Tema 1
Introducción a la POO

Índice

Profesor: Salvador Martínez Bolinches
Centro: IES Font de Sant Lluís
Año: 2025

1.1 Introducción y contexto histórico

Origen de la Programación Orientada a Objetos

La **Programación Orientada a Objetos (POO)** surgió en la década de 1960 como respuesta a las limitaciones de los lenguajes procedimentales tradicionales. Uno de los primeros lenguajes en implementar conceptos similares fue **Simula 67**, desarrollado en Noruega por Ole-Johan Dahl y Kristen Nygaard, orientado inicialmente a la simulación de procesos.

Posteriormente, en la década de 1970, el lenguaje **Smalltalk** refinó el paradigma, estableciendo principios como:

- **Todo es un objeto**.
- **La comunicación entre objetos se hace mediante mensajes**.
- **La herencia permite la reutilización y extensión de código**.

Python, aunque no es puramente orientado a objetos, adopta gran parte de estos conceptos y los combina con características procedimentales y funcionales.

La Programación Orientado a Objetos (POO) se basa en la agrupación de objetos de distintas clases que interactúan entre sí y que, en conjunto, consiguen que un programa cumpla su propósito. En Python cualquier elemento del lenguaje pertenece a una clase y todas las clases tienen el mismo rango y se utilizan del mismo modo.

1.2 Conceptos clave y motivación

Antes de la POO, los programas grandes se desarrollaban siguiendo una **estructura lineal o modular**, pero con el crecimiento de la complejidad, aparecían problemas como:

- Dificultad para **mantener** el código.
- **Duplicación** de lógica en distintas partes del programa.
- Falta de **abstracción** de los datos y comportamientos.

La POO resuelve estos problemas al permitir que el **código se organice en torno a entidades (objetos)** que combinan datos y comportamientos, promoviendo **modularidad, reutilización y extensibilidad**.

Ejemplo ilustrativo: Procedimental vs Orientado a Objetos

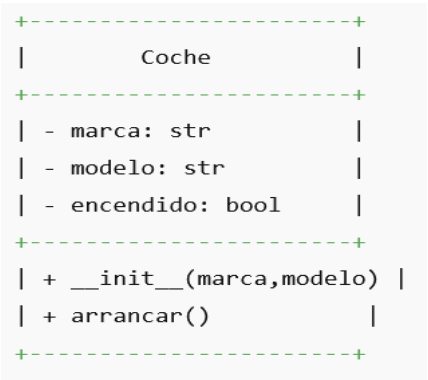
Enfoque procedimental:

```python
# Gestión de un coche usando funciones y diccionarios
def crear_coche(marca, modelo):
    return {"marca": marca, "modelo": modelo, "encendido": False}

def arrancar_coche(coche):
    coche["encendido"] = True
    print(f"El {coche['marca']} {coche['modelo']} ha arrancado.")

mi_coche = crear_coche("Toyota", "Corolla")
arrancar_coche(mi_coche)
```

Enfoque orientado a objetos:

```python
# Gestión de un coche usando clases y objetos
class Coche:
    def __init__(self, marca: str, modelo: str):
        self.marca = marca
        self.modelo = modelo
        self.encendido = False

    def arrancar(self):
        self.encendido = True
        print(f"El {self.marca} {self.modelo} ha arrancado.")

mi_coche = Coche("Toyota", "Corolla")
mi_coche.arrancar()
```

Diagrama UML



1.3 Ventajas y limitaciones

Ventajas de la Programación Orientada a Objetos

1. Modularidad

El código se divide en clases y objetos independientes, lo que facilita el mantenimiento.

Ejemplo:

```
class Motor:
    def encender(self):
        print("Motor encendido.")

class Coche:
    def __init__(self):
        self.motor = Motor()

    def arrancar(self):
        self.motor.encender()

mi_coche = Coche()
mi_coche.arrancar()
```

Aquí, Motor y Coche son módulos independientes, lo que facilita reemplazar o modificar uno sin afectar al otro.

