# Unidad 5

## POO (Programación orientada a objetos)

En la programación, especialmente con lenguajes como Python, el **modelo computacional de objetos** (también conocido como Programación Orientada a Objetos o POO) es una forma de diseñar y escribir software que organiza el código alrededor de **objetos** en lugar de funciones y lógica.

Pensemos en el mundo real: todo lo que nos rodea son objetos con características y comportamientos. La POO intenta simular esto en el software.

Supongamos que queremos representar autos en nuestro programa, sabemos que un auto tiene como características (**atributos**), una marca, un modelo, un color, etc. Además, tiene asociado varios comportamientos (**métodos**) como arrancar, acelerar, frenar, etc. Todo esto nos permite definir una plantilla o un plano de como debe ser un objeto que represente un auto dentro de nuestro código. Esta plantilla es lo que se conoce como **clase**.

## Clase

Una clase no es un objeto en si misma, sino una definición de como deben ser los objetos de cierto tipo. Define los atributos y los métodos que tendrán los objetos que se creen a partir de ella.

Siguiendo con el ejemplo de los autos, podríamos definir nuestra clase Auto de la sig. manera:

```
class Auto:
    #Atributos
    def __init__(self, color, marca, modelo):
        self.color = color
        self.marca = marca
        self.modelo = modelo
    #Metodos
    def arrancar(self):
        return "El auto: " + self.marca + " modelo: " + self.modelo + " ha arrancado."
    def frenar(self):
        return "El auto: " + self.marca + " modelo: " + self.modelo + " ha frenado."
```

## Objeto

Un **objeto** es una **instancia concreta** de una clase. Es decir, es un "ejemplar" real creado a partir del plano de la clase. Podremos crear múltiples objetos a partir de la misma clase, y cada uno de ellos tendrá sus propios valores para sus atributos.

Por ejemplo, podríamos representar varios autos de la sig. manera:

```
mi_auto = Auto("rojo", "Ford", "Fiesta")
tu_auto = Auto("azul", "Toyota", "Corolla")
```

Ambos autos son objetos creados a partir de la clase Auto, pero mi_auto es rojo y tu_auto es azul. Esta diferencia ocurre porque fueron creados con atributos distintos.

## Atributo

Los **atributos** son las **características o propiedades** de un objeto. Son como variables que pertenecen a un objeto y almacenan información sobre él.

Por ejemplo, en los objetos que creamos anteriormente:

- Color, marca y modelo son los atributos.
- Para mi_auto los atributos son “rojo”, “Ford” y “Fiesta”.
- Para tu_auto los atributos son “azul”, “Toyota” y “Corolla”.

Podremos acceder a los atributos de un objeto usando la notación de punto : objeto.atributo

Ejemplo:

```
print(mi_auto.color)    # Muestra por pantalla: rojo
print(tu_auto.marca)    # Muestra por pantalla: Toyota
```

## Método

Son las acciones o comportamientos que un objeto puede realizar. Son funciones que pertenecen a una clase y operan sobre los datos (atributos) del objeto.

Por ejemplo, en la clase Auto definida anteriormente, arrancar() y frenar() son métodos.

Podremos llamar a los métodos de un objeto utilizando la notación de punto: objeto.metodo()

Ejemplo:

```
print(mi_auto.arrancar()) # Salida: El Ford Fiesta de color rojo ha arrancado.
print(tu_auto.frenar())   # Salida: El Toyota Corolla de color azul ha frenado.
```

## Mensaje entre Objetos

Los objetos pueden interactuar entre sí enviándose **mensajes**. Un mensaje es esencialmente una llamada a un método de otro objeto. Cuando un objeto envía un mensaje, le está pidiendo a otro objeto que realice una acción específica o que le proporcione información.

Imaginemos un caso donde necesitemos que un conductor encienda nuestro auto. Vamos a necesitar un objeto que representé a un conductor y que luego ese objeto le envié un mensaje al objeto auto para que este “arranque”.

