Programación Orientada a Objetos

La Programación Orientada a Objetos es un paradigma de programación que utiliza "objetos" y sus interacciones para diseñar aplicaciones y programas. Los objetos son instancias de clases, que pueden contener datos (atributos) y funciones (métodos). La POO facilita la reutilización del código, la modularidad y el mantenimiento de los programas.

Porque es importante la Programación Orientada a Objetos

- Reutilización del Código: Permite usar el mismo código en diferentes partes del programa o en diferentes proyectos.
- Modularidad: Divide el programa en partes pequeñas y manejables (clases y objetos).
- Mantenimiento y Flexibilidad: Facilita la actualización y mejora del código sin afectar a otras partes del programa.
- Abstracción: Simplifica la complejidad al modelar problemas del mundo real en términos de objetos.
- Encapsulamiento: Protege los datos dentro de los objetos, permitiendo acceso controlado a través de métodos.

Definiendo las clases

Una clase es una plantilla o blueprint para crear objetos. Define un conjunto de atributos (datos) y métodos (funciones) que los objetos de esa clase tendrán.

sintaxis básica:

```
 class mi_clase:
     miembros (atributos, constructores, métodos)
```

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
print(Persona)
print(type(Persona))
```

```
<class '__main__.Persona'>
<class 'type'>
```

Modificadores de Acceso

Los modificadores de acceso son una característica importante en la programación orientada a objetos (POO) que controlan la visibilidad y el alcance de las variables y métodos dentro de las clases. En otras palabras, determinan quién puede ver y utilizar ciertos elementos de una

clase.

En Python, los modificadores de acceso no son tan estrictos como en otros lenguajes como Java o C++, pero Python sigue ciertas convenciones para indicar el nivel de acceso.

Tipos de Modificadores de Acceso en Python

Público: (+)Los miembros de la clase que son accesibles desde cualquier lugar. Por defecto, todos los atributos y métodos de una clase en Python son públicos.

Protegido: (#)Los miembros de la clase que son accesibles dentro de la clase y sus subclases. Python no tiene un modificador explícito para esto, pero por convención se usa un guion bajo (_) al principio del nombre.

Privado: (-)Los miembros de la clase que son accesibles solo dentro de la misma clase. Python no tiene modificadores de acceso privados en el sentido estricto, pero por convención se usa un doble guion bajo (__) al principio del nombre.

```
nombre     #atributo público
_nombre    #atributo protegido
__valor   #atributo privado
```

Público

Sintaxis: No tiene ningún prefijo.

Accesibilidad: Accesible desde cualquier lugar (dentro de la clase, fuera de la clase, y desde subclases).

```
class MiClase:
    def __init__(self):
        self.atributo_publico = "Soy público"

    def metodo_publico(self):
        print("Método público")

obj = MiClase()
print(obj.atributo_publico)  # Acceso público
obj.atributo_publico = 'Sigo siendo publico y pueden cambiar mi valor'
print(obj.atributo_publico)  # Acceso público
obj.metodo_publico()  # Llamada pública
```

```
Soy público
Sigo siendo publico y pueden cambiar mi valor
Método público
```

Protegido (#)

Sintaxis: Un guion bajo (_) antes del nombre del atributo o método.

Accesibilidad: Accesible dentro de la clase y sus subclases, pero no se debería acceder a ellos directamente desde fuera de la clase.

```
class MiClase:
    def __init__(self):
        self._atributo_protegido = "Soy protegido"

    def _metodo_protegido(self):
        print("Método protegido")
        print(self._atributo_protegido)

class SubClase(MiClase):
    def mostrar(self):
        print(self._atributo_protegido)  # Acceso protegido
        self._metodo_protegido()  # Llamada protegida

obj = SubClase()
obj.mostrar()
obj._atributo_protegido = "Me cambiaron?"
obj.mostrar()
obj1 = MiClase()
obj1._atributo_protegido ='volvi a cambiar'
obj1._metodo_protegido()
```

```
Soy protegido
Método protegido
Soy protegido
Me cambiaron?
Método protegido
Me cambiaron?
Método protegido
volvi a cambiar
```

Privado

Sintaxis: Doble guion bajo (__) antes del nombre del atributo o método.

Accesibilidad: Accesible solo dentro de la clase donde se define. Python aplica una técnica llamada "name mangling" para modificar el nombre del atributo/método, haciéndolo difícil de acceder desde fuera de la clase.

