Programación Orientada a Objetos

python

INDICE

1. Introducción

La Programación Orientada a Objetos (POO) es un paradigma de programación que permite organizar el código en torno a "objetos" que representan entidades del mundo real.

Cada objeto combina datos (atributos) y funciones (métodos) que definen su comportamiento.

Ejemplo del mundo real:
Piensa en un coche:

- Tiene atributos como color, marca, modelo, año.
- Tiene métodos como arrancar(), frenar(), acelerar().

En Python, una clase define el modelo general (la plantilla del coche), y un objeto es una instancia concreta (por ejemplo, "mi_coche_rojo").

Conceptos fundamentales

Concepto	Descripción	Ejemplo
Clase	Molde o plantilla a partir de la cual se crean objetos.	class Coche:
Objeto / Instancia	Ejemplo concreto creado a partir de una clase.	mi_coche = Coche()
Atributos	Variables que almacenan datos dentro de un objeto.	mi_coche.color = "rojo"
Métodos	Funciones que definen el comportamiento del objeto.	mi_coche.arrancar()
Encapsulación	Ocultar los detalles internos de un objeto.	Métodos y atributos privados
Herencia	Una clase hija hereda de otra clase padre.	class Perro(Mamifero)
Polimorfismo	Permite usar el mismo método en diferentes clases.	animal.hablar() en Perro y Gato
Composición	Una clase contiene objetos de otra clase.	Coche tiene una Batería
Dunder method	Métodos especiales (doble guion bajo) que controlan el comportamiento nativo de Python.	__init__, __str__

2. Crear una clase y una instancia

En Python, una clase es una plantilla que describe cómo serán los objetos que creemos a partir de ella: qué información guardarán (atributos) y qué acciones podrán realizar (métodos).
Cuando usamos una clase para crear un objeto concreto, decimos que instanciamos la clase, y el objeto que obtenemos es una instancia de esa clase.

Para entenderlo bien, imagina lo siguiente:

- La clase es como el plano de un edificio.
- La instancia es el edificio real construido con ese plano.
- Cada edificio puede tener un color o una altura distinta, aunque todos se basen en el mismo plano.

2.1 Estructura básica de una clase en Python

En Python, las clases se crean con la palabra reservada class:

```
class NombreDeClase:
    # Aquí se definen los atributos y métodos
    ...
```

Por convención, el nombre de la clase comienza con mayúscula y si contiene varias palabras se usa el estilo CamelCase, por ejemplo:

class CocheElectrico, class CuentaBancaria, etc.

El método especial __init__()

Cuando creas una clase, normalmente quieres que los objetos empiecen con cierta información básica.
Para eso se utiliza el método constructor __init__().

El método __init__() se ejecuta automáticamente cada vez que se crea una nueva instancia de la clase.
Dentro de este método, se inicializan los atributos del objeto.

Su estructura más simple es esta:

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

¿Qué es self y por qué es obligatorio?

El parámetro self aparece en todos los métodos de las clases en Python (al menos en los métodos de instancia).
Es uno de los conceptos más importantes de la POO en Python.

self representa la propia instancia del objeto que estás creando o manipulando.
Cada vez que accedes a un atributo o método de ese objeto, lo haces a través de self.

Veamos qué significa exactamente:

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

Cuando creas una persona:

```
juan = Persona("Juan")
```

Python traduce internamente esta llamada a algo así como:

```
Persona.__init__(juan, "Juan")
```

Es decir, Python pasa automáticamente el objeto recién creado como primer argumento del método, y ese objeto se asigna al parámetro self.
Por tanto:

- self.nombre = nombre quiere decir: Guarda el valor del parámetro nombre dentro del atributo nombre del objeto actual.
- Así, juan.nombre contendrá "Juan".

Si olvidaras poner self en la definición del método, el intérprete te daría un error del tipo:
TypeError: __init__() takes 2 positional arguments but 3 were given
Esto ocurre porque Python siempre pasa el objeto (self) automáticamente, aunque tú no lo pongas en la llamada.

Crear la clase y las instancias

Veamos un ejemplo completo:

```
class Dog:

    def __init__(self, name, age):
        """Inicializa los atributos del perro.

        - self: referencia al propio objeto.
        - name: nombre del perro.
        - age: edad del perro.
        """
        self.name = name
        self.age = age

    def sit(self):
        """Simula que el perro se sienta."""
        print(f"{self.name} se ha sentado.")

    def roll_over(self):
        """Simula que el perro se revuelca."""
        print(f"{self.name} se da la vuelta.")
```

• class Dog:

Declara la clase llamada Dog. Por convención, los nombres de clase comienzan con mayúscula.

• def __init__(self, name, age): Es el método constructor. Se ejecuta automáticamente cada vez que creamos un nuevo perro.

• self.name = name y self.age = age: Crea los atributos name y age dentro de la instancia. Cada perro tendrá su propio name y age.

Por ejemplo, un perro puede llamarse “Toby” y otro “Luna”.

• def sit(self): y def roll_over(self): Son métodos normales.

Toman self como primer parámetro para acceder a los datos del propio perro.
Dentro de ellos, puedes usar los atributos (self.name, self.age) o definir comportamientos.

2.2 Crear instancias

Una instancia se crea llamando al nombre de la clase como si fuera una función:

```
mi_perro = Dog("Toby", 4)
tu_perro = Dog("Luna", 2)
```

¿Qué pasa internamente cuando haces esto?

1. Python crea un objeto vacío del tipo Dog.
2. Llama al método __init__() pasando ese nuevo objeto como self y los argumentos "Toby" y 4.
3. Dentro del __init__, se guardan los datos en self.name y self.age.
4. Python devuelve el objeto completamente configurado y lo asigna a mi_perro.

Por tanto:

- mi_perro.name → "Toby"
- mi_perro.age → 4

Cada instancia tiene sus propios datos almacenados de forma independiente.

