## Python Object Oriented Programming (OOP)

Object-Oriented Programming (OOP) es muy útil porque nos permite escribir un bloque de código y reutilizarlo creando diferentes clases e instancias de esas clases. Cada una de estas clases puede agregar nuevas funcionalidades sin necesidad de reescribir, cambiar o modificar el código existente. Los programas complejos requieren el uso de clases y objetos porque podemos dividir el código en múltiples partes manejables, lo que facilita su organización, mantenimiento y expansión a lo largo del tiempo.

### *Los principios de la Programación Orientada a Objetos*

Los siguientes principios son generalmente aplicables a todos los lenguajes de programación orientados a objetos:

- **Abstracción**: Permite representar ideas complejas de forma simple mediante clases, ocultando los detalles internos y mostrando solo lo esencial.
- **Encapsulamiento**: Consiste en agrupar datos y métodos que operan sobre ellos en una sola unidad (la clase) y restringir el acceso directo a algunos de los componentes.
- **Herencia**: Permite que una clase hereda atributos y métodos de otra, fomentando la reutilización del código.
- **Polimorfismo**: Permite que objetos de distintas clases respondan al mismo método de diferentes formas.

### *¿Qué otros lenguajes soportan la Programación Orientada a Objetos?*

El lenguaje orientado a objetos más antiguo y conocido es Java. Luego está C#, otro lenguaje OOP desarrollado por Microsoft. Algunos de los lenguajes OOP más famosos son PHP, Ruby, TypeScript y Python.

### *¿Cómo podemos crear clases en Python?*

Para crear una clase en Python, usamos la palabra clave class, seguida del nombre de la clase. Por ejemplo:

---

```
class MiClase:
    pass
```

---

Esta estructura nos permite definir atributos y métodos que darán forma al comportamiento y características de los objetos que creemos a partir de ella.

Como puedes ver, el nombre de la clase comienza con letra mayúscula, y esta es una convención de nombres que deberías adoptar. Otra regla importante es que el nombre de la clase debe ser singular.

Si el nombre de la clase necesita estar compuesto por dos palabras, debes utilizar la convención PascalCase, donde cada palabra empieza con mayúscula y no se utilizan guiones bajos. Por ejemplo:

---

```
class UnEstudiante:
    pass
```

---

Esto mejora la legibilidad del código y sigue las buenas prácticas de programación en Python.

### *¿Qué son las Clases y los Objetos en Python?*

En Python, las clases son como un plano o plantilla que usamos para crear objetos. Dentro de una clase, definimos una serie de propiedades (atributos) y métodos (funciones) que van a representar las características y comportamientos comunes de todos los objetos que creemos a partir de esa clase.

Por otro lado, los objetos (también llamados instancias) son ejemplares concretos creados a partir de esa clase. Cada objeto tiene su propia copia de las propiedades y puede usar los métodos definidos en la clase.

Palabras clave importantes:

- **Clase (Class)**: Es la estructura que define qué atributos y métodos va a tener el objeto.
- **Objeto (Object/Instance):** Es una instancia real creada a partir de una clase.
- **Propiedad (Property):** Es una variable que representa una característica del objeto (por ejemplo, nombre, color, edad).
- **Método (Method):** Es una función definida dentro de una clase que define acciones o comportamientos que el objeto puede realizar.