2. **Reutilización de código**

Una vez definida una clase, puede ser utilizada en distintos programas o proyectos sin necesidad de reescribirla.

3. **Escalabilidad**

La POO permite añadir nuevas funcionalidades sin modificar el código existente, reduciendo riesgos.

4. **Mantenibilidad**

Localizar y corregir errores es más sencillo al estar el código organizado en entidades bien definidas.

5. **Abstracción**

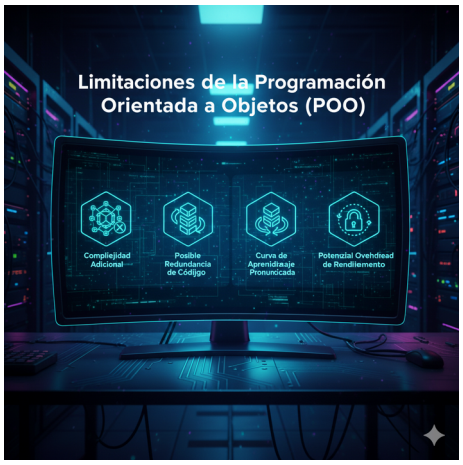
Permite exponer solo lo necesario y ocultar los detalles internos de implementación. Ejemplo:

```python
class CuentaBancaria:
    def __init__(self, saldo):
        self.__saldo = saldo  # atributo privado

    def depositar(self, cantidad):
        self.__saldo += cantidad

    def obtener_saldo(self):
        return self.__saldo
```

6. Polimorfismo

Diferentes clases pueden implementar un mismo método de distintas formas.

```python
class Ave:
    def volar(self):
        pass

class Aguila(Ave):
    def volar(self):
        print("El águila vuela alto.")

class Pinguino(Ave):
    def volar(self):
        print("El pingüino no puede volar.")
```



Limitaciones y desafíos de la POO

1. **Curva de aprendizaje**

   Para programadores procedimentales, la transición a POO requiere cambiar la forma de pensar el diseño del software.

2. **Sobrecarga innecesaria**

   En programas muy simples, usar POO puede añadir complejidad innecesaria.

3. **Consumo de memoria**

   Cada objeto tiene su propio espacio en memoria, lo que puede ser menos eficiente que las estructuras planas.

4. **Mal uso de la herencia**

   Un uso excesivo o inadecuado de la herencia puede generar sistemas difíciles de mantener.

Ejemplo de mala práctica – Herencia mal aplicada:

```python
class Ave:
    def __init__(self):
        self.tipo_ala = "pluma"

class Pez(Ave):  # Error conceptual: un pez no debería heredar de Ave
    def __init__(self):
        super().__init__()
        self.tipo_escama = "fina"
```

Este diseño rompe el principio de **"is-a"** (es-un), fundamental en herencia.

Definición de clase, objeto, atributos y métodos:

- Llamamos clase a la representación abstracta de un concepto. Por ejemplo, “perro”, “número entero” o “servidor web”.
- Las clases se componen de atributos y métodos.
- Un objeto es cada una de las instancias de una clase.
- Los atributos definen las características propias del objeto y modifican su estado. Son datos asociados a las clases y a los objetos creados a partir de ellas.
- Un atributo de clase es compartida por todas las instancias de una clase. Se definen dentro de la clase (después del encabezado de la clase) pero nunca dentro de un método. Este tipo de variables no se utilizan con tanta frecuencia como las variables de instancia.
- Un atributo de objeto se define dentro de un método y pertenece a un objeto determinado de la clase instanciada.
- Los métodos son bloques de código (o funciones) de una clase que se utilizan para definir el comportamiento de los objetos.

