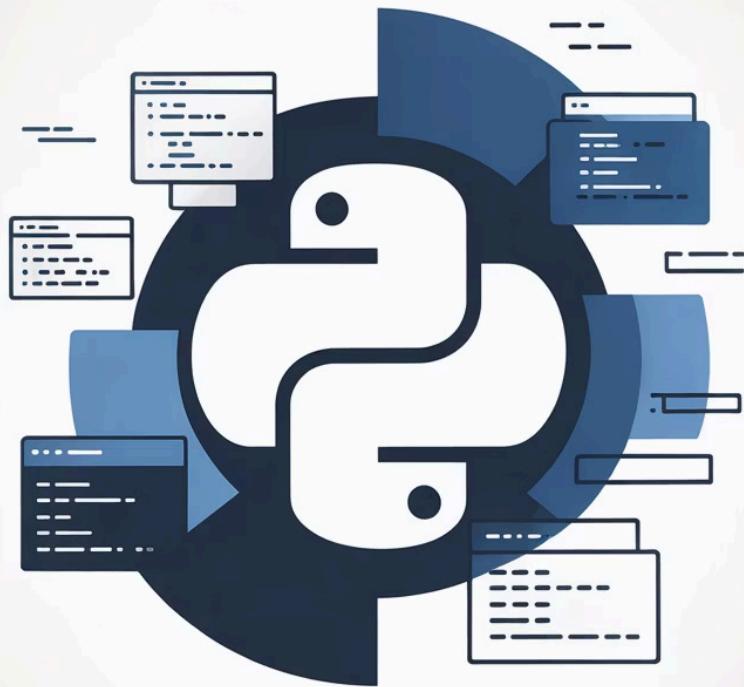


Introducción al Backend y Arquitectura Cliente-Servidor

Curso: Desarrollo Backend con Django



Programación Orientada a Objetos en Python

¿Qué aprenderemos hoy?



1 Fundamentos de POO

Conceptos básicos y ventajas en el desarrollo backend



2 Clases y Objetos

Estructura, atributos y métodos



3 Conceptos Avanzados

Herencia, polimorfismo y encapsulamiento



4 Métodos Especiales

`__init__`, `__str__` y otros métodos "mágicos"



5 Ejemplo Práctico

Creación de un sistema de usuarios paso a paso

¿Qué es la Programación Orientada a Objetos?

Es un **paradigma de programación** que organiza el código en unidades llamadas **objetos**, que son instancias de **clases**.

Permite modelar entidades del mundo real y sus interacciones de forma natural en nuestro código.



¿Por qué usar POO en desarrollo backend?

Organización

Estructura el código de manera lógica, agrupando datos y comportamientos relacionados

Reutilización

Permite crear componentes que pueden usarse en diferentes partes del sistema

Mantenibilidad

Facilita el mantenimiento al aislar los cambios en componentes específicos

Escalabilidad

Permite que el sistema crezca de manera ordenada, añadiendo nuevas funcionalidades sin romper lo existente

POO en el ecosistema Backend

Los frameworks más populares para desarrollo backend en Python como **Django** y **Flask** aprovechan intensivamente la POO para:

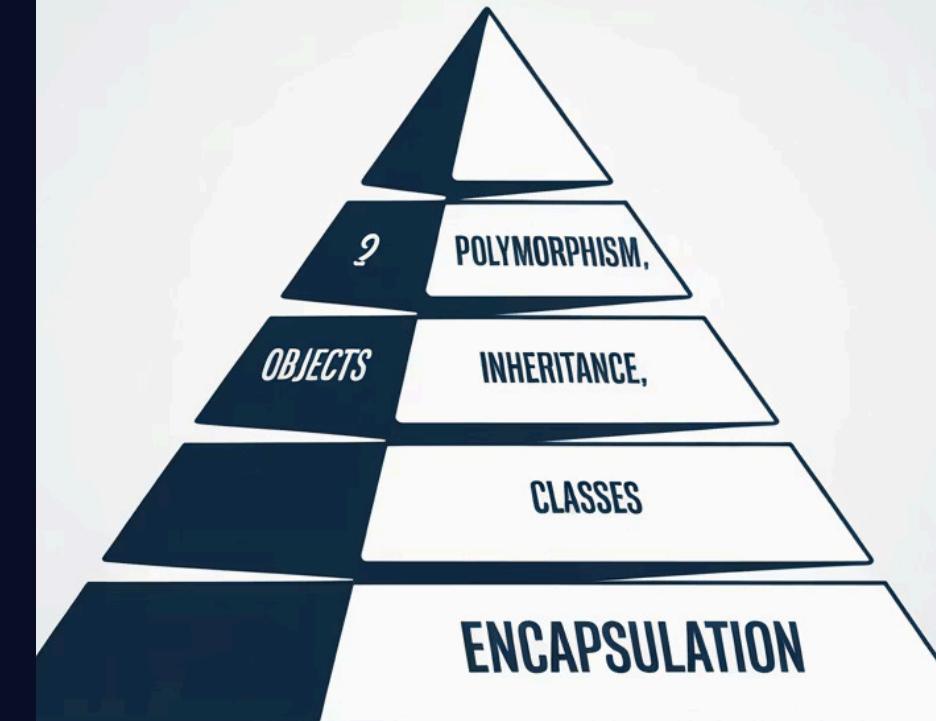
- Definir modelos de datos
- Construir controladores y vistas
- Implementar middlewares y servicios
- Crear extensiones y plugins

Entender la POO es **fundamental** para ser un buen desarrollador backend en Python.