2.3 Acceder a atributos y métodos

Para acceder a los datos (atributos) o ejecutar acciones (métodos) se usa la **notación punto** (.):

```
print(mi_perro.name)   # Accede al atributo 'name'
print(mi_perro.age)    # Accede al atributo 'age'

mi_perro.sit()         # Llama al método sit()
mi_perro.roll_over()   # Llama al método roll_over()
```

Crear varias instancias independientes

Cada objeto creado a partir de la clase Dog es **independiente** de los demás.

```
tu_perro = Dog("Luna", 2)

print(f"Mi perro se llama {mi_perro.name} y tiene {mi_perro.age} años.")
print(f"Tu perro se llama {tu_perro.name} y tiene {tu_perro.age} años.")
```

Aunque ambos son perros creados de la misma clase, cada uno conserva sus propios datos.

Definir un valor por defecto para un atributo

Al crear una instancia de una clase, puede haber atributos definidos sin un valor que se le haya pasado por parámetro al constructor.

```
class Car:
    def __init__(self, make, model, year):

        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
```

Modificar valores de los atributos

Se pueden modificar los valores de los atributos de 2 formas:

- Modificar el valor del atributo directamente: la forma más simple es acceder al atributo a través de su instancia.

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar_info(self):
        print(f"Marca: {self.marca}, Modelo: {self.modelo}")
c1 = Coche("Toyota", "Corolla")
c1.mostrar_info()
c1.marca = "Honda"
```

Accedemos directamente al atributo a través de la instancia de c1 de Coche.

- Modificar el valor del atributo a través de un método: en vez de acceder al atributo directamente, se pasa el valor del atributo a una función que será la encargada de modificar el atributo.

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar_info(self):
        print(f"Marca: {self.marca}, Modelo: {self.modelo}")

    def update_odometer(self, mileage):
        self.odometer_reading = mileage

c1 = Coche("Toyota", "Corolla")
c1.mostrar_info()
c1.update_odometer(23)
```

Solo se ha añadido a la clase el método update_odometer.
Se puede añadir la funcionalidad de "update_odometer"para realizar ciertas acciones antes de asignar el valor a la propiedad:

```
def update_odometer(self, mileage):
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("No uede retroceder el odómetro.")
```

Este método incrementa el odómetro un número de millas que se le pase por parámetro.

2.4 Atributos de instancia vs atributos de clase

- **Instancia**: datos propios de cada objeto (normalmente en __init__). Pertenecen a cada objeto individual.
- **Clase**: valor compartido por todas las instancias.

```
class Empleado:
    empresa = "ACME"          # atributo de clase, compartido
    def __init__(self, nombre):
        self.nombre = nombre  # atributo de instancia

e1 = Empleado("Laura")
e2 = Empleado("Carlos")
print(e1.empresa, e2.empresa)     # ACME ACME
Empleado.empresa = "OpenAI"
print(e1.empresa, e2.empresa)     # OpenAI OpenAI
```

Si asignas e1.empresa = "X", **creas** un atributo de instancia nuevo que **sobrescribirá** al de clase, pero no cambia el de las demás instancias.

2.5 Errores comunes al crear clases

- Olvidar self en los métodos:

```
class Perro:
    def ladrar():
```

```
        print("Guau de "+ self.nombre)
```
- Usar mal los nombres de atributos:
```
class Persona:
    def __init__(self, nombre):
        nombre = nombre  # Error: no se usa self
```
Así no se guarda nada en la instancia. Debe ser self.nombre = nombre.
- Confundir métodos de clase con funciones globales:
Si defines funciones fuera de la clase, **no tienen self**.
Dentro de la clase, **siempre** debe ir self como primer argumento.
Ejemplo completo:
```
class Coche:
    """Modelo simple de un coche."""

    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def arrancar(self):
        print(f"El {self.marca} {self.modelo} está arrancando.")

    def describir(self):
        print(f"Este coche es un {self.marca} {self.modelo} de color
{self.color}.")
```

```
mi_coche = Coche("Toyota", "Corolla", "rojo")
tu_coche = Coche("Ford", "Focus", "azul")

mi_coche.arrancar()
tu_coche.describir()
```

Salida:
El Toyota Corolla está arrancando.
Este coche es un Ford Focus de color azul.

- La clase Coche es el plano.
- mi_coche y tu_coche son coches reales diferentes.
- Cada uno tiene su color y comportamiento independiente.

Ejercicio 1
Crea una clase Persona con los atributos nombre y edad. Debe tener un método presentarse() que imprima algo como: “Hola, me llamo Ana y tengo 25 años.”

2.6 Tipos de métodos: instancia, clase y estáticos

En Python existen **tres tipos principales** de métodos dentro de una clase:
- Método de instancia: su primer parámetro es “self” y pertenece a una instancia(objeto), se usa para acceder o modificar atributos del objeto.
- Método de clase: su primer parámetro es “cls” y pertenece a la clase entera, se usa para crear o modificar datos comunes a todas las instancias.

- Método estático: no tiene parámetros obligatorios, no depende ni de clase ni de objeto y sirve para crear funciones auxiliares relacionadas con la clase.

Métodos de instancia

Son los más comunes. Se definen **con el primer parámetro self**, que representa **la instancia actual** (el propio objeto).

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.velocidad = 0

    def acelerar(self, cantidad):
        self.velocidad += cantidad
        print(f"El coche {self.marca} ahora va a {self.velocidad} km/h.")
```

- self hace referencia al objeto concreto que invoca el método.
- acelerar() accede al atributo velocidad del coche que lo llamó.

Uso:

```
c1 = Coche("Toyota", "Yaris")
c2 = Coche("Ford", "Focus")

c1.acelerar(20)  # El coche Toyota ahora va a 20 km/h.
c2.acelerar(40)  # El coche Ford ahora va a 40 km/h.
```

Cada coche mantiene su propia velocidad: **los métodos de instancia actúan sobre los datos del objeto individual**.

Métodos de clase

Afectan a la **clase en general**, no a un objeto concreto. Se definen con el **decorador @classmethod**, y reciben como primer parámetro cls (que representa la clase).

Ejemplo:

```
class Coche:
    numero_de_coches = 0  # atributo de clase

    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        Coche.numero_de_coches += 1

    @classmethod
    def coches_creados(cls):
        print(f"Se han creado {cls.numero_de_coches} coches en total.")
```

- cls hace referencia a **la clase Coche**, no a un objeto.
- Sirve para **acceder o modificar atributos de clase** (compartidos por todos los objetos).

Uso:

```
c1 = Coche("Toyota", "Yaris")
c2 = Coche("Ford", "Focus")

Coche.coches_creados()  # Se han creado 2 coches en total.
```

También se puede llamar desde una instancia:

```
c1.coches_creados()  # También funciona, pero internamente usa la clase.
```

A veces se usa @classmethod para crear **formas alternativas de construir un objeto**.

