OPTATIVA 2: PROGRAMACIÓN. 2º SMR

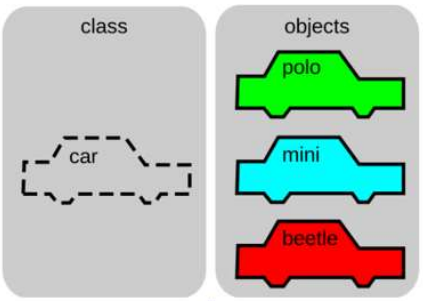
Programación Orientada a Objetos

La POO es un paradigma de programación que se basa en el uso de clases y objetos.

Una clase es una estructura que define las propiedades (atributos) y el comportamiento (métodos) de un objeto.

Un objeto es una instancia (o un caso concreto) de una clase.

Idea: Las clases se pueden considerar los moldes, plantillas o el diseño para crear objetos. Los objetos se crearían basándose en esos *moldes* o *diseños*.



En general, en las clases se definen las propiedades o atributos y los métodos que tendrán los objetos concretos que se creen basándose en una determinada clase.

El estado de un objeto dependerá de sus valores o atributos; el comportamiento de los objetos depende de los métodos definidos:

- Métodos -> funciones.
- Atributos -> variables.

Ejemplos (en seudocódigo):

```
Clase perro{
        Nombre: tipo cadena de caracteres
        Raza: tipo cadena de caracteres
        Edad: tipo número
        Color: tipo cadena de caracteres

        función ladrar(){
                imprimir "Guau"
        }
        función gruñir(){
                imprimir "Grrrr!!!"
        }
}
```

```
Clase coche{
        Marca: tipo cadena de caracteres
        Modelo: tipo cadena de caracteres
        Potencia: tipo número
        Color: tipo cadena de caracteres

        función arrancar(){
                imprimir "Consumiendo combustible"

        }
        función pedir_mantenimiento(){
                        imprimir "Llévame al taller"

        }
}
```

```python
#Ejemplo de POO. Se define la clase perro y se crea un objeto de esta clase.

class perro:
    def __init__(self, nombre, raza, edad, color):
        print("Creado!")
        self.nombre = nombre
        self.raza= raza
        self.edad = edad
        self.color = color

    def ladrar(self):
        print("Guau!")

    def grunir(self):
        print("GRRRRRR!")

Rocky = perro("Rocky", "Bulldog Francés", 11, "Negro")

print(Rocky.nombre)
print(Rocky.raza)
print(Rocky.edad)
print(Rocky.color)

Rocky.ladrar()
Rocky.grunir()
```

```
Creado!
Rocky
Bulldog Francés
11
Negro
Guau!
GRRRRRR!
```

Los cuatro principios de la POO:

La POO se basa en cuatro principios fundamentales:

Encapsulamiento: las variables y los métodos se definen juntos y se controla el acceso a los datos (variables) permitiendo la interacción con las variables solo con los métodos definidos en la clase.

Abstracción: consiste en ocultar los detalles que no tienen que conocerse. Se reduce la complejidad del uso porque el usuario no tiene que conocer los detalles de cómo se hacen las cosas.

Polimorfismo: capacidad de usar un mismo nombre de método para comportamientos diferentes, según el objeto que lo ejecute o el número de parámetros que se usan al llamar al método.

Herencia: mecanismo para que una clase que es hija de otra herede sus métodos y propiedades.

ENCAPSULAMIENTO

El encapsulamiento es un principio de la POO que consiste en ocultar los datos internos de un objeto y permitir el acceso a ellos solo mediante métodos controlados.

Pero en Python el encapsulamiento tiene que crearlo el programador de forma explícita.
Esto se hace definiendo las variables que queremos encapsular usando dos símbolos de subrayado delante del nombre de la variable.

Ejemplo.:

```
class perro:
    def __init__(self, nombre, raza, edad, color):
        print("Creado!")
        self.nombre = nombre
        self.raza= raza
        self.edad = edad
        self.color = color

    def ladrar(self):
        print("Guau!")

    def grunir(self):
        print("GRRRRRR!")

Rocky = perro("Rocky", "Bulldog Francés", 11, "Negro")

print(Rocky.nombre)
print(Rocky.raza)
print(Rocky.edad)
print(Rocky.color)

Rocky.ladrar()
Rocky.grunir()
Rocky.nombre ="Lagartijo"
print(Rocky.nombre)
```

En este caso no hay encapsulamiento porque no se ha hecho (por parte del programador).

