Programación Orientada a Objetos

con Python...

Python es un lenguaje de programación multiparadigma, es compatible con diferentes enfoques de programación.

Uno de los enfoques para resolver un problema de programación es mediante la creación de objetos. Esto se conoce como Programación Orientada a Objetos. (POO)

TODO EN PYTHON ES UN OBJETO.

Un objeto tiene dos propiedades:

ATRIBUTOS → CARACTERISTICAS y MÉTODOS (*) → COMPORTAMIENTO

(*) un método es una función que está definida dentro de una clase.

Por ejemplo, un perro puede ser un objeto. ya que tiene las siguientes propiedades:

nombre, edad, color → atributos (características)

ladrar, caminar → métodos (comportamiento)

Clases

La clase es un 'molde' que sirve para crear un objeto. Podemos pensar en la clase como un boceto (prototipo) de una casa. Contiene todos los detalles sobre pisos, puertas, ventanas, etc. Basándonos en estas descripciones construimos la casa. La casa es el objeto.

```
class Perro:
    pass
```

pass es una operación nula, cuando es ejecutada, nada sucede.

Usamos class para definir una clase.

Objetos

Un objeto (también llamado instancia) es creado a partir de una clase (el molde, boceto...). El proceso de creación de este objeto se denomina instanciación.

creación de clases y objetos

```
class Perro:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} de {self.age} años"

# Instancio la clase = creo el objeto
perro_1 = Perro("Wolf", 7)
print(perro_1)
```

→ Wolf de 7 años

- El constructor **__init__** es aquel método que sirve para inicializar algunos atributos (características) de un objeto. Se ejecuta justo después de crear un objeto a partir de una clase.
- Parámetro **self**: Primer parámetro de **__init__**. 'Self' sirve para trabajar con un objeto futuro dentro de una clase (que será creado posteriormente en el código) permitiendo agregar y leer atributos y métodos desde la definición misma de la clase.

★ método __str__(self) : lo llamamos para crear una cadena de texto que represente a nuestro objeto.

Métodos ...

Los métodos son funciones definidas dentro del cuerpo de una clase. Se utilizan para definir los comportamientos de un objeto.

Crear métodos

```
class Perro:
  def __init__(self,name,age):
    self.name = name
    self.age = age

  #MÉTODO

  def ladrar(self):
    print(f"{self.name} ladra fuerte")

perro_1 = Perro("Wolf", 7)
perro_1.ladrar()
```

output:

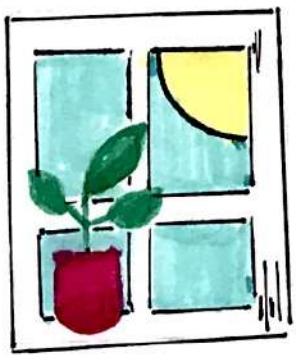
→ Wolf ladra fuerte

Herencia

La herencia es el fenómeno que se da cuando una clase superior, por ejemplo "Profesional", le comparte sus atributos y métodos a una clase derivada, por ejemplo "médico", sin necesidad de que esta última clase defina todos los atributos y métodos nuevamente.

```
class BaseClass:
    Cuerpo de la clase BaseClass
class DerivedClass(BaseClass):
    Cuerpo de la clase derivada
```

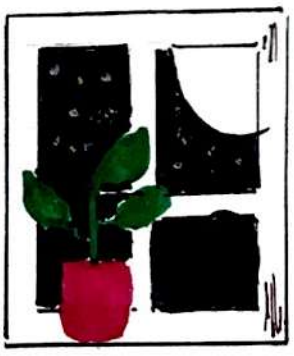
La clase derivada hereda características de la clase base donde se pueden agregar nuevas características. Esto da como resultado la reutilización del código.



```
class cuadrilatero:
    def __init__(self, lados):
        self.lados = lados
        self.suma_angulos = 360
    def perimetro(self):
        return sum(self.lados)

class cuadrado(cuadrilatero):
    def __init__(self, lados):
        super().__init__(lados)

cuadrado_1 = cuadrado([4,4,4,4])
perimetro_1 = cuadrado_1.perimetro()
print(perimetro_1)
print(cuadrado_1.suma_angulos)
```



super() es una función que te permite acceder a los atributos y métodos de la clase base.

En el ejemplo, la función super() llama al constructor de cuadrilátero. De esta manera, cuadrado pasa a heredar todos los atributos y métodos de cuadrilátero.

herencia múltiple

Una clase puede derivarse de más de una clase base en Python.

En la herencia múltiple, las características de todas las clases base se heredan en la clase derivada.

```
class Base1:
  pass

class Base2:
  pass

class MultiDerivada(Base1, Base2):
  pass
```

También podemos heredar de una clase derivada. Esto se llama **HERENCIA MULTINIVEL**.

```
class Base:
  pass

class Derivada2(Base):
  pass

class Derivada3(Derivada2):
  pass
```

Encapsulamiento

~encapsulamiento~

Usando POO, podemos restringir el acceso a métodos y variables. Esto evita que los datos se modifiquen directamente, lo que se denomina **ENCAPSULACIÓN** ó **ENCAPSULAMIENTO**.

Existen dos maneras de crear atributos privados en Python: con un guión bajo al principio del nombre del mismo, o con dos guiones bajos (__) La primera forma establece que los atributos son privados pero solo por convención (internamente, no son realmente privados). La segunda forma hace que los atributos adquieran las propiedades que realmente los hacen privados.

```
class Computer:
    def __init__(self):
        self.__maxprice = 900

    def sell(self)
        print(F"El precio de venta es: {self.__maxprice}")

    def set_maxprice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()                    // El precio de venta es: 900

#cambio el precio
c.__maxprice = 1000         //El precio de venta es: 900
c.sell()
```

```
#usamos setter function

c.set_maxprice(1000)

c.sell()  // El precio de venta es: 1000
```

Existen dos tipos de métodos que permiten lograr el encapsulamiento: los getters y setters. Los getters son métodos que permiten ver el valor de una variable privada, mientras que los setters permiten modificarla.

abstracción

La abstracción es un concepto de la Programación Orientada a objetos que explica que tanto detalle se espera en un objeto a la hora de definir métodos y atributos. Por ejemplo, a la hora de crear un objeto de tipo auto puedo tener dos niveles de abstracción, uno en el que se defina métodos que establezcan el funcionamiento del motor y la caja de cambios, o uno en el que solo defina los atributos y métodos necesarios para conducir.

Depende de quien vaya a usar mi objeto, el nivel de abstracción que elija.

POLIMORFISMO

El **polimorfismo** es la característica de los objetos que permite implementar un método varias veces con diferente funcionalidad cada vez.

```
class Animal:
  def hablar(self):
    pass

class Perro(Animal):
  def hablar(self):
    print("guau")

class Gato(Animal):
  def hablar(self):
    print("Miau")

class Vaca(Animal):
  def Hablar(self):
    print("Muu")
```

```
animales = [Perro(), Gato(), Vaca()]
for animal in animales:
    animal.hablar()

# guau
# Miau
# Muu
```

Tenemos una clase padre **Animal** con un método definido pero no implementado, de la que heredan 3 animales. Cada animal implementa el método común de una manera diferente.