```
class Fecha:
    def __init__(self, dia, mes, año):
        self.dia = dia
        self.mes = mes
        self.año = año

    @classmethod
    def desde_cadena(cls, cadena):
        """Crea una Fecha a partir de una cadena 'DD-MM-AAAA'."""
        d, m, a = cadena.split("-")
        return cls(d, m, a)  # llama al constructor original
```

Uso:

```
f = Fecha.desde_cadena("04-11-2025")
print(f.dia, f.mes, f.año)  # 4 11 2025
```

Aquí el método de clase devuelve una nueva instancia usando un formato distinto.

Métodos estáticos

Son funciones **independientes** de la clase y de sus objetos, pero que se colocan dentro de la clase **porque tienen relación lógica con ella**, no reciben ni self ni cls.

Ejemplo:

```
class Matematicas:
    @staticmethod
    def es_par(numero):
        """Comprueba si un número es par."""
        return numero % 2 == 0
```

Uso:

```
print(Matematicas.es_par(4))  # True
print(Matematicas.es_par(7))  # False
```

Ejemplo combinado de los 3 tipos:

```
class Circulo:
    pi = 3.1416  # atributo de clase

    def __init__(self, radio):
        self.radio = radio

    # Método de instancia
    def area(self):
        """Calcula el área del círculo."""
        return Circulo.pi * (self.radio ** 2)

    # Método de clase
    @classmethod
    def desde_diametro(cls, diametro):
        """Crea un círculo a partir del diámetro."""
        radio = diametro / 2
        return cls(radio)
```

```
    # Método estático
    @staticmethod
    def info():
        """Muestra información general sobre los círculos."""
        print("El área de un círculo se calcula como π * r²")
```

Uso:

```
# 1) Usar método de instancia
c1 = Circulo(5)
print(c1.area())  # 78.54

# 2) Usar método de clase como constructor alternativo
c2 = Circulo.desde_diametro(10)
print(c2.radio)   # 5.0

# 3) Usar método estático
Circulo.info()    # El área de un círculo se calcula como π * r²
```

Ejercicio 2

Crea una clase Rectangulo con:

- Un **método de instancia** area() que devuelva base × altura;
- Un **método de clase** cuadrado(lado) que cree un cuadrado con base = altura.
- Un **método estático** es_valido(base, altura) que devuelva True si ambos son positivos.

3. Encapsulación en Python: público, protegido y privado (convenciones)

Python no impone "private/public", usa **convenciones**:
- self.nombre: **público**.
- self._nombre: **protegido** (convención: úsalo solo internamente).
- self.__nombre: **privado**.

Guión bajo simple

En Python, poner un solo guion bajo al inicio del nombre indica que ese atributo o método no debería usarse fuera de la clase, pero no lo impide.
Es solo una convención de programadores, no una restricción del lenguaje.

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre  # "protegido", por convención

p = Persona("Miguel")
print(p._nombre)  # Se puede acceder, pero no se recomienda
conjunto1.update(conjunto2)
print(conjunto1)
```

Uso típico: Indicar que es un atributo "interno" (no parte de la interfaz pública de la clase).
En herencia, también se usa para atributos "protegidos" (que pueden usar las subclases, pero no desde fuera).

Doble guión bajo

Cuando usas dos guiones bajos al inicio, Python renombra internamente el atributo para evitar que sea sobrescrito accidentalmente en las subclases.

Internamente, el atributo __nombre se convierte en _NombreDeLaClase__nombre. A esto Python lo llama “name mangling”.

```
class Persona:
    def __init__(self, nombre):
        self.__nombre = nombre  # privado con name mangling

    def mostrar(self):
        print(self.__nombre)


p = Persona("Miguel")
p.mostrar()  # Funciona

# print(p.__nombre)  # Error: AttributeError
print(p._Persona__nombre)  # Se puede acceder, pero no deberías
```

Python hace esto internamente:

self.__nombre → se convierte en self._Persona__nombre

Uso típico: Cuando quieres evitar colisiones de nombres en herencia o proteger realmente un atributo del acceso directo externo.

3.1 @property: getters/setters “a lo Python”

Los getters y setters son métodos especiales que se utilizan para controlar el acceso a los atributos de un objeto.

- El getter sirve para obtener (leer) el valor de un atributo.
- El setter sirve para modificar (escribir) el valor de un atributo.

Los getter y setter existen porque en programación orientada a objetos, no siempre es buena idea acceder directamente a los atributos de un objeto (por ejemplo: persona.edad = -5 no tiene sentido).

Los getters y setters te permiten:

- Validar los datos antes de asignarlos.
- Ejecutar acciones cuando un valor cambia.
- Proteger los atributos internos para que no se modifiquen incorrectamente.

Ejemplo básico (sin decoradores)

Antes de usar los decoradores modernos, así se hacía en muchos lenguajes (Java, C++, etc.):

```
class Persona:
    def __init__(self, nombre, edad):
        self._nombre = nombre
        self._edad = edad  # atributo “protegido”

    def get_edad(self):
        return self._edad
```

```
    def set_edad(self, valor):
        if valor < 0:
            print("La edad no puede ser negativa.")
        else:
            self._edad = valor
```

Uso:

```
p = Persona("Ana", 25)
print(p.get_edad())  # 25
p.set_edad(-10)      # La edad no puede ser negativa.
p.set_edad(30)
print(p.get_edad())  # 30
```

Esto funciona, pero no es el estilo Python moderno.
En Python preferimos usar propiedades con el decorador @property.