Conceptos Clave de la POO

1. Clases y Objetos
2. Atributos y Métodos
3. Herencia
4. Polimorfismo
5. Encapsulamiento



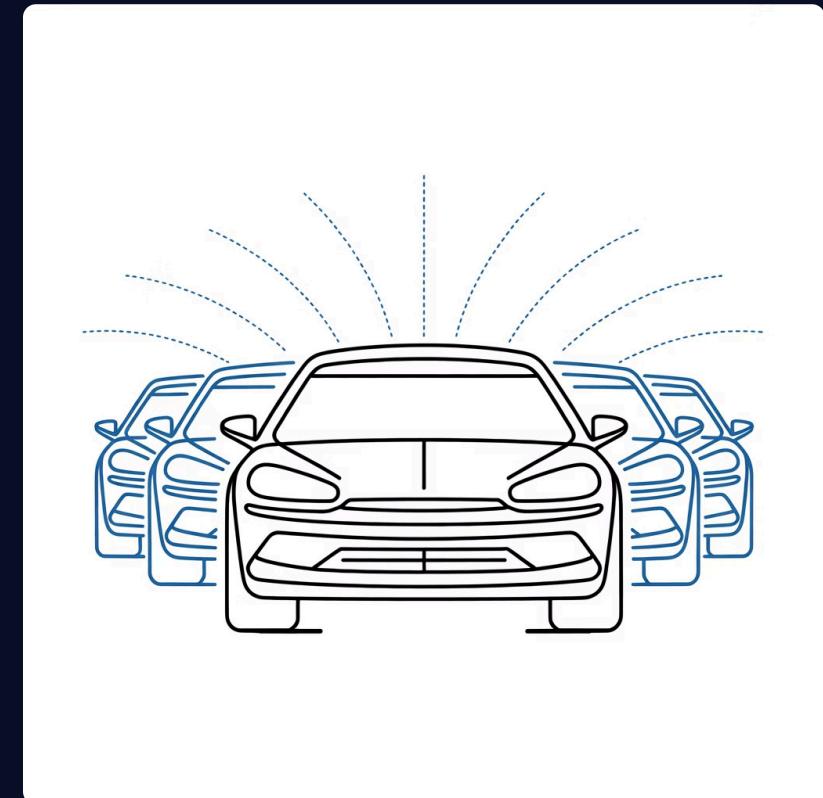
Clases: El molde para crear objetos

Una **clase** es como un plano o molde que define:

- La **estructura** (atributos)
- El **comportamiento** (métodos)

```
class Automovil:  
    # Atributos y métodos  
    # que definen un automóvil
```

Las clases son la unidad fundamental de la POO.



Sintaxis básica de una clase en Python

```
class NombreDeLaClase:  
    # Atributos de clase (compartidos por todas las instancias)  
    atributo_clase = valor  
  
    # Método inicializador  
    def __init__(self, parametro1, parametro2):  
        # Atributos de instancia (específicos de cada objeto)  
        self.atributo1 = parametro1  
        self.atributo2 = parametro2  
  
    # Métodos de la clase  
    def metodo(self, parametros):  
        # Código que implementa el comportamiento  
        return resultado
```

En Python, las clases se definen con la palabra clave `class` seguida del nombre (por convención en PascalCase).

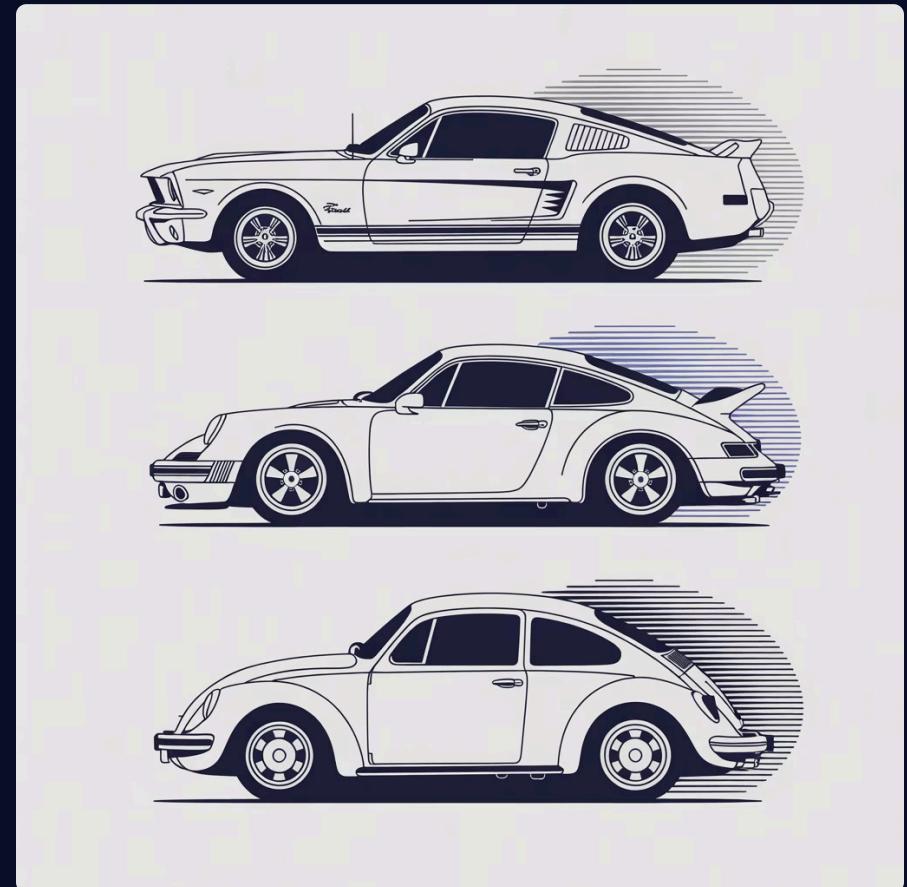
Objetos: Instancias de una clase

Un **objeto** es una instancia concreta de una clase.

Representa una entidad específica con:

- Sus propios **valores** para los atributos
- La capacidad de ejecutar los **métodos** definidos en su clase

```
mi_auto = Automovil("Toyota", "Corolla", 2022)  
tu_auto = Automovil("Mazda", "3", 2021)
```



Atributos: Las características de los objetos



Atributos de Clase

Compartidos por todas las instancias de la clase

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



Atributos de Instancia

Específicos para cada objeto creado

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

Los atributos pueden ser de cualquier tipo: números, strings, listas, diccionarios, e incluso otros objetos.

Veamos un ejemplo completo

```
class Estudiante:  
    # Atributo de clase  
    universidad = "Universidad de los Andes"  
  
    # Método inicializador con atributos de instancia  
    def __init__(self, nombre, codigo, semestre):  
        self.nombre = nombre  
        self.codigo = codigo  
        self.semestre = semestre  
        self.materias = [] # Lista vacía por defecto  
  
    # Creando objetos (instancias) de la clase Estudiante  
estudiante1 = Estudiante("Ana García", "20211034567", 3)  
estudiante2 = Estudiante("Carlos López", "20221045678", 1)  
  
    # Accediendo a los atributos  
print(estudiante1.nombre) # Ana García  
print(estudiante1.universidad) # Universidad de los Andes  
print(estudiante2.semestre) # 1
```

