OPTATIVA 2: PROGRAMACIÓN. 2º SMR

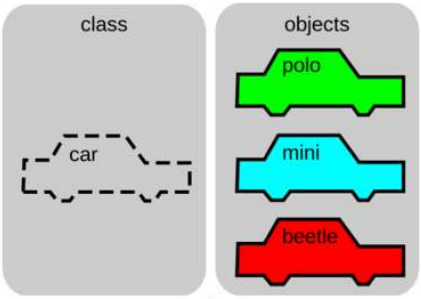
Programación Orientada a Objetos

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. En general tiene que aceptar, al menos, un parámetro: `método(self)`:
  El método __init__ sirve para inicializar el objeto; es opcional pero hay que usarlo si queremos que, al crear el objeto, éste tenga unos valores iniciales.
- 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
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"
```

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

En el ejemplo anterior hay tres métodos:

`__init__(self, nombre, raza, edad, color): -> cuatro parámetros.`

`ladrar(self) -> un único parámetro; obligatorio.`

`grunir(self) -> un único parámetro; obligatorio.`

El parámetro *self* es una forma de representar la propia instancia (el objeto) sobre la que se está llamando al método.

Cuando hacemos una llamada a un método así: `Clase. Método(arg1, arg2)` Python lo traduce internamente por: `Clase. Método(obj, arg1, arg2)` -> se pasa el objeto (o la instancia) como primer argumento, por eso se declara como primer parámetro *self*, que recibirá este primer parámetro (el objeto).

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.

En Python, el encapsulamiento se implementa por **convención** (tiene que hacerlo el programador), usando atributos privados (__atributo) y métodos públicos para controlar el acceso.

Otro detalles a tener en cuenta: en Python existe la convención de nombrar las clases con mayúscula inicial; si el nombre estuviera compuesto por varias palabras cada una se empieza con mayúscula.
Ej.: `class CuentaCorriente:`

A continuación se muestra un ejemplo sin encapsulamiento (no es una buena práctica de programación):

```
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"
```

En este caso no hay encapsulamiento porque el programador no lo ha implementado.

A continuación, se ve un ejemplo en el que se han protegido tres atributos (encapsulados con __) 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._Perro__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 en la clase un método get 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* "pythónico" 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 ha encapsulado correctamente).

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

print(objeto. Método()) 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[1]
Privado (encapsulado)	self.__nombre	NO	Uso interno de la clase

Práctica guiada: clase rectángulo.

[1] Under Your Own Risk -> tú mismo (traducción aproximada)