Usando @property

Python tiene un sistema más limpio y elegante para hacer esto: el decorador @property.
El método *getter*, contiene @property antes de la función getter.
El método *setter*, debe tener el mismo nombre que el getter, y se decora con @nombre.setter.
Ejemplo:

```
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre
        self.edad = edad  # usa el setter

    @property
    def edad(self):
        """Getter: devuelve el valor de _edad."""
        return self.__edad

    @edad.setter
    def edad(self, valor):
        """Setter: valida antes de asignar el valor."""
        if valor < 0:
            raise ValueError("La edad no puede ser negativa.")
        self.__edad = valor
```

Uso:

```
p = Persona("Carlos", 25)
print(p.edad)   # usa el getter → 25

p.edad = 30     # usa el setter
print(p.edad)   # 30

p.edad = -5     # Lanza ValueError: La edad no puede ser negativa
```

Ventajas de @property

- Desde fuera de la clase, parece que estás accediendo a un atributo normal (p.edad), pero en realidad se está ejecutando un método.

- Puedes añadir validaciones, logs o cálculos antes de añadir un valor a una propiedad del objeto.

Ejemplo más completo:

```
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.__titular = titular
        self.saldo = saldo  # usa el setter

    @property
    def saldo(self):
        """Getter: devuelve el saldo actual."""
        return self.__saldo

    @saldo.setter
    def saldo(self, cantidad):
        """Setter: impide saldos negativos."""
        if cantidad < 0:
            raise ValueError("El saldo no puede ser negativo.")
        self.__saldo = cantidad

    def depositar(self, cantidad):
        self.saldo += cantidad  # usa el setter
        print(f"Se depositaron {cantidad}€. Nuevo saldo: {self.__saldo}")
```

Uso:

```
c = CuentaBancaria("Lucía", 1000)
print(c.saldo)   # 1000
c.depositar(500) # Se depositaron 500€. Nuevo saldo: 1500
c.saldo = -10    # ❌ ValueError: El saldo no puede ser negativo.
```

Observa que incluso dentro de la clase, el setter se usa automáticamente al hacer self.saldo = valor.

4. Encapsulación con atributos “privados”

Por convención, los atributos que empiezan con un guion bajo (_) se consideran protegidos.
Si quieres hacerlo más “privado”, usa doble guion bajo (__), aunque Python permite acceder igual (con name mangling).

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

    @property
    def edad(self):
        return self.__edad

    @edad.setter
    def edad(self, valor):
        if valor > 150:
            raise ValueError("Edad irreal.")
```

```
        self.__edad = valor
```

Uso:

```
p = Persona("María", 30)
print(p.edad)    # 30
p.edad = 200     # ❌ ValueError
print(p._Persona__edad)  # Acceso interno posible, pero desaconsejado
```

5. Getter sin setter (atributo de solo lectura)

A veces queremos que un valor solo se pueda consultar, no modificar.

```
class Circulo:
    def __init__(self, radio):
        self._radio = radio

    @property
    def radio(self):
        return f"El radio es: {str(self._radio)}"
```

Uso:

```
c = Circulo(5)
print(c.radio)   # 5
c.radio = 100    # AttributeError Circulo no tiene setter
```

@property sin @setter = solo lectura.

Ejemplo con validaciones:

```
class Producto:
    def __init__(self, nombre, precio):
        self.__nombre = nombre
        self.precio = precio  # usa el setter

    @property
    def precio(self):
        """Getter"""
        return self.__precio

    @precio.setter
    def precio(self, valor):
        """Setter con validación"""
        if valor < 0:
            raise ValueError("El precio no puede ser negativo.")
        self.__precio = round(valor, 2)

    def aplicar_descuento(self, porcentaje):
        """Método normal que usa el setter internamente"""
        self.precio *= (1 - porcentaje / 100)
```

Uso:

```
p = Producto("Portátil", 1200)
print(p.precio)   # 1200.0
```

```
p.aplicar_descuento(10)
print(p.precio)   # 1080.0

p.precio = -50    # ValueError: El precio no puede ser negativo
```

Ejercicio 3
Crea una clase Rectangulo con atributos base y altura.
Usa @property para:

- Validar que ambos sean positivos.
- Calcular el área como propiedad de solo lectura (area).

Ejercicio 4
Crea una clase llamada Producto que represente un artículo en un inventario.
Requisitos de la Clase:

1. Atributos de Instancia:
    - _nombre (string): Nombre del producto.
    - _precio (float): Precio de venta del producto.
    - _stock (int): Cantidad disponible en el inventario.
2. Atributo de Clase:
    - total_inventario (int): Debe llevar la cuenta total de unidades en stock de *todos* los productos.
3. Encapsulación y Validación (@property):
    - Implementa el getter y setter para precio. El setter debe asegurar que el precio sea siempre mayor que 0.0. Si se intenta asignar un precio no válido, debe lanzar una excepción ValueError: raise ValueError("El precio debe ser mayor que cero.")
    - Implementa el getter y setter para stock. El setter debe asegurar que el stock sea mayor o igual a 0. Además, cada vez que se modifique el stock, el atributo de clase total_inventario debe actualizarse.
4. Método de Clase:
    - @classmethod mostrar_total_inventario(): Un método que imprima el valor actual de total_inventario.
5. Método de Instancia:
    - vender(cantidad): Un método que descuente cantidad del stock actual si hay suficiente. Si no hay suficiente stock, debe lanzar una excepción ValueError.

Mejora la representación del objeto para la salida usando __str__, el precio con 2 decimales

6. Herencia: reutilizar y especializar

La Herencia es uno de los pilares de la Programación Orientada a Objetos. Permite que una clase (la clase hija o subclase) adquiera (herede) automáticamente todos los atributos y métodos de otra clase (la clase padre o superclase).
Este concepto modela la relación "es un/a" (por ejemplo, un Perro es un Animal).

6.1 La función super()

La función super() es fundamental en la herencia. Se utiliza dentro de la clase hija, principalmente en el método __init__, para llamar al constructor o a cualquier otro método de la clase padre.
Esto asegura que la clase hija inicialice correctamente los atributos definidos por el padre antes de inicializar sus propios atributos específicos.

Ejemplo de uso de super()