Métodos: El comportamiento de los objetos

Los **métodos** son funciones definidas dentro de una clase que representan las acciones que pueden realizar los objetos.

```
class Estudiante:  
    universidad = "Universidad de los Andes"  
  
    def __init__(self, nombre, codigo, semestre):  
        self.nombre = nombre  
        self.codigo = codigo  
        self.semestre = semestre  
        self.materias = []  
  
    # Método para matricular una materia  
    def matricular_materia(self, materia):  
        self.materias.append(materia)  
        print(f"{self.nombre} ha matriculado {materia}")  
  
    # Método para mostrar información del estudiante  
    def mostrar_info(self):  
        return f"Estudiante: {self.nombre}, Código: {self.codigo}, Semestre: {self.semestre}"
```

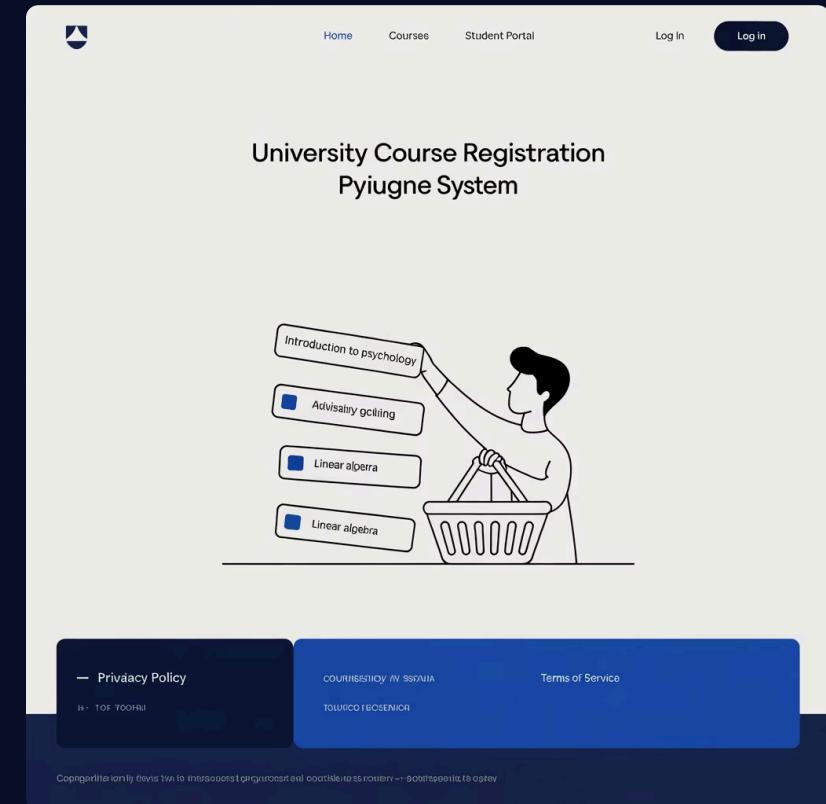
Llamando a los métodos

```
estudiante1 = Estudiante("Ana García", "20211034567", 3)
```

```
# Llamamos al método matricular_materia  
estudiante1.matricular_materia("Programación Orientada a Objetos")  
estudiante1.matricular_materia("Bases de Datos")
```

```
# Llamamos al método mostrar_info  
info = estudiante1.mostrar_info()  
print(info)
```

```
# Salida:  
# Ana García ha matriculado Programación Orientada a Objetos  
# Ana García ha matriculado Bases de Datos  
# Estudiante: Ana García, Código: 20211034567, Semestre: 3
```



Métodos Especiales en Python

También conocidos como "**métodos mágicos**" o "**dunder methods**"

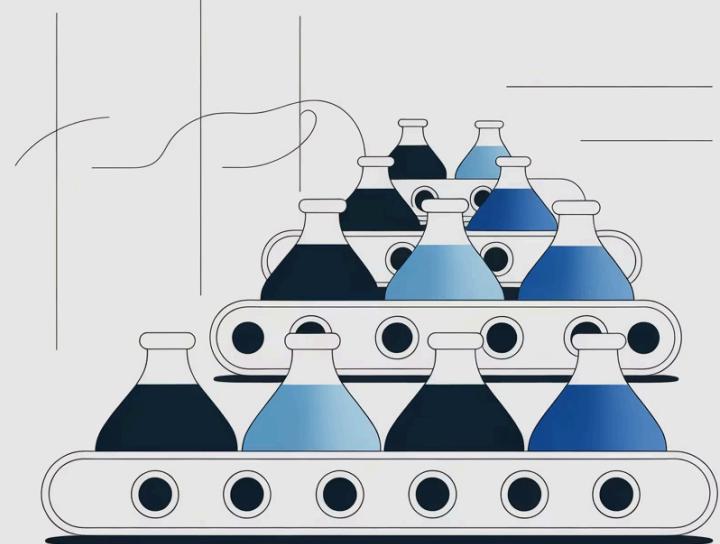


El método `__init__`

El método `__init__` es un método especial que:

- Se ejecuta automáticamente cuando se crea un nuevo objeto
- Inicializa los atributos del objeto
- Es similar a un constructor en otros lenguajes

```
def __init__(self, nombre, edad):  
    self.nombre = nombre  
    self.edad = edad  
    self.activo = True # Valor predeterminado
```



El parámetro self

El parámetro `self` es una referencia a la instancia actual del objeto:

- Es siempre el primer parámetro en los métodos de instancia
- Permite acceder a los atributos y métodos del objeto desde dentro de la clase
- Es similar a `this` en otros lenguajes de programación

```
class Persona:  
    def __init__(self, nombre):  
        self.nombre = nombre # Usamos self para asignar el atributo a esta instancia  
  
    def saludar(self):  
        # Usamos self para acceder al atributo nombre de esta instancia  
        return f"Hola, mi nombre es {self.nombre}"
```

Python pasa automáticamente la referencia al objeto cuando llamamos a un método: `persona.saludar()` se traduce a `Persona.saludar(persona)`

El método `__str__`

El método `__str__`:

- Define la representación en string de un objeto
- Se llama automáticamente cuando usamos `print(objeto)` o `str(objeto)`
- Debe devolver una cadena (string)

```
class Producto:  
    def __init__(self, nombre, precio):  
        self.nombre = nombre  
        self.precio = precio  
  
    def __str__(self):  
        return f"{self.nombre} - ${self.precio:.0f} COP"  
  
# Al usar print(), se llama automáticamente a __str__  
producto = Producto("Laptop", 2500000)  
print(producto) # Salida: Laptop - $2,500,000 COP
```

Otros métodos especiales útiles

__repr__

Representación "oficial" del objeto, usada por `repr()`

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

__len__

Define el comportamiento cuando se usa `len(objeto)`

```
def __len__(self):  
    return len(self.elementos)
```

