. **Abstracción**: Proceso de ocultar los detalles internos y exponer solo lo necesario para la interfaz pública.

2. Definiendo una Clase y Creando Objetos

Una clase se define usando la palabra clave class, y los objetos se crean instanciando esa clase.

```
# Definición de una clase en Python
class Persona:
    def __init__(self, nombre, edad):  # Constructor de la clase
        self.nombre = nombre            # Atributo nombre
        self.edad = edad                # Atributo edad

    def saludar(self):  # Método para saludar
        print(f"Hola, me llamo {self.nombre} y tengo {self.edad}
años.")

# Creación de objetos (instancias de la clase Persona)
persona1 = Persona("Juan", 25)
persona2 = Persona("Ana", 30)

# Llamada a métodos
persona1.saludar()  # "Hola, me llamo Juan y tengo 25 años."
persona2.saludar()  # "Hola, me llamo Ana y tengo 30 años."
```

Explicación:

- El método especial __init__ es el **constructor**. Se llama automáticamente cuando creas un nuevo objeto.
- Los atributos de instancia (nombre y edad) son accesibles y modificables a través de self.

---

3. Encapsulamiento en Python

En Python, el encapsulamiento se consigue usando variables privadas (prefijando con __) o protegiendo atributos (prefijando con _).

```
class Coche:
    def __init__(self, marca, modelo, velocidad_maxima):
        self.marca = marca
        self.__modelo = modelo  # Atributo privado
        self._velocidad_maxima = velocidad_maxima  # Atributo
protegido

    def mostrar_modelo(self):
        print(f"El modelo es {self.__modelo}")

# Instanciando la clase
mi_coche = Coche("Toyota", "Corolla", 180)
mi_coche.mostrar_modelo()  # "El modelo es Corolla"
# Intentar acceder directamente al atributo privado genera un error:
# print(mi_coche.__modelo)  # AttributeError
```

4. Herencia en Python

La **herencia** permite a una clase hija heredar atributos y métodos de una clase padre. Python admite herencia simple y múltiple.

```
# Clase Padre
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass  # Método que será sobreescrito en las clases hijas

# Clase Hija
class Perro(Animal):
    def hacer_sonido(self):
        return "Guau"
```

```python
# Clase Hija
class Gato(Animal):
    def hacer_sonido(self):
        return "Miau"

# Uso de herencia
mi_perro = Perro("Bobby")
mi_gato = Gato("Mimi")

print(mi_perro.hacer_sonido())  # "Guau"
print(mi_gato.hacer_sonido())   # "Miau"
```

Explicación:

- La clase Perro y Gato heredan de la clase Animal, pero sobreescriben el método hacer_sonido.

---

5. Polimorfismo en Python

El polimorfismo permite tratar a objetos de diferentes clases de manera uniforme.

```python
class Ave:
    def volar(self):
        print("El ave está volando")

class Pinguino(Ave):
    def volar(self):
        print("El pingüino no puede volar, pero puede nadar")

def hacer_volar(ave):
    ave.volar()

pajaro = Ave()
pinguino = Pinguino()

hacer_volar(pajaro)    # "El ave está volando"
hacer_volar(pinguino)  # "El pingüino no puede volar, pero puede
nadar"
```

6. Abstracción

La abstracción oculta los detalles internos y solo expone lo esencial. En Python, se puede lograr usando clases abstractas y métodos abstractos con el módulo abc.

```python
from abc import ABC, abstractmethod

class FiguraGeometrica(ABC):
    @abstractmethod
    def area(self):
        pass

class Cuadrado(FiguraGeometrica):
    def __init__(self, lado):
        self.lado = lado

    def area(self):
        return self.lado * self.lado

# No se puede instanciar una clase abstracta
# figura = FiguraGeometrica()  # Error

cuadrado = Cuadrado(4)
print(cuadrado.area())  # "16"
```

7. Propiedades y Métodos Especiales

Python permite la creación de **propiedades** para controlar el acceso a los atributos y agregar lógica cuando se accede o se modifica un valor.