```
class Animal:
    def __init__(self, nombre, edad):
        """Constructor de la clase padre (Animal)."""
        self.nombre = nombre
        self.edad = edad

    def comer(self):
        """Método heredable."""
        print(f"{self.nombre} está comiendo.")

class Perro(Animal):
    def __init__(self, nombre, edad, raza):
        # Llamamos al constructor del padre para inicializar 'nombre' y
'edad'.
        super().__init__(nombre, edad)

        # Inicializamos el atributo específico de Perro.
        self.raza = raza

    def ladrar(self):
        print(f"{self.nombre} (un {self.raza}) dice ¡Guau!")
```

Uso:

```
mi_perro = Perro("Max", 5, "Labrador")
mi_perro.comer()  # Método heredado
mi_perro.ladrar() # Método específico
```

6.2 Sobreescritura de métodos (Overriding)

La Sobreescritura permite a la clase hija cambiar o especializar la implementación de un método que ya existe en la clase padre. Cuando se llama al método desde una instancia de la clase hija, se ejecutará la versión de la hija.
Esto es un ejemplo directo de **Polimorfismo**.

```
class Ave:
    def volar(self):
        """Comportamiento general de un ave."""
        print("El ave vuela alto usando sus alas.")

class Pinguino(Ave):
    def volar(self):
        # Sobreescritura: Cambiamos el comportamiento, el pingüino no vuela.
        print("El pingüino no puede volar, solo camina o nada rápidamente.")
```

Uso:

```
pinguino = Pinguino()
pinguino.volar() # Ejecuta la versión sobreescrita: El pingüino no puede
volar...
```

6.3 Herencia Múltiple (Cuidado)

Python permite que una clase herede de múltiples clases padre a la vez.

```
class PadreA:
    def metodo_a(self):
        print("Método A de PadreA")

class PadreB:
    def metodo_b(self):
        print("Método B de PadreB")

class Hijo(PadreA, PadreB):
    pass

h = Hijo()
h.metodo_a()
h.metodo_b()
```

6.4 El Problema del Diamante y MRO (Method Resolution Order)

Cuando dos clases padre tienen métodos con el mismo nombre, Python utiliza un algoritmo complejo llamado **MRO (Method Resolution Order)** para decidir qué método ejecutar.

Regla General de MRO: Python busca el método de izquierda a derecha en las clases listadas en la herencia (class Hijo(PadreA, PadreB)), y luego busca en sus propios padres.

Puedes consultar el orden de resolución usando la propiedad __mro__:

```
print(Hijo.__mro__)
# Muestra el orden exacto en que Python buscará los métodos.
```

Recomendación: La herencia múltiple es una herramienta potente pero compleja. Se recomienda usarla con mucha cautela. En su lugar, es preferible utilizar la **Composición** (la clase *tiene* funcionalidad de otra), que resulta en un código más claro y menos acoplado.

7. Polimorfismo y “duck typing”

El Polimorfismo (que significa "muchas formas") es la capacidad de diferentes objetos, pertenecientes a clases no relacionadas, de responder de manera única al mismo nombre de método.

En Python, el polimorfismo se implementa a través del principio de Duck Typing. Este principio establece que, si un objeto tiene los métodos necesarios, se considera adecuado para la tarea, sin importar de qué clase específica provenga.

"Si camina como un pato y grazna como un pato, entonces debe ser un pato."

Ejemplo de Polimorfismo

En el siguiente ejemplo, la función hacer_hablar() llama al método hablar(). No le importa si recibe un Gato o una Vaca, solo necesita que el objeto recibido (el "pato") tenga el método hablar().

```
class Gato:
    def hablar(self):
```

```
        return "Miau"

class Vaca:
    def hablar(self):
        return "Muuu"

class Pato:
    def hablar(self):
        return "Cuac"

def hacer_hablar(animal):
    """
    Función polimórfica que usa Duck Typing.
    Solo comprueba si el objeto 'animal' tiene el método 'hablar'.
    """
    print(animal.hablar())

# Creación de instancias
gato = Gato()
vaca = Vaca()
pato = Pato()

# La misma función maneja diferentes objetos
hacer_hablar(gato)
hacer_hablar(vaca)
hacer_hablar(pato)
```

8. Composición (y agregación): “tiene un/a” en lugar de “es un/a”

La Composición es una alternativa a la herencia que promueve la reutilización de código. En lugar de establecer una relación "es un/a" (Herencia), la Composición establece una relación "tiene un/a" (o "usa un/a").
Consiste en que una clase principal incluye instancias de otras clases como atributos. La clase principal delega responsabilidades a los objetos internos para realizar ciertas tareas.

Composición vs. Agregación

Aunque a menudo se usan indistintamente, existe una diferencia sutil:

- Composición: La parte no puede existir sin el todo. Si el objeto principal es destruido, la parte también lo es. (Ejemplo: Un motor es parte de un coche).
- Agregación: La parte puede existir independientemente del todo. (Ejemplo: Un empleado pertenece a una empresa, pero el empleado puede seguir existiendo si la empresa cierra).

En Python, esta distinción se implementa principalmente a nivel conceptual.

Ejemplo de Composición

Imaginemos que queremos crear una clase Coche. En lugar de hacer que Coche herede de Motor, hacemos que Coche tenga un objeto Motor.