Ejemplo:

```
class Conductor:
    def __init__(self, nombre):
        self.nombre = nombre
    def encender_auto(self, auto):
        print(f"{self.nombre} está intentando encender el auto.")
        print(auto.arrancar()) # El Conductor envía un mensaje al objeto Auto
                               # (llama al método arrancar)
```

Definimos la clase Conductor donde el nombre es un atributo y encender_auto es un método. Ahora vamos a crear el objeto mi_conductor para que envie un mensaje al objeto mi_auto.

```
mi_conductor = Conductor("Juan")
mi_conductor.encender_auto(mi_auto)
# Salida:
# Juan está intentando encender el auto.
# El Ford Fiesta de color rojo ha arrancado.
```

## Principios fundamentales de la POO

La programación orientada a objetos esta basada en 4 principios o pilares básicos:

- Encapsulamiento
- Abstracción
- Herencia
- Polimorfismo

### Encapsulamiento

El encapsulamiento hace referencia al ocultamiento del estado interno de una clase al exterior. Dicho de otra manera, encapsular consiste en que hacer que los atributos de una clase no se puedan acceder ni modificar desde fuera. El único que puede acceder a ellos es el propio objeto.

Python no cuenta con modificadores de acceso como en otros lenguajes de programación, sin embargo, se puede indicar el acceso utilizando guiones bajos como prefijos.

Niveles de acceso:

- **Publico**: nombre (sin guiones). Acceso libre, permite acceder y modificar atributos de un objeto.
- **Protegido**: _nombre (un guion bajo). Acceso permitido, se utiliza por convención entre los desarrolladores para indicar que un atributo o método no debe ser accesible desde fuera. Solo deberían poder usarse de forma interna por el objeto.
- **Privado**: __nombre (dos guiones bajos). Acceso privado, Python lo “oculta” para que sea inaccesible desde fuera.

Veamos un ejemplo:

```
class Persona:
    def __init__(self, nombre, dni):
        self.nombre = nombre   # público
        self._dni = dni   # protegido
        self.__clave = "secreta"   # privado
    def mostrar_clave(self):
        print(f"Clave: {self.__clave}")

persona = Persona("Lucía","22333444")
print(persona.nombre)        # Accesible
print(persona._dni)          # Accesible, pero no se recomienda
#print(persona.__clave)      # Error: atributo privado
persona.mostrar_clave()      # Accede desde un método
```

### Método getter y setter

En el ejemplo anterior vimos que para acceder al atributo privado tuvimos que usar un metodo que si tenia acceso a esa información. Este tipo de método se conoce como **getter** ya que se utiliza para obtener información de un atributo oculto del objeto. Otro tipo de método conocido es el **setter**, este nos permite modificar o "setear" un atributo privado dentro de nuestro objeto.

Ejemplo:

```
class Persona:
    def __init__(self, nombre):
        self.__nombre = nombre  # atributo privado
    #getter
    def ver_nombre(self):
        return self.__nombre
    #setter
    def modificar_nombre(self,nombre):
        self.__nombre = nombre
persona = Persona("Pedro") # Creo el objeto con el nombre "Pedro"
print(persona.ver_nombre()) # Muestro "Pedro" utilizando el getter
persona.modificar_nombre("Lucas") # Modifico el nombre utilizando el setter
print(persona.ver_nombre()) # Muestro "Lucas" utilizando el getter
```

### Abstracción

La abstracción hace referencia a la ocultación de la complejidad que puede tener una aplicación al exterior, centrándose solo en como puede ser usada. Un ejemplo puede ser cuando utilizamos un cajero automático para retirar dinero. No necesitamos saber como se valida nuestra contraseña, ni como se actualiza el saldo en la cuenta. Solo necesitamos saber que existe una función para retirar dinero y que hace todo lo demás de forma transparente para nosotros.