```
class MiClase:
    def __init__(self):
        self.__atributo_privado = "Soy privado"

    def __metodo_privado(self):
        print("Método privado")

    def mostrar_privado(self):
        print(self.__atributo_privado)  # Acceso privado dentro de la clase
        self.__metodo_privado()  # Llamada privada dentro de la clase

obj = MiClase()
obj.mostrar_privado()

# Acceso desde fuera de la clase causará error
print(obj.__atributo_privado)  # AttributeError
obj.__metodo_privado()  # AttributeError
```

```
Soy privado
Método privado
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-627d22ff8c05> in <cell line: 16>()
     14
     15 # Acceso desde fuera de la clase causará error
---> 16 print(obj.__atributo_privado)  # AttributeError
     17 obj.__metodo_privado()  # AttributeError

AttributeError: 'MiClase' object has no attribute '__atributo_privado'
```

Próximos pasos: Explicar error

El parámetro self

Cada clase da origen a diferentes objetos. El parámetro **self** que se recibe en los constructores o métodos de una clase hace referencia al objeto en particular que está accediendo al comportamiento definido en la clase en un momento determinado.

```
 obtener_nombre(self):
     return self.nombre      #se devuelve el nombre del objeto actual
```

```
#Acceder a los Atributos de la Instancia:
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre}")

p = Persona("Juan")
p.saludar()  # Salida: Hola, mi nombre es Juan
```

```
Hola, mi nombre es Juan
```

Todos los métodos en Python deben recibir como primer parámetro el parámetro **self** de esta manera se tiene una referencia al objeto actual que está haciendo uso de los miembros de la clase.

```
#Llamar a Otros Métodos de la Clase:
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre}")

    def presentarse(self):
        self.saludar()  # Llama al método saludar

p = Persona("Juan")
p.presentarse()  # Salida: Hola, mi nombre es Juan
```

```
class Persona:
  def saludar(self, nombre):
    print("hola ", nombre)
    print(self)
##############################################
p = Persona()
print(p)
print(type(p))
p.saludar ("Juan")
```

```
p1 = Persona()
p1.saludar("Lina")
```

El constructor

Si en Python no especificamos un constructor, de igual manera cada clase tendrá un constructor por defecto que nos permite instanciar objetos de esa clase. Miremos:

```
class Persona:
  pass

juan = Persona()
print (juan)
print ("memoria juan:", juan) #muestra la posición de memoria
print ("tipo juan:", type(juan)) #muestra de qué tipo es juan
```

```
<__main__.Persona object at 0x7d992ff96cb0>
memoria juan: <__main__.Persona object at 0x7d992ff96cb0>
tipo juan: <class '__main__.Persona'>
```

En el código anterior, se creó un objeto de la clase Persona, el objeto se llama **juan** y no hace nada, ya que la clase Persona no tiene código. Pero el objeto existe y tiene memoria asignada para su almacenamiento.

__ init__ ()

En Python, el constructor es un método especial llamado **init**. Es el primer método que se ejecuta cuando se crea una instancia de una clase. Se utiliza para inicializar los atributos de la instancia.

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

p = Persona("Juan", 30)
print(p)
print(p.nombre)
print(p.edad)
```

```
class Persona:
  def __init__(self, n, e, c):
    self.nombre = n
    self.edad = e
    self.ciudad = c

p = Persona("juan", 23, "Cali")
print (p.ciudad)
```

```
Cali
```

Incluso se puede usar para realizar cualquier configuración inicial necesaria para el objeto, como en este caso que el constructor **init** se utiliza para inicializar el estado del objeto(abrir el archivo), mientras que los métodos 'escribir' y 'cerrar' permiten realizar operaciones especificas en el archivo de manera controlada y organizada.

```
class Archivo:
    def __init__(self, nombre):
        self.archivo = open(nombre, 'w')

    def escribir(self, texto):
        self.archivo.write(texto)

    def cerrar(self):
        self.archivo.close()

a = Archivo("mi_archivo.txt")
a.escribir("Hola Mundo ADSO 8!")
a.cerrar()
```