```
class Motor:
    def __init__(self, tipo):
        self.tipo = tipo

    def arrancar(self):
        return f"Motor {self.tipo} arrancando. Vrooom."
```

```
class Coche:
    def __init__(self, color, tipo_motor):
        self.color = color
        # Composición: El Coche crea y "tiene" un Motor
        self.motor = Motor(tipo_motor)

    def conducir(self):
        # Delegación: El Coche delega la tarea de arrancar al Motor
        arranque_status = self.motor.arrancar()
        print(f"El coche {self.color} se pone en marcha. Status:
{arranque_status}")

# Uso
mi_coche = Coche("Rojo", "Diesel")
mi_coche.conducir()
```

Ventajas de la Composición:

1. Flexibilidad: Permite cambiar el comportamiento del Coche simplemente cambiando el tipo de Motor sin modificar la jerarquía de clases.
2. Bajo Acoplamiento: La clase Coche solo necesita saber que Motor tiene un método arrancar(), no cómo lo implementa internamente. Esto hace que el código sea más fácil de mantener y probar.

9. Clases Base Abstractas (ABC)

Las Clases Base Abstractas (ABC), disponibles en el módulo abc, se utilizan para definir un conjunto de métodos que deben ser implementados por cualquier subclase que herede de ellas.

Propósito de las ABC

1. Imponer Contrato (Interfaz): Aseguran que todas las clases hijas (concretas) cumplan un *contrato* de métodos.
2. Prevención de Instanciación: No se pueden crear instancias de una ABC directamente. Intentar hacerlo generará un error (TypeError), ya que una ABC es una plantilla incompleta.

Implementación

Se utiliza el decorador @abstractmethod sobre los métodos de la clase base que deben ser implementados por las clases hijas.

```
from abc import ABC, abstractmethod

# 1. Definición de la ABC
class FiguraGeometrica(ABC):
    """Clase base abstracta para cualquier figura."""

    @abstractmethod
    def area(self) -> float:
        """Calcula el área de la figura. Debe ser implementado."""
        pass # La implementación de la ABC es solo 'pass'

    @abstractmethod
    def perimetro(self) -> float:
        """Calcula el perímetro de la figura. Debe ser implementado."""
```

```
        pass

    # Un método concreto que las subclases heredan
    def describir(self):
        print("Esta es una figura geométrica.")


# 2. Clase Concreta que hereda e implementa
class Rectangulo(FiguraGeometrica):
    def __init__(self, ancho: float, alto: float):
        self.ancho = ancho
        self.alto = alto

    # Implementación OBLIGATORIA del método abstracto 'area'
    def area(self) -> float:
        return self.ancho * self.alto

    # Implementación OBLIGATORIA del método abstracto 'perimetro'
    def perimetro(self) -> float:
        return 2 * (self.ancho + self.alto)

# Uso
r = Rectangulo(5, 4)
print(f"Área: {r.area()}")
# f = FiguraGeometrica() # ERROR: No se puede instanciar FiguraGeometrica
```

10. Protocolos de Tipado (typing.Protocol)

Los Protocolos son el enfoque moderno y preferido en Python para definir interfaces, especialmente cuando se trabaja con Duck Typing y type hints. Están disponibles en el módulo typing (desde Python 3.8).

Propósito de los Protocolos

1. Definir la Interfaz de Duck Typing: Permite especificar, con type hints, qué métodos y atributos debe tener un objeto para ser considerado de ese tipo.
2. No Requiere Herencia: A diferencia de las ABC, una clase no necesita heredar del Protocolo; solo necesita implementar la estructura definida.

<u>Implementación</u>

Se define un Protocolo creando una clase que hereda de typing.Protocol. Los métodos se definen con solo la firma y ...(puntos suspensivos).

```
from typing import Protocol, runtime_checkable

# 1. Definición del Protocolo (la interfaz 'Guardable')
@runtime_checkable # Permite usar isinstance() con este Protocolo
class Guardable(Protocol):
    """
    Cualquier objeto que tenga un método 'guardar' y un atributo
'fecha_creacion'
    cumple con este Protocolo.
    """
    fecha_creacion: str # Atributo requerido
```

```python
    def guardar(self, ruta: str) -> bool: ... # Método requerido (sin
implementación)

# 2. Clases que cumplen el Protocolo SIN HEREDAR
class Documento:
    def __init__(self, nombre: str):
        self.nombre = nombre
        self.fecha_creacion = "2024-10-25" # Cumple con el atributo

    def guardar(self, ruta: str) -> bool: # Cumple con el método
        print(f"Documento '{self.nombre}' guardado en {ruta}")
        return True

class Imagen:
    def __init__(self, nombre: str):
        self.nombre = nombre
        self.fecha_creacion = "2024-10-26" # Cumple con el atributo

    def guardar(self, ruta: str) -> bool: # Cumple con el método
        print(f"Imagen '{self.nombre}' exportada a {ruta}")
        return True

# 3. Función que utiliza el Protocolo
def procesar_guardado(objeto: Guardable, archivo: str):
    """La función solo necesita que el objeto sea Guardable."""
    if objeto.guardar(archivo):
        print(f"Objeto procesado con fecha: {objeto.fecha_creacion}")

# Uso
doc = Documento("Reporte_Mensual")
img = Imagen("Foto_Perfil")

procesar_guardado(doc, "/data/reporte.txt")
procesar_guardado(img, "/data/foto.jpg")

# Verificación de cumplimiento del Protocolo
print(f"Documento es Guardable: {isinstance(doc, Guardable)}") # True
```

11. Métodos especiales (dunder methods) más útiles

Los Dunder Methods (de *Double Underscore*, doble guion bajo) son métodos especiales de Python que te permiten definir cómo deben comportarse tus objetos cuando interactúan con funciones o sintaxis nativas del lenguaje. Al definirlos en tus clases, puedes dar a tus objetos el mismo comportamiento de los tipos integrados como listas, cadenas o números.

Dunder Methods Esenciales

Dunder Method	Uso Nativo	Descripción

__init__(self, ...)	Clase(...)	**Constructor**. Se llama al crear una nueva instancia.
__str__(self)	print(objeto) o str(objeto)	**Representación legible (informal)**. Debe devolver una cadena simple que un usuario leería.
__repr__(self)	repr(objeto) o consola interactiva	**Representación no ambigua (formal)**. Debe devolver una cadena que permita reconstruir el objeto (Clase(arg1, arg2)).
__len__(self)	len(objeto)	Define el tamaño o longitud del objeto. Debe devolver un entero.

Ejemplo:

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

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

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

Uso:

```
p = Persona("Miguel", 30)

print(p)        # usa __str__
# 👉 Miguel, 30 años

print(str(p))   # también usa __str__
# 👉 Miguel, 30 años

print(repr(p))  # usa __repr__
# 👉 Persona('Miguel', 30)
```

Dunder Methods para Comparación

Dunder Method	Operador	Descripción
__eq__(self, otro)	objeto1 == objeto2	Define la igualdad (equal to).
__lt__(self, otro)	objeto1 < objeto2	Define la comparación "menor que" (less than).

Ejemplo de una clase Libro:
Al implementar estos métodos, el objeto Libro puede usarse con funciones nativas (print, len) y operadores (==) como si fuera un tipo integrado.