A continuación, se ve un ejemplo en el que se han protegido tres atributos (encapsulados) y otro no:

```
class perro:
    def __init__(self, nombre, raza, edad, color):
        print("Creado!")
        self.__nombre = nombre
        self.__raza= raza
        self.edad = edad #ATRIBUTO NO ENCAPSULADO
        self.__color = color

    def ladrar(self):
        print("Guau!")

    def grunir(self):
        print("GRRRRRR!")

Rocky = perro("Rocky", "Bulldog Francés", 11, "Negro")

print(Rocky.nombre)
print(Rocky.raza)
print(Rocky.edad)
print(Rocky.color)

Rocky.ladrar()
Rocky.grunir()
Rocky.nombre ="Lagartijo"
Rocky.edad = 14
print(Rocky.nombre)
print(Rocky.edad)
```

Al intentar ejecutarlo:

```
Creado!
Traceback (most recent call last):
  File "D:\Curso25_26\Opt_2\POO\poo_1_perro.py", line 19, in <module>
    print(Rocky.nombre)
AttributeError: 'perro' object has no attribute 'nombre'
```

Pregunta: ¿qué tendríamos que hacer en el código anterior para que no nos dé error?

¿Cómo podemos modificar los atributos encapsulados?

Se puede hacer de varias formas:

FORMAS DE ACCEDER A UN ATRIBUTO ENCAPSULADO

Forma 1 (NO RECOMENDADA): a través de la clase
objeto._clase__atributo → ¡¡atención a los guiones bajos!!

Ejemplo:

```
print(Rocky._perro__nombre)
```

Forma 2 (RECOMENDADA): usar un método o propiedad de tipo get (obtener) -> *guetizar*

Se define una clase que devuelve el dato buscado:

```
def get_dato(self):
        return self.__dato
```

Y se llama así:

objeto.get_dato()

Ejemplo:

```
def get_nombre(self):
    return self.__nombre

print(Rocky.get_nombre())
```

Forma 3 (RECOMENDADA y más *pythónica*): usando un decorador @property.

De esta forma se crea un *getter* "pythonico" que permite acceder a un método como si fuera un atributo de **solo lectura**.

En la definición de la clase se añade:

```
@property
def dato(self):
        return self.__dato
```

Y se llama así:

print(obj.dato)

Ejemplo:

```
@property
def raza(self):
    return self.__raza
```

Y se llama así:

```
print(Rocky.raza)
```

Conclusiones/resumen rápido: (en todos los casos se supone que el atributo se ha encapsulado correctamente).

print(objeto.__atributo) -> No funciona (si el atributo se encapsulado correctamente).

print(objeto._Clase__atributo) -> Funciona , pero NO se recomienda.

print(objeto.metodo()) o bien: print(objeto.propiedad) -> Funciona y es la forma recomendada.

Resumen de los tipos de atributos en Python:

Forma del atributo	**Ejemplo**	**Accesible desde fuera**	**Significado**
Público	self.nombre	SÍ	Sin restricciones
Protegido	self._nombre	SÍ	No recomendado. UYOR
Privado (encapsulado)	self.__nombre	NO	Uso interno de la clase

ABSTRACCIÓN

La abstracción es el proceso de no tener en cuenta las características no relevantes de algo para quedarnos solo con las características esenciales. En el caso de la POO se usa abstracción cuando se crean clases en las que se omiten los detalles no esenciales de la clase definida. La abstracción permite, además, modificar partes del software sin necesidad de retocar todo un programa. Ejemplo: si definimos la clase persona podríamos tener en cuenta los aspectos esenciales de una persona: edad, sexo, peso, estatura, raza,... pero no, en cambio, otros atributos como, por ejemplo: número de pelos en la cabeza, si tiene las uñas largas o cortas,...

Otro ejemplo de abstracción: cuando se usa la clase *turtle* de Python para hacer gráficos no tenemos que conocer los detalles sobre cómo están programadas las clases de esta librería de software; nos basta con saber usarla (crear objetos, conocer los atributos y la forma de llamar a los métodos).

Algunos ejemplos con código:

Un ejemplo básico:

```
#Ejemplo de abstracción

class Interruptor:
    def __init__(self):
        self.__proveedor = "Iberdrola"
        self.__tipocontrato = "Doméstico"
        self.__tarifa = "Estándar"

    def encender(self):
        print("Luz encendida")

    def apagar(self):
        print("Luz apagada")


i = Interruptor()
i.encender()
i.apagar()
```

```
Luz encendida
Luz apagada
```

Para definir el objeto y llamar a los métodos encender y apagar no tenemos que conocer los detalles sobre el proveedor, tipo de contrato, tarifa,... porque no son necesarios y, por lo tanto, se ocultan al usuario.