__eq__

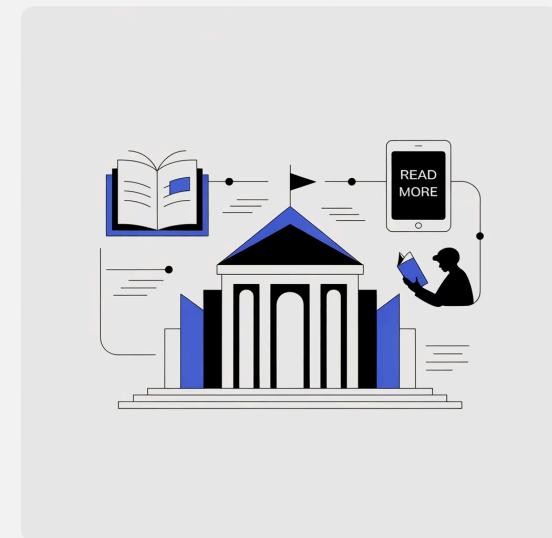
Define la igualdad entre objetos (`==`)

```
def __eq__(self, otro):  
    return self.id == otro.id
```

Estos métodos permiten que nuestros objetos se comporten como los tipos integrados de Python.

Ejemplo Práctico: Sistema de Biblioteca

Vamos a crear un sistema simple para gestionar libros en una biblioteca. Este ejemplo nos ayudará a aplicar los conceptos de la Programación Orientada a Objetos de forma práctica.



Paso 1: Diseñando la clase Libro

Datos iniciales para nuestra clase Libro:

Atributos

- título
- autor
- año_publicacion
- disponible (estado)

Métodos

- prestar()
- devolver()
- mostrar_info()

Paso 2: Implementando la clase Libro

```
class Libro:  
    def __init__(self, titulo, autor, año_publicacion):  
        self.titulo = titulo  
        self.autor = autor  
        self.año_publicacion = año_publicacion  
        self.disponible = True # Por defecto, el libro está disponible al ser creado
```

Paso 3: Añadiendo métodos a la clase Libro

```
def prestar(self):
    if self.disponible:
        self.disponible = False
        print(f"{self.titulo} ha sido prestado.")
    else:
        print(f"{self.titulo} no está disponible para préstamo.")

def devolver(self):
    if not self.disponible:
        self.disponible = True
        print(f"{self.titulo} ha sido devuelto.")
    else:
        print(f"{self.titulo} ya está disponible.")

def mostrar_info(self):
    estado = "Disponible" if self.disponible else "No disponible"
    print(f"Título: {self.titulo}")
    print(f"Autor: {self.autor}")
    print(f"Año de publicación: {self.año_publicacion}")
    print(f"Estado: {estado}")

def __str__(self):
    return f"{self.titulo} por {self.autor} ({self.año_publicacion})"
```

Paso 4: Usando nuestra clase Libro

```
# Crear instancias de la clase Libro
libro1 = Libro("Cien años de soledad", "Gabriel García Márquez", 1967)
libro2 = Libro("1984", "George Orwell", 1949)
libro3 = Libro("Don Quijote de la Mancha", "Miguel de Cervantes", 1605)

print("--- Información inicial de los libros ---")
libro1.mostrar_info()
libro2.mostrar_info()

print("\n--- Prestando libros ---")
libro1.prestar()
libro2.prestar()
libro2.prestar() # Intentar prestar un libro no disponible

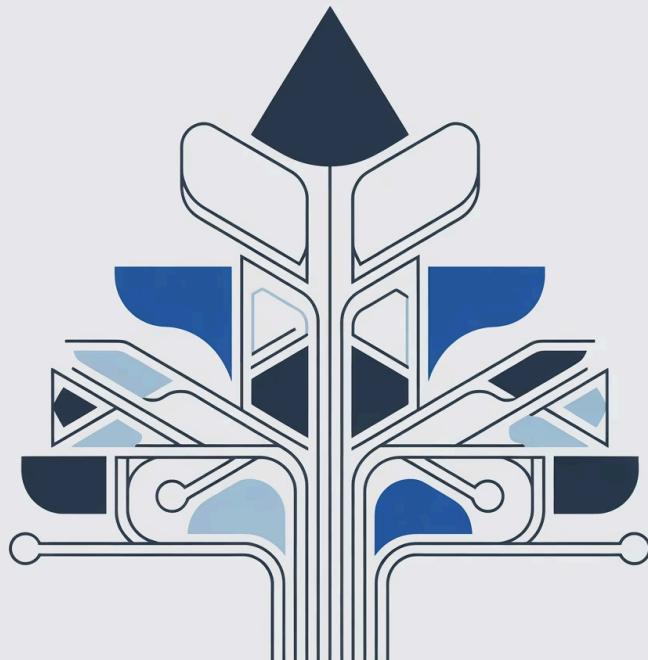
print("\n--- Información de los libros después de prestar ---")
libro1.mostrar_info()
libro2.mostrar_info()

print("\n--- Devolviendo libros ---")
libro1.devolver()
libro3.devolver() # Intentar devolver un libro ya disponible

print("\n--- Información final de los libros ---")
libro1.mostrar_info()
libro3.mostrar_info()
```



Herencia



Es el mecanismo que permite a una clase (subclase) heredar atributos y métodos de otra clase (superclase).

Permite **reutilizar código** y crear jerarquías de clases relacionadas.

Sintaxis de la herencia en Python

```
class ClaseBase:  
    # Atributos y métodos de la clase base  
  
class ClaseDerivada(ClaseBase):  
    # La clase derivada hereda todo de ClaseBase  
    # Y puede añadir nuevos atributos y métodos  
    # O modificar los existentes
```

Python permite la herencia múltiple (heredar de varias clases):

```
class ClaseDerivada(ClaseBase1, ClaseBase2, ClaseBase3):  
    # Hereda de todas las clases base mencionadas  
    pass
```

Se utiliza `pass` cuando necesitamos una clase vacía o como placeholder mientras desarrollamos.