```
class Libro:
    def __init__(self, titulo, paginas):
        self.titulo = titulo
        self.paginas = paginas
```

```
    # 1. Representación legible para el usuario (print)
    def __str__(self):
        return f"{self.titulo} ({self.paginas} páginas)"

    # 2. Representación no ambigua para el desarrollador (repr)
    def __repr__(self):
        return f"Libro(titulo='{self.titulo}', paginas={self.paginas})"

    # 3. Permite usar len()
    def __len__(self):
        return self.paginas

    # 4. Permite usar el operador == (comparación por título y páginas)
    def __eq__(self, otro):
        if isinstance(otro, Libro):
            return self.titulo == otro.titulo and self.paginas == otro.paginas
        return False

# Uso
libro_a = Libro("Cien años de soledad", 417)
libro_b = Libro("Cien años de soledad", 417)
libro_c = Libro("Ficciones", 176)

print(libro_a)         # Llama a __str__: Cien años de soledad (417 páginas)
print(repr(libro_a))   # Llama a __repr__: Libro(titulo='Cien años de
soledad', paginas=417)
print(len(libro_a))    # Llama a __len__: 417
print(libro_a == libro_b) # Llama a __eq__: True
print(libro_a == libro_c) # Llama a __eq__: False
```

12. @dataclass: clases de datos sin boilerplate

El decorador @dataclass (introducido en Python 3.7 y disponible en el módulo dataclasses) es una herramienta poderosa para simplificar la creación de clases cuyo propósito principal es almacenar datos, en lugar de manejar lógica compleja.
Al usar @dataclass, Python genera automáticamente gran parte del código repetitivo ("boilerplate") que normalmente tendrías que escribir a mano, incluyendo los Dunder Métodos esenciales:

- El método constructor __init__().
- El método de representación __repr__().
- El método de comparación de igualdad __eq__().

Ejemplo: Comparación de sintaxis
Al comparar una clase estándar con una @dataclass, la ventaja en concisión es notable: el foco se pone directamente en la definición de los campos de datos.

```
from dataclasses import dataclass

# Clase tradicional (requiere __init__ y __repr__ manuales)
class PuntoTradicional:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y
```

```
    def __repr__(self):
        return f"PuntoTradicional(x={self.x}, y={self.y})"

# La clase equivalente usando @dataclass (más concisa)
@dataclass
class PuntoDataclass:
    x: int
    y: int

# Uso y verificación
p1_trad = PuntoTradicional(10, 20)
p1_data = PuntoDataclass(10, 20)
p2_data = PuntoDataclass(10, 20)

print("Tradicional:", p1_trad)
print("Dataclass:", p1_data)
print("Comparación (eq):", p1_data == p2_data) # El comparador '==' se genera automáticamente.
```

Opciones clave de @dataclass

El decorador acepta argumentos que modifican el comportamiento de la clase generada:

- repr=False (predeterminado: True): Evita la generación automática de __repr__.
- eq=False (predeterminado: True): Evita la generación automática de __eq__ (igualdad).
- order=True (predeterminado: False): Genera los métodos de comparación (__lt__, __le__, __gt__, __ge__) basándose en el orden de declaración de los campos. Útil para ordenar listas de objetos.
- frozen=True (predeterminado: False): Hace que las instancias sean inmutables; no se puede cambiar el valor de los campos después de la creación, similar a una tupla.

Ejemplo:

```
@dataclass(frozen=True, eq=False)
class ProductoInmutable:
    nombre: str
    codigo: int
```

13. Copia de objetos: superficial vs profunda

En Python, al asignar una variable a otra (objeto_b = objeto_a), solo se crea una nueva referencia al mismo objeto. Esto significa que si modificas un objeto mutable a través de objeto_b, también modificas el objeto original al que apunta objeto_a.
Para crear copias verdaderamente independientes, utilizamos el módulo copy.

Copia Superficial (copy.copy())

Una copia superficial crea un nuevo objeto compuesto, pero luego inserta referencias a los objetos que se encuentran dentro del original.

- Tipos Inmutables (cadenas, números, tuplas): Funciona perfectamente, ya que son inmutables.
- Tipos Mutables Anidados (listas, objetos): NO copia los objetos anidados, solo sus referencias. Si modificas un objeto mutable dentro de la copia, el objeto original también se verá afectado.+

Copia Profunda (copy.deepcopy())

Una copia profunda crea un nuevo objeto compuesto y luego, recursivamente, inserta copias de los objetos que se encuentran en el original.

- Asegura que la copia sea completamente independiente del original, incluso si contiene objetos mutables anidados.
- Es la opción más segura cuando trabajas con estructuras de datos complejas o jerárquicas, pero es más costosa en términos de rendimiento.