---

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

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

persona1 = Persona("Ana", 30) # Crear un objeto (instancia) de la clase Persona

persona1.saludar() # Usar un método
```

---

### *Construir clases, atributos y métodos*

En Python, los métodos de clase que comienzan y terminan con doble guion bajo (por ejemplo, __init__) se conocen como métodos especiales, dunder methods (por “double underscore”) o métodos mágicos.

Estos métodos tienen un propósito especial dentro de las clases. Uno de los más comunes es __init__, también llamado constructor, que se ejecuta automáticamente cuando se crea un nuevo objeto de esa clase.

Aunque los métodos usan la misma sintaxis que las funciones (es decir, comienzan con la palabra clave def), la diferencia clave es que:

- Las funciones existen en el ámbito global (fuera de una clase).
- Los métodos están definidos dentro de una clase y son usados por los objetos que se crean a partir de esa clase.

Por eso, en la documentación o libros, muchas veces se habla de funciones dentro de clases como si fueran intercambiables, pero técnicamente, si están dentro de una clase, se consideran métodos.

En resumen:

- def: se usa tanto para definir funciones como métodos.
- __init__: es un método especial llamado automáticamente al crear un objeto.
- __nombre__: cualquier método con doble guion bajo es un método especial (como __str__, __len__, etc.).

Un atributo de objeto de clase (o atributo de clase) es distinto de los demás atributos definidos dentro del método __init__ porque no pertenece a una instancia específica, sino a la clase en sí. Es decir, es compartido por todos los objetos creados a partir de esa clase.

---

```
class Persona:
    is_person = True  # Atributo de clase

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

    def saludar(self):
        print(f"Hola, me llamo {self.nombre} {self.apellido} y tengo {self.edad} años.")
```

---

En este caso:

- is_person es un atributo de clase: se define fuera de __init__ y no depende de un objeto particular.
- Se puede acceder a él desde cualquier instancia (por ejemplo, persona1.is_person) o directamente desde la clase (Persona.is_person).
- Los atributos como nombre, apellido y edad son atributos de instancia, porque se asignan en __init__ con self.

---

```
persona1 = Persona("Ana", "López", 28)
print(persona1.is_person)     # True
```

---

La palabra clave self se usa en los métodos para referirse a la instancia que invoca el método, y así acceder a sus atributos individuales. En cambio, los atributos estáticos como is_person no necesitan self para ser llamados desde la clase.

Los atributos de clase en Python son variables que se definen directamente dentro de la clase y fuera de cualquier método, por lo que son compartidos por todas las instancias u objetos que se creen a partir de esa clase. Se consideran estáticos, ya que su valor no cambia entre objetos, a menos que se modifique desde la clase misma. En cambio, los atributos de instancia se definen dentro del método __init__ usando self, y cada objeto creado tendrá su propia copia de estos atributos con valores individuales. Esto permite que una clase contenga tanto información general (común a todos los objetos) como información específica de cada instancia.

## *El método __init__()*

El método __init__ en Python es conocido como el constructor de una clase y se ejecuta automáticamente cada vez que se crea una nueva instancia de dicha clase. Su función principal es inicializar los atributos del objeto, es decir, asignar los valores iniciales que definen su estado. Se utiliza la palabra clave self para referirse a la propia instancia que se está creando. Aunque ya se explicó su uso básico, es importante destacar que el __init__ puede aceptar tantos parámetros como se necesiten, lo que permite personalizar cada objeto desde su creación. Además, se pueden incluir dentro del constructor validaciones, condiciones o incluso llamadas a otros métodos si se requiere una configuración más compleja del objeto al momento de su instalación.

El constructor __init__ cumple con la función de crear una nueva instancia u objeto a partir de una clase, como por ejemplo la clase Person. Este nuevo objeto podrá usar los atributos y métodos definidos en la clase. El primer parámetro del constructor es siempre la palabra clave self, que actúa como una referencia al propio objeto que se está creando. Aunque self parece un parámetro más, es especial porque permite acceder y diferenciar los datos de cada

instancia. Cada objeto que creamos con la clase tendrá su propio self, lo que garantiza que cada instancia sea única y maneje su propia información de manera independiente. Al principio puede parecer confuso, pero self es esencial para entender cómo funcionan los objetos en Python.

### *Crear métodos de clase utilizando @classmethod*

Para crear métodos de clase en Python que pertenezcan directamente a la clase y no a una instancia específica, usamos el decorador @classmethod. Este decorador se coloca justo antes de la definición del método, y dentro del método usamos el parámetro cls (abreviatura de class) en lugar de self. El parámetro cls hace referencia a la clase y permite acceder o modificar atributos de clase.

---

