+++ date = '2025-02-21T10:20:19-08:00' draft = false title = 'Practica2' +++

Biblioteca.py

*El programa que analizaremos en esta practica es el mismo programa que analizamos en la practica 1 pero esta vez con el **paradigma orientado a objetos***

Descripción general del programa

Este programa implementa un sistema de gestión de biblioteca en Python utilizando los principios más relevantes del paradigma orientado a objetos (POO). Permite registrar libros (físicos o digitales), administrar miembros y gestionar préstamos y devoluciones. A continuación, se describen tanto su funcionamiento general como su relación con los conceptos clave de POO.

Como y por que funciona

El programa funciona gracias a la correcta aplicación de conceptos de programación orientada a objetos, que permiten estructurar el código de forma modular, reutilizable y mantenible.

Nombres

*Los nombres se utilizan para identificar variables, funciones, clases, atributos, etc. En este código, algunos ejemplos son **library**, **books**, **members**, **add_book**, **DigitalBook**, **issue_book**, entre otros.*

Objetos

*Los objetos son instancias de clases. En este programa, por ejemplo, **book**, **digital_book**, **member** y **library** son objetos creados a partir de las clases `Book`, `DigitalBook`, `Member` y `Library`. Estos objetos encapsulan datos y comportamientos relacionados con la biblioteca.*

Entornos

Un entorno es el contexto de ejecución de las variables y funciones. En este código, el entorno principal es la función `main()`, donde se crean instancias y se gestionan las operaciones del sistema. También hay entornos locales dentro de los métodos de cada clase.

Bloques

Los bloques se delimitan con indentación (en Python). Cada clase y método contiene un bloque de código. Por ejemplo:

```
def add_book(self, book):
    '''Método para agregar un libro a la biblioteca'''
    self.books.append(book)
    print("\nEl libro fue agregado exitosamente!\n")
```

Alcance

El alcance determina dónde pueden ser usadas las variables. En este programa:

- *Variables locales como **book_id** o **member_id** en `main()` solo existen dentro de esa función.*
- *Atributos como **self.books** o **self.members** tienen alcance dentro del objeto de la clase `Library`, y pueden ser usados por todos sus métodos.*

Clases

Las clases son plantillas que definen la estructura y el comportamiento de los objetos. En este programa, se definen varias clases:

- `Book`: Representa libros físicos.
- `DigitalBook`: Hereda de `Book`, representa libros digitales.
- `Member`: Representa a un usuario de la biblioteca.
- `Library`: Administra libros y miembros.
- `Genre`: Enumera géneros literarios disponibles.

Herencia

La herencia se usa cuando una clase toma atributos y métodos de otra. En este código, `DigitalBook` hereda de `Book`, reutilizando atributos y extendiendo funcionalidad con `file_format`.

Polimorfismo

El polimorfismo permite que una misma interfaz se use para distintos tipos. En este programa, el método `to_dict()` es redefinido en `DigitalBook`, pero puede usarse igual que en `Book`, adaptándose al tipo de objeto.

Composición

La clase `Library` contiene objetos de tipo `Book` y `Member`, lo cual es un ejemplo de composición: una biblioteca "está compuesta" por libros y miembros.

Modularidad

El código está organizado en módulos. Por ejemplo, el archivo que contiene estas clases puede importarse desde otros scripts como un módulo. Además, se usa el módulo `memory_management` para controlar la asignación de memoria.

Encapsulamiento

Cada clase define sus propios atributos y métodos, encapsulando su comportamiento. Esto permite proteger los datos y manejar la complejidad, por ejemplo, los atributos `self.books` y `self.members` solo son manipulados por los métodos de `Library`.

Cómo funciona el programa (pasos para hacerlo funcionar)

1. **Importar el módulo**: Asegúrate de tener los archivos necesarios, como el módulo `memory_management.py` y el archivo principal con las clases (`library_module.py`).
2. **Ejecución del programa**: Al ejecutar el archivo principal (por ejemplo, `python library_module.py`), se inicia la función `main()`.
3. **Carga inicial**: Al iniciar, el sistema intenta cargar los datos de libros (`library.json`) y miembros (`members.json`).
4. **Menú de usuario**: Se muestra un menú interactivo donde puedes:
    - Agregar libros (físicos o digitales).
    - Mostrar libros existentes.
    - Agregar miembros.
    - Prestar libros.
    - Devolver libros.
    - Ver miembros y sus libros.
    - Buscar un miembro.
    - Guardar la información y salir.
5. **Persistencia**: Antes de salir, el sistema guarda los cambios en archivos `.json` para mantener la información entre sesiones.
6. **Gestión de memoria**: Se lleva un control de la memoria dinámica usando `memory_management`, registrando las asignaciones y liberaciones de objetos.

Memory_managemet.py

Clases

*La clase **MemoryManagement** define una estructura que agrupa atributos (como `heap_allocations` y `heap_deallocations`) y métodos (como `increment_heap_allocations`, `increment_heap_deallocations` y `display_memory_usage`) para encapsular el comportamiento relacionado con la gestión de memoria.*

Encapsulamiento

Se logra mediante el uso de métodos que controlan el acceso y modificación de los atributos internos. Por ejemplo, los contadores de memoria solo se pueden modificar a través de los métodos definidos en la clase.

Abstracción

La clase oculta los detalles internos del seguimiento de memoria y proporciona una interfaz clara mediante métodos públicos que permiten interactuar con ese comportamiento sin exponer directamente los atributos.

Objetos

*Se crea un objeto llamado **memory_management** a partir de la clase **MemoryManagement**, que se utiliza globalmente para rastrear las operaciones de asignación y liberación de memoria.*