```
import copy

# Lista con un objeto mutable anidado (la lista interior)
lista_original = ["A", [1, 2], "C"]

# --- 1. Copia Superficial ---
lista_superficial = copy.copy(lista_original)

# Modificamos el objeto mutable anidado en la copia
lista_superficial[1].append(99)

print("--- Copia Superficial ---")
print(f"Original: {lista_original}")      # ¡El original se modificó! -> ['A',
[1, 2, 99], 'C']
print(f"Superficial: {lista_superficial}") # Muestra ['A', [1, 2, 99], 'C']

# --- 2. Copia Profunda ---
# Reiniciamos la lista original para el segundo ejemplo
lista_original = ["A", [1, 2], "C"]
lista_profunda = copy.deepcopy(lista_original)

# Modificamos el objeto mutable anidado en la copia profunda
lista_profunda[1].append(88)

print("\n--- Copia Profunda ---")
print(f"Original: {lista_original}")      # El original permanece intacto ->
['A', [1, 2], 'C']
print(f"Profunda: {lista_profunda}")      # Muestra ['A', [1, 2, 88], 'C']
```

Usando clases:

```
import copy

# Clase para el objeto anidado mutable
class Parte:
    def __init__(self, valor):
        self.valor = valor
    def __repr__(self):
        return f"Parte(valor='{self.valor}')"

# Clase contenedora (el objeto principal)
class Contenedor:
    def __init__(self, id_unico, parte_mutable):
        self.id = id_unico
        self.parte = parte_mutable # Objeto mutable anidado (Parte)
```

```
    def __repr__(self):
        return f"Contenedor(id={self.id}, parte={self.parte})"


# 1. Creamos el objeto principal y su parte mutable
parte_original = Parte(valor="Inicial")
objeto_original = Contenedor(id_unico=1, parte_mutable=parte_original)

# --- 1. Copia Superficial ---
objeto_superficial = copy.copy(objeto_original)

# Modificamos el objeto mutable anidado a través de la copia superficial
objeto_superficial.parte.valor = "Modificado_Superficial"

print("--- Copia Superficial ---")
# El objeto original.parte TAMBIÉN cambia porque ambas copias apuntan a la
misma instancia de 'Parte'
print(f"Original: {objeto_original}")
print(f"Superficial: {objeto_superficial}")

# --- 2. Copia Profunda ---
# 2. Reiniciamos los objetos para el segundo ejemplo
parte_original_2 = Parte(valor="Inicial")
objeto_original_2 = Contenedor(id_unico=2, parte_mutable=parte_original_2)

objeto_profundo = copy.deepcopy(objeto_original_2)

# Modificamos el objeto mutable anidado en la copia profunda
objeto_profundo.parte.valor = "Modificado_Profundo"

print("\n--- Copia Profunda ---")
# El objeto original_2 permanece INTACTO
print(f"Original: {objeto_original_2}")
print(f"Profunda: {objeto_profundo}")
```

14. Type hints y typing aplicados a POO (opcional pero recomendado)

Python es un lenguaje de tipado dinámico, lo que significa que no se requiere declarar el tipo de una variable. Sin embargo, los Type Hints (sugerencias de tipo), introducidos en Python 3.5 y disponibles en el módulo typing, permiten añadir información de tipo para mejorar la legibilidad, facilitar la refactorización y, lo más importante, habilitar la verificación de errores estáticos mediante herramientas como *Pylance.*
Las sugerencias de tipo son *sugerencias*; no detienen el código si se proporciona un tipo incorrecto en tiempo de ejecución, pero son una práctica estándar en el desarrollo moderno de Python.

Type Hints en Clases y Métodos
Se aplican a atributos de instancia, parámetros de métodos y el valor de retorno.

```
from typing import List, Optional
# Para tipos complejos como listas y opcionales
class Empleado:
```

```python
    # Type hints para los atributos de instancia (Python 3.6+)
    nombre: str
    salario: float
    proyectos: List[str]

    def __init__(self, nombre: str, salario: float, proyectos: List[str]):
        """
        Type hints para los parámetros del constructor.
        """
        self.nombre = nombre
        self.salario = salario
        self.proyectos = proyectos

    def calcular_impuesto(self, tasa: float = 0.20) -> float:
        """
        Type hint para el valor de retorno (-> float).
        """
        return self.salario * tasa

    def obtener_lider(self) -> Optional['Empleado']:
        """
        Optional: indica que el método puede devolver un Empleado o None.
        Comillas: Se usa 'Empleado'
        """
        return None

# Uso y verificación (los Type Checkers verificarían estos errores)
e1 = Empleado("Ana García", 50000.0, ["Proyecto A", "Proyecto B"])
impuesto = e1.calcular_impuesto() # El checker sabe que 'impuesto' es float
# e2 = Empleado(12345, 50000.0, []) # Esto daría un error estático
```

Type Hints y Herencia

Los *type hints* son esenciales para definir Interfaces claras, incluso si no usas herencia explícita.

1. Polimorfismo con Tipos Abstractos (Abstract Base Classes - ABC): El módulo abc permite definir clases base con métodos abstractos que obligan a las clases hijas a implementarlos. Esto asegura que la clase hija satisfaga una "interfaz".
2. Protocolos (Python 3.8+): El módulo typing.Protocol es la forma moderna de implementar Duck Typing con seguridad de tipos. Si un objeto tiene los métodos y atributos definidos en un Protocolo, cumple el tipo, sin necesidad de herencia formal.

```python
from typing import Protocol, runtime_checkable

# 1. Definimos un Protocolo (la interfaz que deben cumplir los objetos)
@runtime_checkable
class PuedeHacerSonido(Protocol):
    def hacer_sonido(self) -> str: ...
```

```python
# 2. Clases que implementan (sin heredar) el Protocolo
class Perro:
    def hacer_sonido(self) -> str:
        return "Guau!"

class Campana:
    def hacer_sonido(self) -> str:
        return "Ring!"

# 3. Función que espera el Protocolo
def alertar(objeto: PuedeHacerSonido):
    # El Type Checker garantiza que objeto.hacer_sonido() existe
    print(f"Alerta: {objeto.hacer_sonido()}")

# Uso
perro = Perro()
campana = Campana()

alertar(perro)    # Funciona
alertar(campana)  # Funciona (ambos cumplen el Protocolo)

# Verificación de Duck Typing en tiempo de ejecución
print(isinstance(perro, PuedeHacerSonido)) # True
```

15. Buenas prácticas (PEP 8 + POO)

- Clases en CamelCase: MiClase.
- Métodos/atributos en snake_case: mi_metodo.
- Docstrings en clases y métodos (qué hace, parámetros, retorno).
- Evita herencias profundas. Mejor composición si no es “es un/a”.
- Expón una API clara: usa @property para validación, oculta internos con _.
- Separa responsabilidades: una clase, un motivo de cambio.