Definimos nuestra primera clase:

```
>>> class clase():
...    at_clase=1
...    def metodo(self):
...      self.at_objeto=1
...
>>> type(clase)
<class 'type'>
>>> clase.at_clase
1
>>> objeto=clase()
>>> objeto.at_clase
1
>>> objeto.at_objeto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'clase' object has no attribute 'at_objeto'
>>> objeto.metodo()
>>> objeto.at_objeto
1
```

1.4 Comparación con otros paradigmas

POO vs Programación Estructurada

Programación Estructurada

- Basada en funciones, procedimientos y control de flujo secuencial.
- Los datos suelen manejarse de forma independiente al código que los procesa.
- Ejemplo típico: C, Pascal.

POO

- Basada en clases y objetos que encapsulan datos y comportamiento.
- El flujo del programa surge de las interacciones entre objetos.
- Ejemplo: Python, Java, C#.

Ejemplo comparativo en Python

Versión estructurada:

```python
# Representar un estudiante y calcular su nota media
def crear_estudiante(nombre, notas):
    return {"nombre": nombre, "notas": notas}

def calcular_media(estudiante):
    return sum(estudiante["notas"]) / len(estudiante["notas"])

alumno = crear_estudiante("Ana", [8, 9, 7])
print(calcular_media(alumno))
```

Versión POO:

```python
class Estudiante:
    def __init__(self, nombre, notas):
        self.nombre = nombre
        self.notas = notas

    def media(self):
        return sum(self.notas) / len(self.notas)

alumno = Estudiante("Ana", [8, 9, 7])
print(alumno.media())
```

POO vs Programación Funcional

Programación Funcional

- Se basa en funciones puras, sin efectos secundarios y datos inmutables.
- Favorece operaciones como map, filter y reduce.
- Ejemplo: Haskell, parte de Python.

POO

- Favorece la mutabilidad y el cambio de estado de los objetos.
- Encapsula datos y comportamiento en la misma entidad.

Ejemplo funcional en Python:

```
notas = [8, 9, 7]
media = sum(notas) / len(notas)
print(media)
```

Ejemplo POO equivalente:

```
class Calificaciones:
    def __init__(self, notas):
        self.notas = notas

    def media(self):
        return sum(self.notas) / len(self.notas)

registro = Calificaciones([8, 9, 7])
print(registro.media())
```

Tabla comparativa de paradigmas

Característica	POO	Estructurada	Funcional
Unidad básica	Objeto	Función/procedimiento	Función pura
tab	Sí	Sí	No
Reutilización de código	Alta (herencia, composición)	Media	Alta (funciones puras)
Abstracción de datos	Sí	No	Sí
Ejemplos de uso	Apps modulares, videojuegos	Scripts pequeños	Procesamiento de datos

1.5 Ejemplos en Python paso a paso

Ejemplo 1 – Gestión básica de empleados

Objetivo: Representar empleados de una empresa con sus datos básicos y un método para mostrar su información.

```python
class Empleado:
    def __init__(self, nombre: str, puesto: str, salario: float):
        self.nombre = nombre
        self.puesto = puesto
        self.salario = salario

    def mostrar_informacion(self) -> None:
        print(f"Empleado: {self.nombre}")
        print(f"Puesto: {self.puesto}")
        print(f"Salario: {self.salario:.2f} €")
```

UML

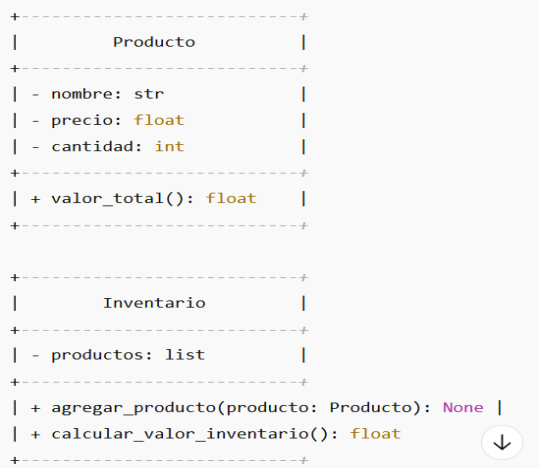
```
+------------------------------+
|          Empleado            |
+------------------------------+
| - nombre: str                |
| - puesto: str                |
| - salario: float             |
+------------------------------+
| + __init__(nombre, puesto, salario) |
| + mostrar_informacion(): None       |
+------------------------------+
```