Ejemplo de herencia

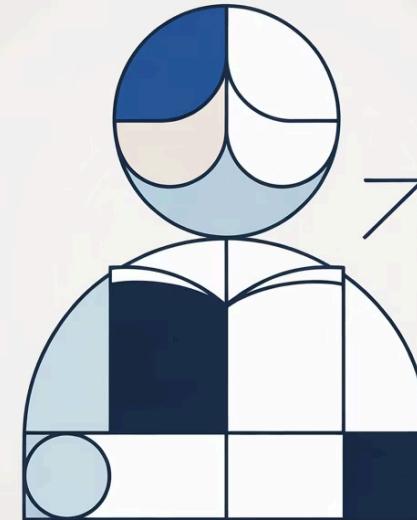
```
class Persona:  
    def __init__(self, nombre, edad):  
        self.nombre = nombre  
        self.edad = edad  
  
    def presentarse(self):  
        return f"Hola, me llamo {self.nombre} y tengo {self.edad} años."  
  
class Estudiante(Persona):  
    def __init__(self, nombre, edad, codigo, carrera):  
        # Llamamos al inicializador de la clase padre  
        super().__init__(nombre, edad)  
        # Añadimos atributos específicos de Estudiante  
        self.codigo = codigo  
        self.carrera = carrera  
  
    def estudiar(self):  
        return f"{self.nombre} está estudiando {self.carrera}."
```

Uso de la herencia

```
estudiante = Estudiante(  
    "Laura Gómez",  
    20,  
    "2023102030",  
    "Ingeniería de Sistemas"  
)
```

```
# Método heredado de Persona  
print(estudiante.presentarse())  
# Salida: Hola, me llamo Laura Gómez  
# y tengo 20 años.
```

```
# Método propio de Estudiante  
print(estudiante.estudiar())  
# Salida: Laura Gómez está estudiando  
# Ingeniería de Sistemas.
```



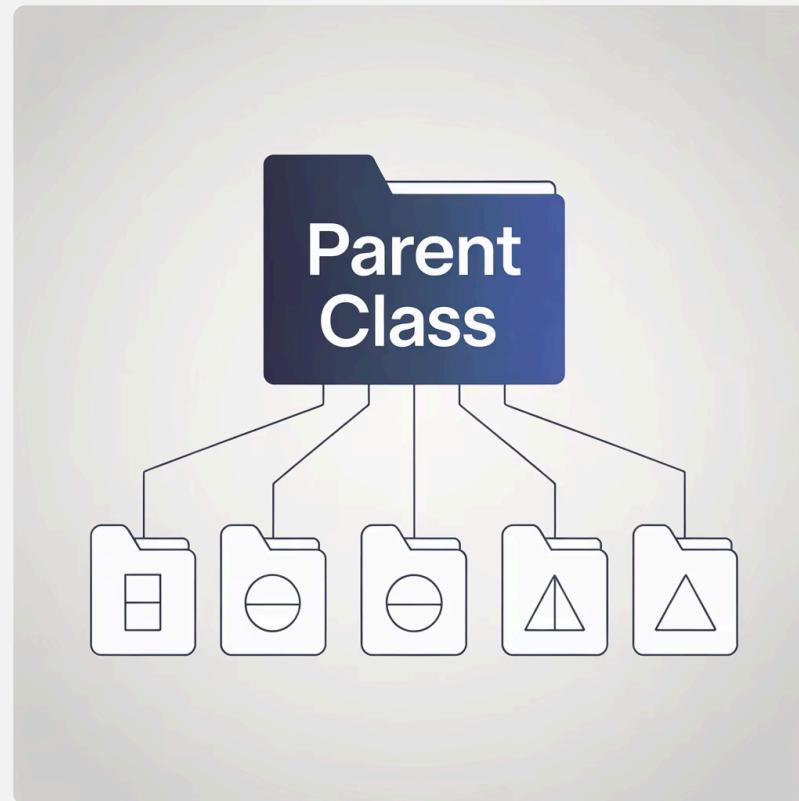
Función super()

La función `super()` nos permite:

- Acceder a métodos y propiedades de la clase padre
- Evitar repetir código
- Mantener la cadena de herencia intacta

```
class EmpleadoTiempoCompleto:  
    def __init__(self, nombre, salario_base):  
        self.nombre = nombre  
        self.salario_base = salario_base  
  
    def calcular_salario(self):  
        return self.salario_base  
  
class EmpleadoConComision(EmpleadoTiempoCompleto):  
    def __init__(self, nombre, salario_base, comisiones):  
        super().__init__(nombre, salario_base) # Llama al __init__ del parente  
        self.comisiones = comisiones  
  
    def calcular_salario(self):  
        # Usa el método del parente y le añade las comisiones  
        return super().calcular_salario() + self.comisiones
```

¿Por qué la herencia reduce código repetido?



Beneficios de la herencia:

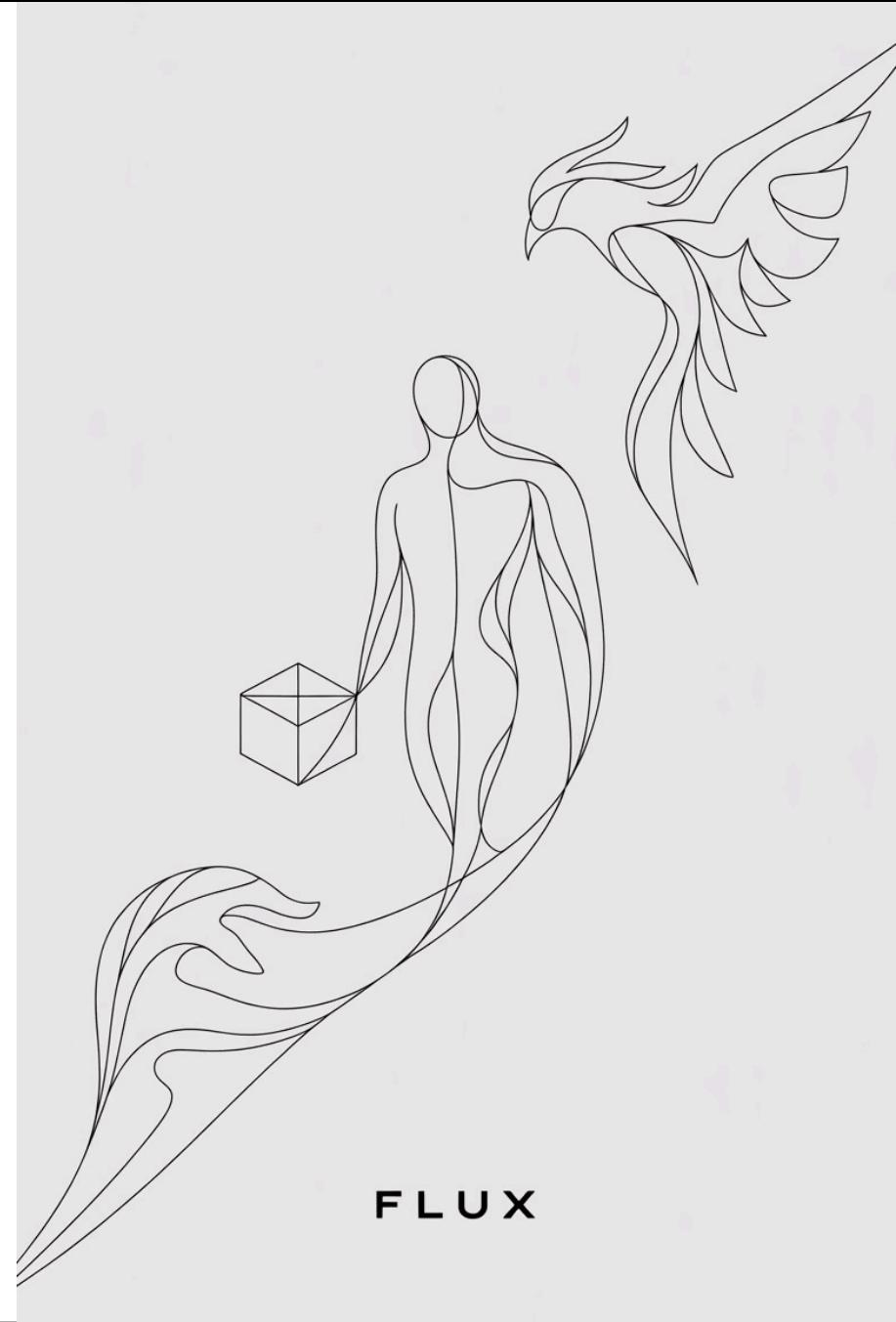
- Centraliza el código común en la clase base
 - Cambios en la clase base se propagan a todas las subclases
 - Las subclases solo implementan lo específico
 - Promueve el principio DRY (Don't Repeat Yourself)
- "La herencia permite escribir una vez y reutilizar muchas veces."

