reporte 2.md 2025-10-10

# Explicacion de Programacion Orientada a Objetos (POO) y Comparación entre C y Python

Este documento explica los conceptos principales de la **Programación Orientada a Objetos (POO)** utilizando ejemplos del sistema de biblioteca migrado de C a Python.

También se incluye una breve comparación entre ambas versiones y las conclusiones sobre las ventajas de la POO.

# 1. Conceptos de Programacion Orientada a Objetos

#### Clase

Una **clase** es una plantilla que define las propiedades y comportamientos que tendrán los objetos de un tipo específico.

#### **Ejemplo en Python:**

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

# **Objeto**

Un **objeto** es una instancia concreta de una clase. Es un ejemplar real creado a partir del molde que define la clase.

#### **Ejemplo:**

```
libro1 = Book("1984", "George Orwell")
libro2 = Book("Cien años de soledad", "Gabriel García Márquez")
```

#### Herencia

La **herencia** permite crear clases nuevas basadas en otras existentes, reutilizando código y comportamientos.

# **Ejemplo:**

```
class Item:
pass
```

reporte\_2.md 2025-10-10

```
class Book(Item):
   pass
```

En el sistema de biblioteca, Book y Magazine heredan de Item.

### **Encapsulamiento**

El **encapsulamiento** consiste en ocultar los detalles internos de una clase y controlar el acceso a sus datos mediante propiedades o métodos públicos.

#### **Ejemplo:**

```
class User:
    def __init__(self, name):
        self._name = name # atributo protegido

@property
    def name(self):
        return self._name
```

Esto protege los datos internos y evita su modificación directa desde fuera de la clase.

#### **Abstraccion**

La **abstracción** permite definir *qué* deben hacer las clases hijas, pero no *cómo* lo hacen.

#### **Ejemplo:**

```
from abc import ABC, abstractmethod

class Item(ABC):
    @abstractmethod
    def display(self):
        pass
```

Item es una clase abstracta: no puede instanciarse directamente y obliga a las subclases a implementar display().

#### **Polimorfismo**

El **polimorfismo** permite que diferentes clases usen el mismo método con comportamientos distintos.

#### **Ejemplo:**

reporte\_2.md 2025-10-10

```
class Book(Item):
    def display(self):
        return "Libro"

class Magazine(Item):
    def display(self):
        return "Revista"
```

Cuando se llama item.display(), Python ejecuta la versión adecuada según el tipo de objeto (Book o Magazine).

# 2. Comparacion entre la version en C y la version en Python

Aspecto	Versión en C	Versión en Python (POO)
Estructura	Usa struct y funciones separadas	Usa clases y métodos
Herencia	No existe de forma nativa	Soportada con class Hija(Padre)
Encapsulamiento	Sin control real de acceso	Atributos privados y propiedades
Abstraccion	Simulada con punteros a funciones	Clases abstractas con ABC
Polimorfismo	Manual con switch o enum	Automático mediante herencia
Gestion de memoria	Manual (malloc, free)	Automática con recolector de basura
Persistencia	Archivos con fscanf y fprintf	Uso de json para guardar estructuras complejas

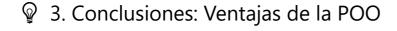
# **Ejemplo practico:**

En C:

```
void addBook(book_t **library, int *count);
```

## En Python:

```
library.register_book(101, "El Principito", "Antoine de Saint-Exupéry", 1943, "Fantasía", 5)
```



reporte\_2.md 2025-10-10

• Mejor organizacion y claridad: el código se estructura en clases como Libro, Usuario, Biblioteca.

- **Reutilizacion de codigo:** la herencia evita duplicación y facilita añadir nuevas clases (DVD, Periódico, etc.).
- Extensibilidad: permite ampliar funcionalidades sin alterar el código base.
- Proteccion de datos: el encapsulamiento evita acceso no controlado a los atributos.
- Flexibilidad: el polimorfismo permite tratar objetos diferentes de manera uniforme.
- Mantenibilidad: el código POO es más fácil de depurar, probar y escalar.

# Referencias

https://github.com/adrianbalderas373488/Portafolio