Ejemplo 2 – Inventario de productos

Objetivo: Manejar un inventario que almacene productos y permita calcular su valor total.

```python
class Producto:
    def __init__(self, nombre: str, precio: float, cantidad: int):
        self.nombre = nombre
        self.precio = precio
        self.cantidad = cantidad

    def valor_total(self) -> float:
        return self.precio * self.cantidad


class Inventario:
    def __init__(self):
        self.productos = []

    def agregar_producto(self, producto: Producto) -> None:
        self.productos.append(producto)

    def calcular_valor_inventario(self) -> float:
        return sum(p.valor_total() for p in self.productos)


# Uso
p1 = Producto("Teclado", 20.0, 5)
p2 = Producto("Ratón", 15.0, 10)

inv = Inventario()
inv.agregar_producto(p1)
inv.agregar_producto(p2)

print(f"Valor total del inventario: {inv.calcular_valor_inventario():.2f} €")
```

UML



Ejemplo 3 – Biblioteca de libros

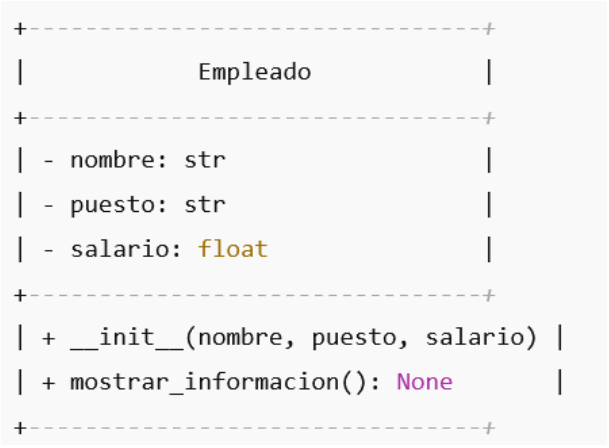
Objetivo: Gestionar libros y autores en una biblioteca.

```python
class Autor:
    def __init__(self, nombre: str, nacionalidad: str):
        self.nombre = nombre
        self.nacionalidad = nacionalidad

class Libro:
    def __init__(self, titulo: str, autor: Autor, anio: int):
        self.titulo = titulo
        self.autor = autor
        self.anio = anio

    def descripcion(self) -> str:
        return f"'{self.titulo}' de {self.autor.nombre} ({self.anio})"

class Biblioteca:
    def __init__(self):
        self.libros = []

    def agregar_libro(self, libro: Libro):
        self.libros.append(libro)

    def listar_libros(self):
        for libro in self.libros:
            print(libro.descripcion())

# Uso
a1 = Autor("Gabriel García Márquez", "Colombiana")
l1 = Libro("Cien años de soledad", a1, 1967)

biblio = Biblioteca()
biblio.agregar_libro(l1)
biblio.listar_libros()
```