Polimorfismo

En el contexto de la programación orientada a objetos (POO), el polimorfismo es la capacidad de diferentes objetos para responder a un mismo mensaje de maneras diferentes, adaptándose a su propio tipo o clase.

Conceptos clave:

- Permite tratar objetos de diferentes clases de manera uniforme.
- Promueve la flexibilidad y extensibilidad del código.
- Puede implementarse a través de la sobreescritura de métodos (runtime) o la sobrecarga (compile-time).
- Facilita la creación de interfaces comunes para diversas implementaciones.



¿Qué es el polimorfismo?

El **polimorfismo** permite que objetos de diferentes clases sean tratados a través de una interfaz común.

En Python, esto ocurre naturalmente debido a su tipado dinámico.

Principio: si un objeto tiene los métodos/atributos que estamos usando, funcionará independientemente de su clase específica.



Ejemplo de polimorfismo

```
class Animal:  
    def hacer_sonido(self):  
        pass # Método base que será sobreescrito  
  
class Perro(Animal):  
    def hacer_sonido(self):  
        return "¡Guau guau!"  
  
class Gato(Animal):  
    def hacer_sonido(self):  
        return "¡Miau!"  
  
class Vaca(Animal):  
    def hacer_sonido(self):  
        return "¡Muuu!"  
  
# Lista de animales de diferentes clases  
animales = [Perro(), Gato(), Vaca()]  
  
# Polimorfismo en acción  
for animal in animales:  
    print(animal.hacer_sonido()) # Cada animal hace su sonido específico
```

Duck Typing en Python

"Si camina como un pato y hace cuac como un pato, entonces es un pato."

En Python, el polimorfismo se basa en el comportamiento de los objetos, no en su tipo formal.

```
def hacer_nadar(objeto):
    # No verifica el tipo, solo si tiene el método nadar()
    objeto.nadar()

class Pato:
    def nadar(self):
        print("El pato está nadando")

class Robot:
    def nadar(self):
        print("El robot está nadando con propulsores")

# Ambos funcionan aunque son clases totalmente diferentes
hacer_nadar(Pato()) # El pato está nadando
hacer_nadar(Robot()) # El robot está nadando con propulsores
```



Encapsulamiento



Es el principio de **ocultar los detalles internos** de un objeto y exponer solo lo necesario.

Permite controlar el acceso a los datos y proteger la integridad del objeto.

Convenciones de encapsulamiento en Python

Público (sin prefijo)

Accesible desde cualquier parte del código

```
self.nombre = nombre #  
Atributo público
```

Protegido (prefijo _)

Por convención, no debería accederse desde fuera de la clase o sus subclases

```
self._saldo = 1000 # Atributo  
protegido
```

Privado (prefijo __)

Python oculta estos atributos mediante name mangling

```
self.__pin = "1234" # Atributo  
privado
```

Python sigue la filosofía de "somos todos adultos responsables" y no impone restricciones estrictas, sino que usa convenciones.

Ejemplo de encapsulamiento

```
class CuentaBancaria:  
    def __init__(self, titular, saldo_inicial):  
        self.titular = titular # Público - accesible para todos  
        self._saldo = saldo_inicial # Protegido - solo para uso interno  
        self.__pin = "0000" # Privado - muy restringido  
  
    def depositar(self, cantidad):  
        if cantidad > 0:  
            self._saldo += cantidad  
            return True  
        return False  
  
    def retirar(self, cantidad, pin):  
        if pin == self.__pin and cantidad > 0 and cantidad <= self._saldo:  
            self._saldo -= cantidad  
            return True  
        return False  
  
    def obtener_saldo(self):  
        return self._saldo  
  
    def cambiar_pin(self, pin_actual, pin_nuevo):  
        if pin_actual == self.__pin:  
            self.__pin = pin_nuevo  
            return True  
        return False
```

Uso de la clase CuentaBancaria



```
cuenta = CuentaBancaria("Ana López", 1000000)

# Acceso a atributo público
print(cuenta.titular) # Ana López

# Interfaz pública para interactuar con la cuenta
cuenta.depositar(500000)
print(cuenta.obtener_saldo()) # 1500000

exito = cuenta.retirar(200000, "0000")
if exito:
    print("Retiro exitoso")
else:
    print("Retiro fallido")

# Intento de acceso directo (no recomendado)
print(cuenta._saldo) # Funciona, pero viola encapsulamiento

# Intento de acceso a atributo privado
# print(cuenta.__pin) # Error!
# El nombre real es _CuentaBancaria__pin
```

Properties: Una mejor forma de encapsulamiento

Las **properties** permiten exponer atributos como si fueran públicos pero controlar su acceso:

```
class Persona:  
    def __init__(self, nombre, edad):  
        self._nombre = nombre  
        self._edad = edad  
  
    @property  
    def edad(self):  
        return self._edad  
  
    @edad.setter  
    def edad(self, nueva_edad):  
        if nueva_edad > 0 and nueva_edad < 120:  
            self._edad = nueva_edad  
        else:  
            raise ValueError("Edad inválida")  
  
    @property  
    def nombre(self):  
        return self._nombre  
  
# Uso  
persona = Persona("Carlos", 25)  
print(persona.edad) # Usa el getter -> 25  
persona.edad = 30 # Usa el setter  
# persona.edad = -5 # Lanza una excepción
```

Mejorando nuestro Sistema de Biblioteca con POO

Ahora que conocemos todos los conceptos de Programación Orientada a Objetos (POO), vamos a mejorar nuestro sistema de biblioteca aplicando:



Herencia

Crearemos `LibroFisico` y `LibroDigital` que heredarán de una clase base `Libro`.