Veamos un ejemplo:

```
class Cajero:
    def __init__(self, saldo):
        self.__saldo = saldo
    def retirar_dinero(self, monto):
        #Si el saldo del cajero es válido se permite retirar dinero
        if self.__validar_saldo(monto):
            self.__saldo -= monto
            print(f"Retiraste ${monto}")
        else: # Caso contrario no se puede retirar dinero
            print("Saldo insuficiente")
    #Metodo privado
    def __validar_saldo(self, monto):
        return self.__saldo >= monto
cajero = Cajero(10000) # Creo el objeto con 10000 de saldo
cajero.retirar_dinero(8000) # Muestra "Retiraste $8000"
cajero.retirar_dinero(4000) # Muestra "Saldo insuficiente"
```

En el ejemplo se oculta la función que valida el saldo y solo podemos utilizar la función retirar_dinero().

Podemos resumir el principio de abstracción en que es una forma de diseñar objetos donde se oculta lo complejo y se muestra lo esencial, todo esto con el objetivo de simplificar su uso.

## Herencia

La **herencia** es un proceso mediante el cual se puede crear una clase **hija** que hereda de una clase **padre**, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobreescribir los métodos o atributos, o incluso definir unos nuevos.

Conceptos clave:

- **Clase padre**: Contiene métodos y atributos de la clase base.
- **Clase hija:** Hereda métodos y atributos de la clase padre.
- **super():** Función especial que llama métodos de la clase padre desde la hija.

Anteriormente vimos como representar un automóvil como objeto, supongamos que ahora se nos pide representar otro tipo de vehículo, por ejemplo, una motocicleta. Si analizamos las características de cada vehículo podemos ver que tienen algunos atributos en común.

Los atributos podrían ser:

- **Auto**: marca, modelo y puertas.
- **Moto**: marca, modelo y cilindrada.

Lo que podemos notar es que marca y modelo son atributos que coinciden en ambos objetos. Cuando suceden este tipo de similitudes, lo conveniente es diseñar una clase base (Vehículo) que defina los atributos en común (marca y modelo) y luego las clases hijas (Auto y Moto) puedan heredarlos.

A los atributos en común, también podemos incorporar los comportamientos en común (métodos). Por ejemplo, sabemos que tanto un auto como una moto pueden arrancar, acelerar o frenar.

Veamos un ejemplo de como se define una **clase padre**, en este caso la de vehículo:

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

    def arrancar(self):
        print(f"{self.marca} {self.modelo} está arrancando...")
```

La clase padre Vehículo define los atributos de base marca y modelo que luego van a heredar las clases Auto y Moto.

Ahora vamos a definir las clases hijas:

**Auto**:

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

    def encender_aire_acondicionado(self):
        print("Aire acondicionado encendido.")
```

**Moto**:

```
#Clase Hija Moto
class Moto(Vehiculo):
    def __init__(self, marca, modelo, cilindrada):
        super().__init__(marca, modelo)
        self.cilindrada = cilindrada

    def bloquear_manubrio(self):
        print("Se bloqueo el manubrio correctamente.")
```

Podemos notar que tanto la clase Auto como la clase Moto reciben como parámetro de entrada la clase Vehículo, esto es lo que nos va a permitir heredar los atributos y métodos de la clase padre.

Luego la función super() lo que nos permite es acceder a los atributos de la clase padre para poder modificarlos.

Veamos un ejemplo:

```
auto = Auto("Toyota","Corolla","4")
auto.encender_aire_acondicionado() # (Método propio) Muestra "Aire acondicionado
encendido."
auto.arrancar() # (Método Heredado) Muestra "Toyota Corolla está arrancando..."
moto = Moto("Yamaha", "FZ", 250)
moto.bloquear_manubrio() # (Método propio) Muestra "Se bloqueo el manubrio correctamente."
moto.arrancar() # (Método Heredado) Muestra "Yamaha FZ está arrancando..."
```

Creamos un objeto auto y un objeto moto que heredan marca y modelo de la clase Vehículo. Además, utilizamos el método arrancar heredado en ambos objetos, luego invocamos los metodos para encender el aire acondicionado en el auto y bloquear el manubrio en la moto. La pregunta es ¿Qué sucede si intento invocar el método encender_aire_acondicionado() desde el objeto moto? Claramente esto no seria posible ya que ese método solo es accesible por el objeto auto. Lo mismo sucede para el método bloquer_manubrio(), solo es accesible por el objeto moto.