1.6 Diagramas UML

Ejemplo 1 – Gestión de empleados

Diagrama UML

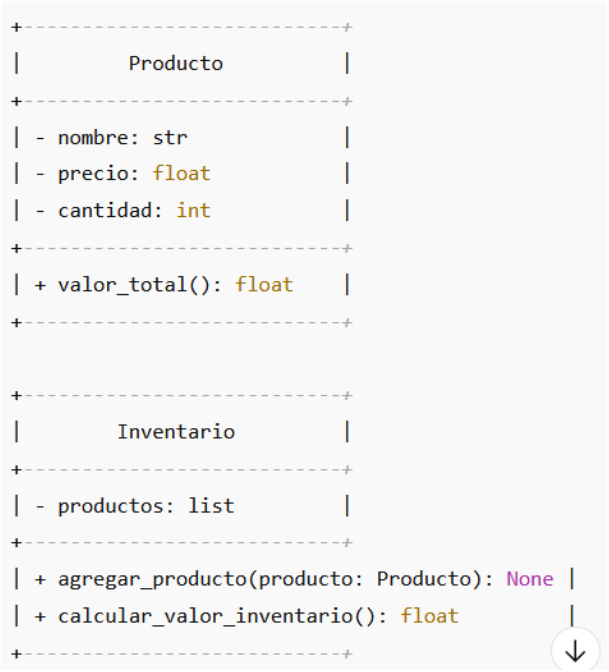


Explicación de diseño

- La clase Empleado **representa una sola entidad** con atributos que definen su estado y un método que expone su comportamiento.
- Los atributos se declaran en el constructor (__init__), y su visibilidad es **pública** en este ejemplo por simplicidad.
- El método mostrar_informacion() es responsable de mostrar los datos, manteniendo así la cohesión de la clase.

Ejemplo 2 – Inventario de productos

Diagrama UML



Explicación de diseño

- Producto es una **clase entidad** que almacena información básica y ofrece un método para calcular su valor total.
- Inventario es una **clase contenedora** que mantiene una colección de Producto y ofrece operaciones sobre ese conjunto.
- Este patrón de diseño se conoce como **composición**, donde un objeto contiene otros objetos y delega parte de su funcionalidad en ellos.

Ejemplo 3 – Biblioteca de libros

Diagrama UML

Explicación de diseño

- Autor y Libro tienen una relación de **asociación**, ya que un libro tiene un autor como uno de sus atributos.
- Biblioteca se relaciona con muchos Libro a través de una lista interna, representando una **agregación** (la biblioteca puede existir sin un libro específico).
- La separación de clases permite **extender el diseño fácilmente**, por ejemplo, añadiendo funcionalidades de búsqueda o clasificación sin modificar las clases base.

1.7 Errores comunes y como evitarlos

Error 1 – Usar atributos públicos sin control

Mal ejemplo

```
class Cuenta:
    def __init__(self, saldo):
        self.saldo = saldo  # Público

cuenta = Cuenta(100)
cuenta.saldo -= 50  # Modificación directa sin validación
```

Problema: No se valida que la operación sea válida. Un acceso directo puede dejar la clase en un estado inconsistente.

Solución

```
class Cuenta:
    def __init__(self, saldo):
        self.__saldo = saldo  # Privado

    def retirar(self, cantidad):
        if 0 < cantidad <= self.__saldo:
            self.__saldo -= cantidad
        else:
            print("Operación no válida.")

    def obtener_saldo(self):
        return self.__saldo
```

Se usan atributos **privados** (__saldo) y **métodos públicos** para controlar las modificaciones.

Error 2 – Abusar de la herencia

Mal ejemplo

```python
class Ave:
    pass

class Pato(Ave):
    pass

class Avion(Ave):  # Un avión no es un ave
    pass
```

Problema: Violación del principio de sustitución de Liskov: un Avion no debería sustituir a un Ave.

Solución: Usar **composición** en lugar de herencia.

```python
class Motor:
    pass

class Avion:
    def __init__(self):
        self.motor = Motor()
```

Error 3 – Métodos demasiado largos

Mal ejemplo

```python
class Factura:
    def calcular_total(self, productos, impuestos):
        total = 0
        for p in productos:
            total += p['precio']
        total += total * impuestos
        print(f"Total: {total}")
        # ... más lógica mezclada
```

Problema: El método mezcla cálculo, lógica de negocio y presentación.