A continuación, un ejemplo no tan sencillo en el que se usa una clase abstracta: las clases abstractas son clases que no permiten instanciar, directamente, objetos de ellas; sirven como una plantilla para definir qué métodos existirán en las clases hijas (las que se crean basándose en ellas) pero no cómo se tienen que implementar.

Se importa el módulo abc (Abstract Base Classes) que permite usar ABC: clases base para crear clases abstractas y @abstractmethod: decorador para hacer que un método sea obligatorio.

Como se ve en el código: se crea una clase abstracta MetodoPago, que sirve como una *plantilla* para crear las clases (se consideran subclases o hijas) Tarjeta y Paypal.

```
#Ejemplo de uso de abstracción en Python
from abc import ABC, abstractmethod

class MetodoPago(ABC):
    @abstractmethod
    def pagar(self, cantidad):
        pass

class Tarjeta(MetodoPago):
    def pagar(self, cantidad):
        print(f"Pagando {cantidad} € con tarjeta")

class Paypal(MetodoPago):
    def pagar(self, cantidad):
        print(f"Pagando {cantidad} € con Paypal")

def procesar_pago(metodo: MetodoPago):
    metodo.pagar(50)

procesar_pago(Tarjeta())
procesar_pago(Paypal())
```

El uso de una clase abstracta sirve para forzar a las clases hijas (las que se crean basándose en ella) a implementar ciertos métodos, definiendo comportamientos comunes a las clases hijas pero sin entrar en cómo se implementan estos métodos en las clases hijas.

Se usa el "decorador" *@abstractmethod* para hacer que un método sea obligatorio: tiene que definirse en las clases hijas, pero como se quiera.

En las clases abstractas no se incluye código, pero eso se pone la palabra reservada *pass,* para indicar que en el interior de esa clase no se va a definir nada.

Sin intentáramos instanciar un objeto de una clase abstracta pasaría esto:

```python
#Ejemplo de uso de abstracción en Python
from abc import ABC, abstractmethod

class MetodoPago(ABC):
    @abstractmethod
    def pagar(self, cantidad):
        pass

class Tarjeta(MetodoPago):
    def pagar(self, cantidad):
        print(f"Pagando {cantidad} € con tarjeta")

class Paypal(MetodoPago):
    def pagar(self, cantidad):
        print(f"Pagando {cantidad} € con Paypal")

def procesar_pago(metodo: MetodoPago):
    metodo.pagar(50)

procesar_pago(Tarjeta())
procesar_pago(Paypal())
pago = MetodoPago()
```

```
Pagando 50 € con tarjeta
Pagando 50 € con Paypal
Traceback (most recent call last):
  File "C:\Python\POO\poo_abstraccion.py", line 22, in <module>
    pago = MetodoPago()
TypeError: Can't instantiate abstract class MetodoPago without an implementation
 for abstract method 'pagar'
```

POLIMORFISMO

El polimorfismo es la característica, en programación, de usar un mismo interfaz (por ejemplo: mismo nombre en distintos métodos) para codificar en esos métodos distintos comportamientos (diferente código).

El polimorfismo permite tener un código más flexible, más fácil de mantener; permite extender el programa sin modificar el código que ya existe; además mejora la legibilidad y la reutilización.

Existen dos tipos de polimorfismo:

Polimorfismo en tiempo de ejecución (en Python): un mismo método puede tener distinto comportamientos, según el tipo del objeto.

Polimorfismo en tiempo de compilación (no en Python): el compilador decide qué método usar, según el número, orden y tipo de parámetros.

Un ejemplo muy básico en Python es la función print() que sirve para imprimir cadenas de caracteres, número enteros, números con decimales o, incluso, una tupla o una lista enteras,

Otro ejemplo:

```
#Ejemplo de polimorfismo sin herencia:
class Perro:
    def sonido(self):
        print("GUAU!")

class Gato:
    def sonido(self):
        print("MIAU!")

class Robot:
    def sonido(self):
        print("01010101010")

def habla(objeto):
    objeto.sonido()

#Se define una lista con objetos, uno de cada clase.
mascotas = [Perro(), Gato(), Robot()]
for m in mascotas:
    habla(m)
```

```
GUAU!
MIAU!
01010101010
```

En el código anterior se definen los objetos dentro de una lista; se haría referencia a cada objeto usando el nombre de la lista y el índice:

```
habla(mascotas[0]) #Perro
habla(mascotas[1]) #Gato
habla(mascotas[2]) #Robot
```

```
GUAU!
MIAU!
01010101010
```