```
class Persona:
    especie = "Humano"

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

    @classmethod
    def cambiar_especie(cls, nueva_especie):
        cls.especie = nueva_especie
```

---

cambiar_especie es un método de clase que permite modificar el atributo especie para todas las instancias de la clase Persona, porque especie es un atributo estático (de clase). Al usar @classmethod, este método puede ser llamado así:

---

```
Persona.cambiar_especie("Alienígena")
```

---

Para resumir lo que hemos aprendido sobre los métodos de clase en Python:

- El método de clase está vinculado a la clase y no a una instancia (objeto) específica.
- Utiliza el parámetro cls, que representa la clase misma, y no self, que representa una instancia.
- Tiene acceso a todos los atributos de clase (también llamados atributos estáticos), no a los atributos individuales de cada objeto.
- Puede acceder y modificar el estado de la clase, lo cual es útil para mantener un valor compartido entre todas las instancias.

### *Metodo estatico @staticmethod*

El método estático en Python se define usando el decorador @staticmethod y es una función que pertenece a la clase, pero no necesita acceso ni a la instancia (self) ni a la clase (cls). Es decir:

- No accede ni modifica atributos de instancia ni de clase.
- Es útil para crear métodos utilitarios que están relacionados conceptualmente con la clase, pero que no dependen de su estado interno.
- Se puede llamar tanto desde la clase como desde una instancia, pero su comportamiento será siempre el mismo.

---

```
class Calculadora:

    @staticmethod
    def sumar(a, b):
        return a + b

# Llamada desde la clase
print(Calculadora.sumar(5, 3))

# Llamada desde una instancia
calc = Calculadora()
print(calc.sumar(10, 7))
```

---

Esto es especialmente útil para mantener el código organizado: puedes agrupar funciones relacionadas dentro de una clase sin que estas funciones necesiten acceder al estado de dicha clase.

## *1er pilar de la programación orientada a objetos - Encapsulado*

Encapsulation es un principio fundamental de la programación orientada a objetos (OOP) que consiste en agrupar en una misma clase tanto los datos (atributos) como los métodos (funciones) que manipulan esos datos. Esto crea una especie de "caja" o "paquete" que protege el estado interno del objeto, permitiendo el acceso y la modificación sólo a través de métodos específicos. Por ejemplo, en una clase Persona, los atributos como nombre y edad están encapsulados junto con métodos como saludar() o cumplir_años(). Esta estructura no solo organiza el código, sino que también mejora la seguridad y el control, ya que se puede restringir el acceso directo a ciertos atributos usando modificadores como _atributo o __atributo para indicar distintos niveles de privacidad.

---

```
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre  # Atributo privado
        self.__edad = edad      # Atributo privado

    def saludar(self):
        print(f"Hola, mi nombre es {self.__nombre} y tengo {self.__edad} años.")

    def cumplir_años(self):
        self.__edad += 1

    def obtener_edad(self):
        return self.__edad

    def establecer_nombre(self, nuevo_nombre):
        self.__nombre = nuevo_nombre


persona1 = Persona("Ana", 30)
persona1.saludar()  # Hola, mi nombre es Ana y tengo 30 años.
persona1.cumplir_años()
print(persona1.obtener_edad())  # 31
persona1.establecer_nombre("Ana María")
persona1.saludar()  # Hola, mi nombre es Ana María y tengo 31 años.
```

---

**¿Qué muestra la encapsulación aquí?**

- Los atributos __nombre y __edad están encapsulados, es decir, no son accesibles directamente desde fuera de la clase.
- Solo se pueden modificar o leer usando métodos definidos, lo que proporciona control y seguridad sobre cómo se accede a los datos internos.

## *2do pilar de la programación orientada a objetos - Abstracción*

Abstracción significa ocultar los detalles internos de implementación y mostrar solo lo esencial para el usuario. Esto ayuda a simplificar la interacción con objetos complejos, mostrando únicamente lo que se necesita.

---

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

    def __procesar_nombre(self):
        # Método interno que no necesita conocer el usuario
        return self.__nombre.strip().title()

    def saludar(self):
        nombre_limpio = self.__procesar_nombre()
        print(f"Hola, mi nombre es {nombre_limpio} y tengo {self.__edad} años.")

persona1 = Persona("   ana", 25)
persona1.saludar()  # Hola, mi nombre es Ana y tengo 25 años.
```