Solución: Separar responsabilidades.

```python
class Factura:
    def calcular_subtotal(self, productos):
        return sum(p['precio'] for p in productos)

    def calcular_total(self, productos, impuestos):
        subtotal = self.calcular_subtotal(productos)
        return subtotal + subtotal * impuestos
```

Error 4 – Ignorar encapsulación de comportamiento

Mal ejemplo

```python
class Usuario:
    def __init__(self, nombre):
        self.nombre = nombre
        self.logged_in = False

usuario = Usuario("Ana")
usuario.logged_in = True  # Modificación externa
```

Solución

```python
class Usuario:
    def __init__(self, nombre):
        self.nombre = nombre
        self.__logged_in = False

    def iniciar_sesion(self, password):
        # Aquí iría la validación
        self.__logged_in = True

    def esta_conectado(self):
        return self.  logged in
```

Conclusión de esta sección

Los errores más comunes en POO ocurren cuando no se aplican principios como **encapsulación**, **responsabilidad única** o **composición sobre herencia**. Siguiendo buenas prácticas, los programas serán más **mantenibles** y **escalables**.

1.8 Ejercicios propuestos

Ejercicio 1 – Clase Rectangulo

Crea una clase Rectangulo que:

- Reciba **base** y **altura** en el constructor.
- Tenga métodos para calcular el **área** y el **perímetro**.
- Permita mostrar sus dimensiones en un formato legible.

Objetivo: Practicar **atributos**, **métodos** y **cálculo de propiedades geométricas**.

Ejercicio 2 – Clase Coche con encapsulación

Crea una clase Coche que:

- Tenga atributos **marca**, **modelo**, y un atributo privado **velocidad** inicializado en 0.
- Tenga métodos acelerar y frenar que modifiquen la velocidad de forma controlada.
- Tenga un método estado que indique la velocidad actual.

Objetivo: Practicar **atributos privados** y **control de acceso**.

Ejercicio 3 – Sistema de pedidos

Crea:

- Una clase Producto con nombre y precio.
- Una clase Pedido que almacene varios productos y calcule el total.
- Métodos para añadir productos y mostrar el resumen del pedido.

Objetivo: Practicar **composición** de clases y **listas de objetos**.

Ejercicio 4 – Herencia con animales

Crea:

- Una clase base Animal con método hacer_sonido.
- Clases derivadas Perro y Gato que sobrescriban ese método con un sonido distinto.

Objetivo: Practicar **herencia** y **polimorfismo**.

Ejercicio 5 – Biblioteca mejorada

Amplía el ejemplo de la biblioteca:

- Agrega búsqueda por autor.
- Agrega un método para eliminar un libro dado su título.

Objetivo: Practicar **operaciones CRUD** en listas de objetos.

Nota: No es obligatorio resolver todos los ejercicios de una sola vez. Pueden abordarse de forma gradual, reforzando así cada concepto.

1.9 Ejercicios resueltos

Ejercicio 1 – Clase Rectangulo

```python
class Rectangulo:
    def __init__(self, base: float, altura: float):
        self.base = base
        self.altura = altura

    def area(self) -> float:
        return self.base * self.altura

    def perimetro(self) -> float:
        return 2 * (self.base + self.altura)

    def mostrar(self) -> None:
        print(f"Rectángulo de base {self.base} y altura {self.altura}")

# Uso
r = Rectangulo(5, 3)
r.mostrar()
print(f"Área: {r.area()}")
print(f"Perímetro: {r.perimetro()}")
```

Conceptos aplicados: atributos, métodos, encapsulación simple.

Ejercicio 2 – Clase Coche con encapsulación

```python
class Coche:
    def __init__(self, marca: str, modelo: str):
        self.marca = marca
        self.modelo = modelo
        self.__velocidad = 0

    def acelerar(self, cantidad: int):
        if cantidad > 0:
            self.__velocidad += cantidad

    def frenar(self, cantidad: int):
        if 0 < cantidad <= self.__velocidad:
            self.__velocidad -= cantidad

    def estado(self):
        print(f"{self.marca} {self.modelo} - Velocidad: {self.__velocidad} km/h")

# Uso
c = Coche("Toyota", "Corolla")
c.acelerar(50)
c.estado()
c.frenar(20)
c.estado()
```

Conceptos aplicados: atributos privados, control de acceso.