Encapsulamiento

Protegeremos atributos clave como `ISBN` y `saldo` utilizando propiedades.



Polimorfismo

Permitiremos que diferentes tipos de libros (físicos o digitales) se comporten de manera uniforme a través de una interfaz común.



Métodos Especiales

Añadiremos el método `__eq__` para comparar libros de forma eficiente basándonos en su `ISBN`.



Clase base Libro mejorada

```
class Libro:
    def __init__(self, titulo, autor, año_publicacion, isbn):
        self.titulo = titulo
        self.autor = autor
        self.año_publicacion = año_publicacion
        self._isbn = isbn # Protegido
        self._disponible = True # Protegido
        self._veces_prestado = 0 # Protegido

    @property
    def disponible(self):
        return self._disponible

    @property
    def isbn(self):
        return self._isbn

    def prestar(self):
        if self._disponible:
            self._disponible = False
            self._veces_prestado += 1
            return True
        return False

    def devolver(self):
        if not self._disponible:
            self._disponible = True
            return True
        return False

    def __str__(self):
        return f'{self.titulo} por {self.autor}'

    def __eq__(self, otro):
        return isinstance(otro, Libro) and self._isbn == otro._isbn
```

Subclases: LibroFisico y LibroDigital

```
class LibroFisico(Libro):
    def __init__(self, titulo, autor, año_publicacion, isbn, ubicacion, estado_fisico="Bueno"):
        super().__init__(titulo, autor, año_publicacion, isbn)
        self.ubicacion = ubicacion # Ej: "Estante A-3"
        self.estado_fisico = estado_fisico

    def mostrar_info(self):
        estado = "Disponible" if self._disponible else "Prestado"
        return f"{self} - Ubicación: {self.ubicacion}, Estado: {estado}, Condición: {self.estado_fisico}"

class LibroDigital(Libro):
    def __init__(self, titulo, autor, año_publicacion, isbn, formato, tamaño_mb):
        super().__init__(titulo, autor, año_publicacion, isbn)
        self.formato = formato # Ej: "PDF", "EPUB"
        self.tamaño_mb = tamaño_mb
        self._descargas_simultaneas = 0
        self._max_descargas = 3

    def prestar(self):
        if self._descargas_simultaneas < self._max_descargas:
            self._descargas_simultaneas += 1
            self._veces_prestado += 1
            return True
        return False

    def devolver(self):
        if self._descargas_simultaneas > 0:
            self._descargas_simultaneas -= 1
            return True
        return False

    def mostrar_info(self):
        return f"{self} - Formato: {self.formato}, Tamaño: {self.tamaño_mb}MB, Descargas: {self._descargas_simultaneas}/{self._max_descargas}"
```



Sistema mejorado en acción

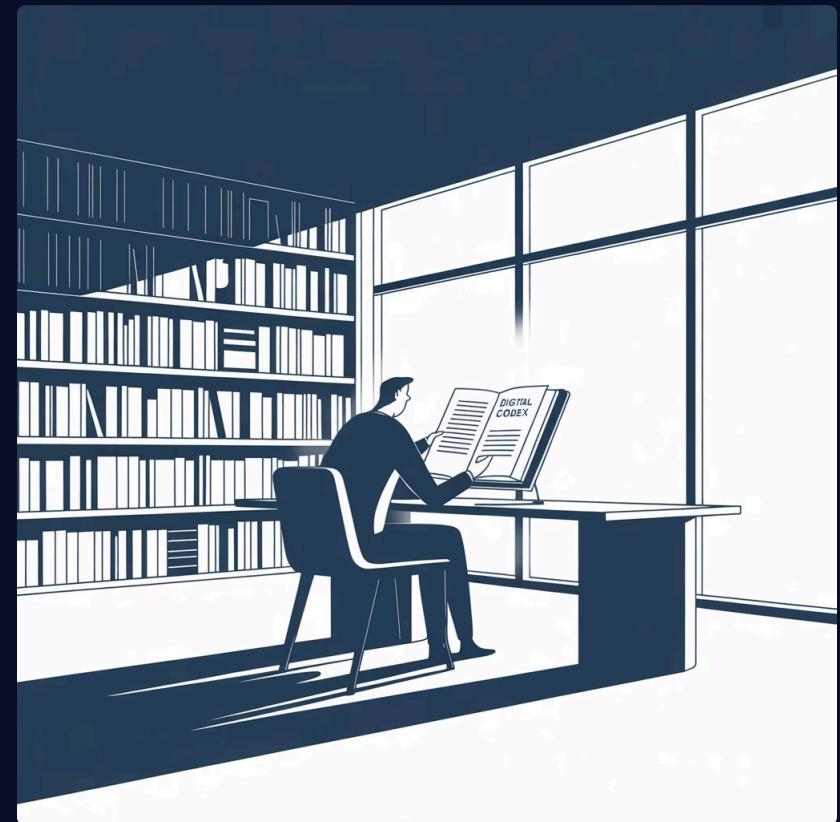
```
# Crear diferentes tipos de libros
libro_fisico = LibroFisico("Cien años de soledad", "Gabriel García Márquez", 1967, "978-84-376-0494-7",
                           "Estante A-3")
libro_digital = LibroDigital("1984", "George Orwell", 1949, "978-0-452-28423-4", "PDF", 2.5)

# Lista de libros (polimorfismo)
biblioteca = [libro_fisico, libro_digital]

# Procesar todos los libros de manera uniforme
for libro in biblioteca:
    print(libreria.mostrar_info())

# Prestar libro
if libro.prestar():
    print(f"✓ {libro} prestado exitosamente")
else:
    print(f" No se pudo prestar {libro}")

# Comparar libros usando __eq__
libro_fisico2 = LibroFisico("Otra edición", "Otro autor", 2000, "978-84-376-0494-7", "Estante B-1")
print(f"¿Son el mismo libro? {libro_fisico == libro_fisico2}") # True (mismo ISBN)
```