---

**¿Dónde está la abstracción?**

- El método __procesar_nombre() es interno (privado). El usuario de la clase no necesita saber cómo se procesa el nombre, solo le interesa que saludar() funcione correctamente.
- El usuario accede a una interfaz simple (saludar()) sin preocuparse por la lógica interna.

### *Variables publicas vs privadas*

En Python, las variables privadas y públicas se usan para controlar el acceso a los atributos de una clase. Aunque Python no tiene encapsulación estricta como otros lenguajes (por ejemplo, Java o C++), sigue ciertas convenciones que ayudan a marcar una variable como privada o pública.

***Variables publicas***

Estas variables se pueden acceder y modificar libremente desde fuera de la clase.

---

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

p = Persona("Ana")
print(p.nombre)  # Acceso permitido
p.nombre = "Laura"  # Modificación permitida
print(p.nombre)
```

---

***Variables Privadas***

En Python, las variables privadas se indican anteponiendo doble guión bajo (__) al nombre de la variable. Esto activa el name mangling, haciendo que no se pueda acceder directamente desde fuera de la clase.

---

```
class Persona:
    def __init__(self, nombre):
        self.__nombre = nombre  # privada

p = Persona("Ana")
# print(p.__nombre) ❌ Esto da error: AttributeError

# Acceso usando name mangling (no recomendado)
print(p._Persona__nombre)  # ✅ Acceso posible pero no es una buena práctica
```

---

- Para proteger datos sensibles.
- Para evitar modificaciones accidentales desde fuera de la clase.
- Para forzar a usar métodos de acceso como getters y setters.

### *3er pilar de la programación orientada a objetos - Herencia*

Es uno de los pilares fundamentales de la programación orientada a objetos. Permite crear una nueva clase (llamada clase hija o derivada) que hereda los atributos y métodos de una clase existente (llamada clase padre o base), lo que promueve la reutilización del código y la organización jerárquica. En Python, esto se logra indicando entre paréntesis el nombre de la clase padre al definir la clase hija. La clase hija puede extender la funcionalidad de la clase padre agregando nuevos atributos o métodos, o incluso sobrescribiendo los existentes. A continuación, se muestra un ejemplo donde la clase Student hereda de la clase Person, aprovechando el constructor y el método greet definidos en la clase base:

---

```
# Clase base (padre)
class Person:
    def __init__(self, name, lastname, age):
        self.name = name
        self.lastname = lastname
        self.age = age

    def greet(self):
        print(f"Hola, soy {self.name} {self.lastname} y tengo {self.age} años.")

# Clase derivada (hija)
class Student(Person):
    def __init__(self, name, lastname, age, student_id):
        super().__init__(name, lastname, age)  # Llamamos al constructor de la clase padre
        self.student_id = student_id

    def mostrar_estudiante(self):
        print(f"Soy un estudiante con ID: {self.student_id}")

# Creamos una instancia de Student
estudiante = Student("Lucía", "González", 21, "A12345")
estudiante.greet()              # Método heredado de la clase Person
estudiante.mostrar_estudiante()  # Método propio de Student
```

---

- super().__init__() permite llamar al constructor de la clase padre (Person) para reutilizar código.
- Student hereda automáticamente todos los métodos y atributos públicos de Person.
- Podés añadir más métodos o atributos en la clase hija, o incluso sobrescribir los del padre si necesitás comportamiento diferente.