Ejercicio 3 – Sistema de pedidos

```python
class Producto:
    def __init__(self, nombre: str, precio: float):
        self.nombre = nombre
        self.precio = precio

class Pedido:
    def __init__(self):
        self.productos = []

    def agregar_producto(self, producto: Producto):
        self.productos.append(producto)

    def total(self) -> float:
        return sum(p.precio for p in self.productos)

    def mostrar_resumen(self):
        print("Resumen del pedido:")
        for p in self.productos:
            print(f"- {p.nombre}: {p.precio} €")
        print(f"Total: {self.total()} €")

# Uso
p1 = Producto("Teclado", 20)
p2 = Producto("Ratón", 15)
pedido = Pedido()
pedido.agregar_producto(p1)
pedido.agregar_producto(p2)
pedido.mostrar_resumen()
```

Conceptos aplicados: composición, listas de objetos.

Ejercicio 4 – Herencia con animales

```python
class Animal:
    def hacer_sonido(self):
        print("Sonido genérico")

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau")

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

# Uso
animales = [Perro(), Gato()]
for a in animales:
    a.hacer_sonido()
```

Conceptos aplicados: herencia, polimorfismo, sobreescritura de métodos.

Ejercicio 5 – Biblioteca mejorada

```python
class Autor:
    def __init__(self, nombre: str):
        self.nombre = nombre

class Libro:
    def __init__(self, titulo: str, autor: Autor):
        self.titulo = titulo
        self.autor = autor

class Biblioteca:
    def __init__(self):
        self.libros = []

    def agregar_libro(self, libro: Libro):
        self.libros.append(libro)

    def buscar_por_autor(self, nombre_autor: str):
        return [l for l in self.libros if l.autor.nombre == nombre_autor]

    def eliminar_libro(self, titulo: str):
        self.libros = [l for l in self.libros if l.titulo != titulo]

# Uso
a1 = Autor("Gabriel García Márquez")
l1 = Libro("Cien años de soledad", a1)

b = Biblioteca()
b.agregar_libro(l1)

print([libro.titulo for libro in b.buscar_por_autor("Gabriel García Márquez")])
b.eliminar_libro("Cien años de soledad")
print(b.libros)
```

Conceptos aplicados: CRUD, listas, relaciones entre clases.

1.10 Referencias bibliográficas y enlaces oficiales

Libros:

1. Lutz, M. (2013). *Learning Python*. O'Reilly Media.
    - Cubre desde conceptos básicos hasta programación avanzada en Python.
    - Incluye capítulos sobre POO muy completos y detallados.
2. Sweigart, A. (2019). *Automate the Boring Stuff with Python*. No Starch Press.
    - Excelente para principiantes, con aplicaciones prácticas de POO.
3. Zelle, J. (2017). *Python Programming: An Introduction to Computer Science*. Franklin, Beedle & Associates.
    - Combina teoría de programación y práctica en Python, ideal para entornos académicos.
4. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
    - Referencia fundamental para patrones de diseño, aplicables también en Python.

Documentación oficial y recursos en línea:

- **Documentación oficial de Python:**

https://docs.python.org/3/
Incluye la sección *Classes* con explicación detallada de sintaxis y funcionalidades.

- **PEP 8 – Style Guide for Python Code:**

https://peps.python.org/pep-0008/
Guía oficial de estilo en Python, imprescindible para escribir código limpio.

- **Python Tutor – Visualizador de código:**

https://pythontutor.com/
Herramienta interactiva para seguir la ejecución del código paso a paso.

- **Real Python:**

https://realpython.com/
Plataforma con artículos y tutoriales de alta calidad sobre Python y POO.