Buenas prácticas en POO con Python



Nombres descriptivos

Usa nombres claros para clases (PascalCase) y métodos/atributos (snake_case)



Principio de responsabilidad única

Cada clase debe tener una sola responsabilidad bien definida



Documentación

Usa docstrings para documentar tus clases y métodos



Getters y setters

Usa @property en lugar de métodos get/set tradicionales



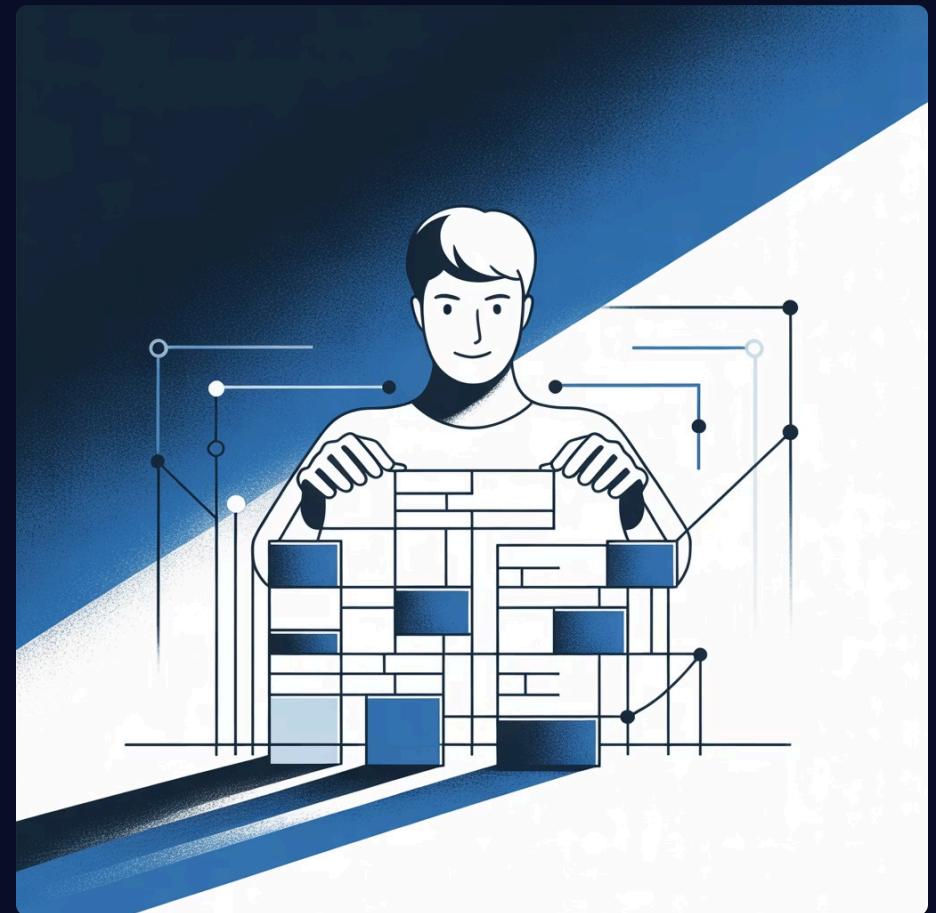
Pruebas unitarias

Escribe pruebas para verificar el comportamiento de tus clases

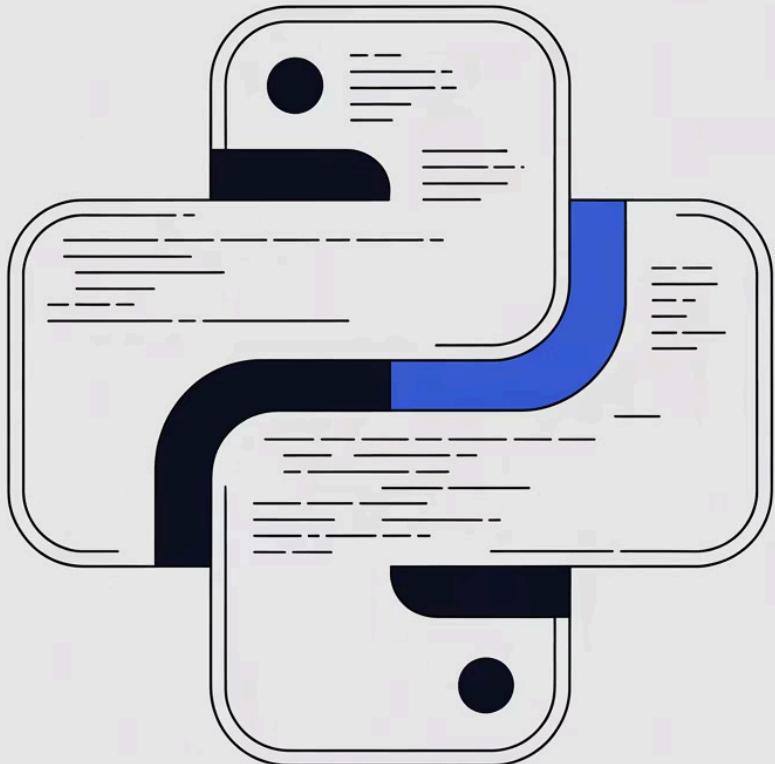
Reflexión final: ¿Por qué usar POO?

La POO nos permite crear sistemas complejos que:

- Son más **mantenibles** a largo plazo
- Tienen código más **reutilizable**
- Son más **fáciles de entender** para otros desarrolladores
- Modelan mejor el **mundo real** y sus relaciones
- Son más **escalables** y adaptables al cambio



Python



¡Gracias!

¿Preguntas?

#EDCOUNIANDES

<https://educacioncontinua.uniandes.edu.co/>

Contacto: educacion.continua@uniandes.edu.co

© - Derechos Reservados: La presente obra, y en general todos sus contenidos, se encuentran protegidos por las normas internacionales y nacionales vigentes sobre propiedad Intelectual, por lo tanto su utilización parcial o total, reproducción, comunicación pública, transformación, distribución, alquiler, préstamo público e importación, total o parcial, en todo o en parte, en formato impreso o digital y en cualquier formato conocido o por conocer, se encuentran prohibidos, y solo serán lícitos en la medida en que se cuente con la autorización previa y expresa por escrito de la Universidad de los Andes.