En Python no tenemos que declarar explícitamente los atributos de una clase. Ellos van a estar siempre almacenados en el parámetro self. Miremos:

```
class Persona:
  def __init__(self, nombre, edad, ciudad):
    self.nombre = nombre
    self.edad = edad
    self.ciudad = ciudad

  def mostrar_informacion(self):  #mediante el self el método reconoce al objeto que está invocando el método
    print("El nombre es ", self.nombre, self.edad, self.ciudad)
##############################################

juan = Persona("Juan Villa", 23, "Manizales")
pedro = Persona("Pedro Valencia", 40, "Pereira")

juan.mostrar_informacion()
pedro.mostrar_informacion()
```

En Python se pueden crear atributos "al vuelo", es decir, después de crear un objeto se le pueden asignar los atributos deseados y estos formarán parte de los miembros de este objeto, los cuales recordemos podemos acceder a través del parámetro **self** Veamos:

```
class Persona:

  def mostrar_informacion(self):
    print(vars(self))      # devuelve un diccionario con los atributos y valores del objeto actual
##############################################

juan = Persona()
juan.altura = 180
juan.peso = 85

juan.mostrar_informacion()

obj = Persona()
obj.sexo = "masculino"
obj.salario = 1000000
obj.tipo_sangre = "o+"
obj.mostrar_informacion()


#juan.mostrar_informacion()
juan.salario = 1200000
juan.mostrar_informacion()
```

```
{'altura': 180, 'peso': 85}
{'sexo': 'masculino', 'salario': 1000000, 'tipo_sangre': 'o+'}
{'altura': 180, 'peso': 85, 'salario': 1200000}
```

Atributos de Instancia y de Clase

DE INSTANCIA

Los atributos de los ejemplos anteriores son atributos de instancia, esto quiere decir que cada instancia (objeto) tiene su propio conjunto de atributos con sus propios valores. Estos atributos son accedidos a través del parámetro **self** y el cambio de valor en un atributo de instancia no tiene efecto en los demás objetos. Incluso, gracias a los atributos dinámicos en Python, dos instancias de la misma clase pueden tener atributos diferentes con valores por supuesto diferentes. Miremos:

```
class Estudiante:
  def __init__(self, nom, nota):
    self.nombre = nom
    self.nota = nota

e1 = Estudiante ("Gustavo", 5)
e2 = Estudiante ("Galo", 4.8)

print (e1.nota)
print (e2.nota)

e2.nota = 3

print (e1.nota)
print (e2.nota)
```

DE CLASE

Un atributo de clase por su parte, es un atributo cuyo valor es el mismo para todas las instancias de una clase pues pertenece a la clase y no a sus instancias, aunque estas lo puedan acceder.

```
class Persona:
    especie = "Humano"  # Atributo de clase

    def __init__(self, nombre, edad):
        self.nombre = nombre  # Atributo de instancia
        self.edad = edad      # Atributo de instancia

p1 = Persona("Juan", 30)
p2 = Persona("Ana", 25)

print(p1.especie)  # Salida: Humano
print(p2.especie)  # Salida: Humano

# Cambiar el atributo de clase
Persona.especie = "Homo sapiens"
print(p1.especie)  # Salida: Homo sapiens
print(p2.especie)  # Salida: Homo sapiens


class Persona:
  personas_total = 0 #este es un atributo de clase
  def _init_(self):
    Persona.personas_total += 1  #cada vez que se crea un objeto se incrementa el atributo de clase
################################################

juan = Persona()
print ("personas_total a través de juan: ", juan.personas_total)


pedro = Persona()
print ("personas_total a través de pedro: ", pedro.personas_total)
print ("personas_total a través de juan: ", juan.personas_total)


luis = Persona()
print ("personas_total a través de luis: ", luis.personas_total)
print ("personas_total a través de juan: ", juan.personas_total)
print ("personas_total a través de pedro: ", pedro.personas_total)

Persona.personas_total = 50

print ("juan ->", juan.personas_total)
print ("pedro ->", pedro.personas_total)
```

Métodos

Los métodos son funciones definidas dentro de una clase, que describen los comportamientos de los objetos de esa clase. Se definen igual que una función con la diferencia de siempre recibir el primer parámetro **self** que como vimos antes hace referencia al objeto actual.