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

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

persona = Persona("Pedro", 25)
print(persona.nombre)  # "Pedro"
persona.nombre = "Juan"
print(persona.nombre)  # "Juan"
```

8. Ejemplo Completo de POO en Python

Vamos a crear un sistema simple de gestión de empleados utilizando clases, herencia, y encapsulamiento.

```python
class Empleado:
    def __init__(self, nombre, salario):
        self.nombre = nombre
        self._salario = salario

    def mostrar_informacion(self):
        return f"Empleado: {self.nombre}, Salario: {self._salario}"

    @property
    def salario(self):
        return self._salario

    @salario.setter
    def salario(self, nuevo_salario):
        if nuevo_salario < 0:
            print("El salario no puede ser negativo")
        else:
            self._salario = nuevo_salario

class Gerente(Empleado):
    def __init__(self, nombre, salario, departamento):
        super().__init__(nombre, salario)
        self.departamento = departamento

    def mostrar_informacion(self):
        return f"Gerente: {self.nombre}, Departamento:
{self.departamento}, Salario: {self.salario}"

# Uso de las clases
empleado1 = Empleado("Carlos", 2500)
empleado1.salario = 2700  # Cambiando el salario
print(empleado1.mostrar_informacion())

gerente1 = Gerente("Ana", 5000, "Ventas")
print(gerente1.mostrar_informacion())
```

Asociación entre Dos Clases en Programación Orientada a Objetos

La **asociación** es una relación entre dos clases en la que los objetos de una clase se relacionan con objetos de otra clase. No implica ninguna dependencia fuerte (como en la composición o agregación), pero permite que dos clases colaboren. Este tipo de relación es muy común en programación orientada a objetos, y representa situaciones como: "un empleado pertenece a una empresa", "un estudiante asiste a una escuela", etc.

En una asociación, las clases pueden:

- Comunicarse entre sí (una clase puede tener una referencia de la otra).
- Mantener independencia (ninguna clase "posee" a la otra).

Ejemplo de Asociación entre Dos Clases

Imaginemos que tenemos dos clases: **Persona** y **Coche**. Una **Persona** puede tener un **Coche**, pero **Coche** y **Persona** no dependen completamente el uno del otro.

Código:

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
        self.coche = None  # No necesariamente tiene coche cuando se
crea

    def asignar_coche(self, coche):
        self.coche = coche

    def mostrar_info(self):
        if self.coche:
            print(f"{self.nombre} tiene un coche {self.coche.marca}
{self.coche.modelo}.")
        else:
            print(f"{self.nombre} no tiene coche.")


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

# Creando los objetos
persona1 = Persona("Carlos")
coche1 = Coche("Toyota", "Corolla")

# Asociando la persona con su coche
persona1.asignar_coche(coche1)

# Mostrando la información
persona1.mostrar_info()  # Output: Carlos tiene un coche Toyota
Corolla.
```

Explicación:

- La clase Persona tiene un atributo coche, que inicialmente es None (es decir, la persona puede no tener coche).

- La clase Coche no tiene ninguna referencia a Persona, lo que muestra que no existe una dependencia estricta entre ambas clases.
- El método asignar_coche establece una relación entre un objeto Persona y un objeto Coche, creando una asociación entre ellos.

Tipos de Asociación

1. **Unidireccional**: Como en el ejemplo anterior, donde solo la clase Persona conoce la relación con Coche, pero Coche no conoce a Persona.
2. **Bidireccional**: Ambas clases mantienen una referencia entre sí. Ejemplo: si Coche también tuviera una referencia al propietario (una Persona).

Ejemplo de Asociación Bidireccional:

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
        self.coche = None

    def asignar_coche(self, coche):
        self.coche = coche
        coche.propietario = self  # El coche también conoce a la
persona
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.propietario = None  # Al principio, el coche no tiene
propietario

    def mostrar_info(self):
        if self.propietario:
            print(f"Este {self.marca} {self.modelo} pertenece a
{self.propietario.nombre}.")
        else:
            print(f"Este {self.marca} {self.modelo} no tiene
propietario.")