Sintaxis:

```
 def mi_metodo(self):
     cuerpo del método


 def mi_metodo(self, parametro1, parametro2, parametro_n):
     cuerpo del método
```

```
class Persona:

  def __init__(self, n, e, c):
    self.nombre = n
    self.edad = e
    self.ciudad = c

  def cumplir_anios(self):
    self.edad += 1

  def get_edad(self):
    return self.edad

  def set_edad(self, x):
    self.edad = x

  def puede_votar(self):
    if self.edad >= 18:
      return True
    else:
      return False

  def cambiar_ciudad(self, nueva_ciudad):
    self.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {self.ciudad}. Podrá votar??? {self.puede_votar()}")
##############################################################################################
p = Persona("Raul Diaz", 17, "Manizales")
p.mostrar_info()
p.cumplir_anios()
p.cambiar_ciudad("Cali")
p.mostrar_info()
```

Métodos de Clase

De la misma manera que existen los atributos de clase, también existen los métodos de clase. Un método de clase es un método que tiene acceso a la clase (atributos de clase u otros métodos de clase).

Para declarar un método como método de clase debemos poner el decorador **@classmethod** antes de la declaración del método de clase.

Estos métodos en lugar de recibir como primer parámetro self, reciben como primer parámetro **cls** que representa a la clase y a través de este parámetro puede acceder a los miembros de la clase.

```
class Persona:
  ciudad = "Manizales"  #atributo de clase

  def __init__(self, nombre, edad):
    self.nombre = nombre
    self.edad = edad

  def cumplir_anios(self):
    self.edad += 1

  @classmethod    #este es un método de clase
  def cambiar_ciudad(cls, nueva_ciudad):
    cls.ciudad = nueva_ciudad

  def mostrar_info(self):
    print (f"{self.nombre} tiene {self.edad} años y vive en {Persona.ciudad} ")
#############################################################################

p1 = Persona("Hugo", 35)
p2 = Persona("Paco", 20)
p3 = Persona("Luis", 29)

p1.cumplir_anios()
p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()

p1.cambiar_ciudad("Cali")  #la invocación de un método de clase se realiza a través de la clase

p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()

Persona.cambiar_ciudad("Popayan")
p1.mostrar_info()
p2.mostrar_info()
p3.mostrar_info()
```

Métodos mágicos

Cuando un método tiene dos guiones bajos al principio y final de sus nombres se denomina un método mágico. Los métodos mágicos no se pueden llamar directamente desde el código como lo hacemos con los demás métodos, son "mágicos" porque Python los llama automáticamente en determinadas situaciones.

```
 __metodo_magico__()   #el doble guión bajo antes y después del nombre del método lo identifica como método mágico
```

__ str__ ()

El método mágico **str** en una clase tiene el comportamiento por defecto de retornar una cadena con la clase a la que pertenece el objeto y su dirección de memoria. Este método mágico es llamado cada vez que a la función **str()** le enviamos por parámetro un objeto. Miremos:

En toda clase podemos sobreescribir el método mágico **str** para obtener otro tipo de resultado al imprimir un objeto de dicha clase

```
class Instructor:
  def __str__(self):
    return f"Soy el instructor {self.nombre} "
#########################################################
i1 = Instructor()
i1.nombre = "Carlos"
print(i1)



class Lenguaje:
  def __init__(self, nombre):
    self.nombre = nombre
    print ("se ha creado un objeto")

  def __str__(self):
    return self.nombre
```

```
##########
l1 = Lenguaje("Java")
print(l1)


class Persona:
  def __init__(self, nombre, edad, direccion):
    self.nombre = nombre
    self.edad = edad
    self.direccion = direccion

  def __str__(self):
    return self.nombre + " tiene " + str(self.edad)

#############################################

objeto1 = Persona("Pedro Fabian Rengifo", 23, "calle 3")
print (objeto1)
objeto2 = Persona("Ximena Diaz", 35, "calle 66")
print (objeto2)


class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    # def __repr__(self):
    #     return f"Persona('{self.nombre}', {self.edad})"

p = Persona("Juan", 30)
print(p)
print(repr(p))  # Salida: Persona('Juan', 30)
```

Métodos Estáticos

Los métodos estáticos no acceden ni modifican el estado de los objetos (self) ni de las clases (cls). Son métodos que cumplen una función particular y que quedan asociados a una determinada clase.

Para declarar un método como método estático debemos poner el decorador **@staticmethod** antes de la declaración del método estático.

Estos métodos no reciben ni el parámetro self ni el parámetro cls ya que como lo dijimos antes no conocen ni modifican el estado de la clase ni de sus objetos.

```
import random
class Persona:
```