# Creando los objetos
persona1 = Persona("Carlos")
coche1 = Coche("Toyota", "Corolla")

# Asociando la persona con su coche y viceversa
persona1.asignar_coche(coche1)

# Mostrando información desde ambos lados
persona1.mostrar_info()  # "Carlos tiene un coche Toyota Corolla."
coche1.mostrar_info()    # "Este Toyota Corolla pertenece a Carlos."
```

En este caso, la asociación es bidireccional, ya que tanto Persona como Coche se conocen entre sí.

Asociación Uno a Muchos / Muchos a Uno en Programación Orientada a Objetos

La relación **Uno a Muchos** implica que una instancia de una clase está asociada con múltiples instancias de otra clase. Por otro lado, la relación **Muchos a Uno** es lo inverso, donde varias instancias de una clase están asociadas con una instancia de otra clase.

Ejemplo de Asociación Uno a Muchos y Muchos a Uno

Vamos a implementar un ejemplo en el que una **Empresa** puede tener **muchos empleados** (uno a muchos) y **cada empleado** trabaja en una sola empresa (muchos a uno).

Código:

```
class Empresa:
    def __init__(self, nombre):
        self.nombre = nombre
        self.empleados = []  # Lista para almacenar empleados
(relación uno a muchos)

    def agregar_empleado(self, empleado):
        self.empleados.append(empleado)
        empleado.empresa = self  # Relación muchos a uno, el empleado
conoce su empresa

    def mostrar_empleados(self):
        print(f"Empleados de {self.nombre}:")
        for empleado in self.empleados:
            print(f"- {empleado.nombre}")


class Empleado:
    def __init__(self, nombre):
        self.nombre = nombre
        self.empresa = None  # Relación muchos a uno, al principio no
tiene empresa asignada

    def mostrar_empresa(self):
        if self.empresa:
            print(f"{self.nombre} trabaja en {self.empresa.nombre}.")
        else:
            print(f"{self.nombre} no tiene empresa asignada.")
```

```python
# Creando una empresa
empresa1 = Empresa("TechCorp")

# Creando empleados
empleado1 = Empleado("Ana")
empleado2 = Empleado("Luis")
empleado3 = Empleado("Maria")

# Asociando los empleados a la empresa (uno a muchos)
empresa1.agregar_empleado(empleado1)
empresa1.agregar_empleado(empleado2)
empresa1.agregar_empleado(empleado3)

# Mostrando empleados de la empresa (uno a muchos)
empresa1.mostrar_empleados()

# Mostrando empresa de cada empleado (muchos a uno)
empleado1.mostrar_empresa()  # Ana trabaja en TechCorp
empleado2.mostrar_empresa()  # Luis trabaja en TechCorp
empleado3.mostrar_empresa()  # Maria trabaja en TechCorp
```

Explicación:

- La clase Empresa tiene una lista de empleados, lo que representa la relación **uno a muchos**. Una empresa puede tener múltiples empleados.
- La clase Empleado tiene un atributo empresa, lo que representa la relación **muchos a uno**. Un empleado solo puede estar asociado con una empresa.
- Cuando se agrega un empleado a una empresa, el empleado también es asignado a esa empresa.

Diagrama Conceptual:

- **Uno a Muchos (Empresa -> Empleados)**: Una empresa puede tener muchos empleados.
- **Muchos a Uno (Empleado -> Empresa)**: Un empleado pertenece a una única empresa.

Salida del Código:

```
Empleados de TechCorp:
- Ana
- Luis
- Maria
Ana trabaja en TechCorp.
Luis trabaja en TechCorp.
Maria trabaja en TechCorp.
```

Este ejemplo ilustra cómo modelar relaciones **uno a muchos** y **muchos a uno** en programación orientada a objetos, donde una empresa puede tener varios empleados y cada empleado solo pertenece a una empresa.