+++ date = '2025-05-16T08:43:05-08:00' draft = false title = 'Práctica 2: Python Paradigma Orientado a Objetos' summary = 'Edgar Rafael Garcia Ceseña' +++

Introducción

Esta práctica se centra en el desarrollo de un sistema de gestión para una biblioteca, implementado en Python. El programa aplica los fundamentos de la Programación Orientada a Objetos (POO), representando entidades del mundo real como libros, usuarios y la biblioteca mediante clases. Además, se complementa con una interfaz web utilizando el framework Flask.

Desarrollo

memory_management.py

Este archivo define una clase encargada de gestionar el uso de memoria dentro del programa.

```
class MemoryManagement:
    '''Class to manage memory usage'''
    def __init__(self):
        self.heap_allocations = 0
        self.heap_deallocations = 0

    def increment_heap_allocations(self, size):
        '''Increment heap allocations'''
        self.heap_allocations += size

    def increment_heap_deallocations(self, size):
        '''Increment heap deallocations '''
        self.heap_deallocations += size

    def display_memory_usage(self):
        '''Display memory usage'''
        print(f"Heap allocations: {self.heap_allocations} bytes")
        print(f"Heap deallocations: {self.heap_deallocations} bytes")

memory_management = MemoryManagement()
```

En esta clase, el método constructor **init** inicializa dos contadores en cero:

```
def __init__(self):
    self.heap_allocations = 0
    self.heap_deallocations = 0
```

Posteriormente, la clase cuenta con tres métodos: uno para aumentar las asignaciones en el heap, otro para las desasignaciones, y el tercero para mostrar el estado actual de ambos contadores.

```
def increment_heap_allocations(self, size):
    '''Increment heap allocations'''
    self.heap_allocations += size

def increment_heap_deallocations(self, size):
    '''Increment heap deallocations '''
    self.heap_deallocations += size

def display_memory_usage(self):
    '''Display memory usage'''
    print(f"Heap allocations: {self.heap_allocations} bytes")
    print(f"Heap deallocations: {self.heap_deallocations} bytes")
```

Finalmente, se crea una instancia de la clase para su uso posterior:

```
memory_management = MemoryManagement()
```

biblioteca.py

Este módulo comienza con dos importaciones necesarias:

```
import json
from memory_management import memory_management
```

El módulo **json** permite guardar y cargar los datos desde archivos **.json**, mientras que **memory_management** importa la instancia que gestiona el uso de memoria.

A continuación se define la clase **Genre**, que funciona como una especie de enumeración para clasificar los libros:

```
class Genre:
    '''Clase para definir los generos de los libros'''
    FICTION = "Ficcion"
    NON_FICTION = "No Ficcion"
    SCIENCE = "Ciencia"
    HISTORY = "Historia"
    FANTASY = "Fantasia"
    BIOGRAPHY = "Biografia"
    OTHER = "Otro"

    @classmethod
    def all_genres(cls):
        return [cls.FICTION, cls.NON_FICTION, cls.SCIENCE, cls.HISTORY,
cls.FANTASY, cls.BIOGRAPHY, cls.OTHER]
```

La clase **Book** representa un libro físico, con métodos para transformar objetos en diccionarios y viceversa. También registra el uso de memoria mediante el objeto **memory_management**.

```
class Book:
    '''Clase para definir los libros de la biblioteca'''
    def __init__(self, book_id, title, author, publication_year, genre, quantity):
        self.id = book_id
        self.title = title
        self.author = author
        self.publication_year = publication_year
        self.genre = genre
        self.quantity = quantity
        memory_management.increment_heap_allocations(1)

    def __del__(self):
        memory_management.increment_heap_deallocations(1)

    def to_dict(self):
        '''Método para convertir los datos del libro en un diccionario'''
        return {
            "id": self.id,
            "title": self.title,
            "author": self.author,
            "publication_year": self.publication_year,
            "genre": self.genre,
            "quantity": self.quantity
        }

    @staticmethod
    def from_dict(data):
        '''Método para crear un objeto libro a partir de un diccionario'''
        return Book(
            data["id"],
            data["title"],
            data["author"],
            data["publication_year"],
            data["genre"],
            data["quantity"]
        )
```

La clase **DigitalBook** extiende a Book y agrega un atributo para el formato de archivo. También sobreescribe métodos, aplicando así polimorfismo.

```
class DigitalBook(Book):
    '''Clase para definir los libros digitales de la biblioteca'''
    def __init__(self, book_id, title, author, publication_year, genre, quantity,
file_format):
        '''Constructor de la clase DigitalBook'''
        super().__init__(book_id, title, author, publication_year, genre,
quantity)
```

```
        self.file_format = file_format

    def to_dict(self):
        '''Método para convertir los datos del libro digital en un diccionario'''
        data = super().to_dict()
        data["file_format"] = self.file_format
        return data

    @staticmethod
    def from_dict(data):
        '''Método para crear un objeto libro digital a partir de un diccionario'''
        return DigitalBook(
            data["id"],
            data["title"],
            data["author"],
            data["publication_year"],
            data["genre"],
            data["quantity"],
            data["file_format"]
        )
```

La clase **Member** sigue una estructura similar a **Book**, con atributos y métodos para representar a un usuario de la biblioteca.

```
class Member:
    '''Clase para definir los miembros de la biblioteca'''
    def __init__(self, member_id, name):
        '''Constructor de la clase Member'''
        self.id = member_id
        self.name = name
        self.issued_books = []
        memory_management.increment_heap_allocations(1)

    def __del__(self):
        memory_management.increment_heap_deallocations(1)

    def to_dict(self):
        '''Método para convertir los datos del miembro en un diccionario'''
        return {
            "id": self.id,
            "name": self.name,
            "issued_books": self.issued_books
        }

    @staticmethod
    def from_dict(data):
        '''Método para crear un objeto miembro a partir de un diccionario'''
        member = Member(data["id"], data["name"])
        member.issued_books = data["issued_books"]
        return member
```

Library es la clase principal del sistema y gestiona la colección de libros y la lista de miembros.

```
class Library:
    '''Clase para definir la biblioteca'''
    def __init__(self):
        '''Constructor de la clase Library'''
        self.books = []
        self.members = []
        memory_management.increment_heap_allocations(1)
```

También contiene métodos para agregar libros, emitir préstamos, recibir devoluciones y guardar datos.

La función principal **main()** ofrece una interfaz en consola para interactuar con el sistema:

```
def main():
    '''Función principal para ejecutar el programa'''
    library = Library()
    library.load_library_from_file("library.json")
    library.load_members_from_file("members.json")

    while True:
        print("\nMenu de sistema de manejo de biblioteca\n")
        print("\t1. Agregar un libro")
        print("\t2. Mostrar libros disponibles")
        print("\t3. Agregar un miembro")
        print("\t4. Prestar libro")
        print("\t5. Devolver libro")
        print("\t6. Mostrar miembros disponibles")
        print("\t7. Buscar miembro")
        print("\t8. Guardar y salir")
        choice = int(input("Indica tu opcion: "))

        if choice == 1:
            book_id = int(input("Ingresa ID del libro: "))
            title = input("Ingresa titulo del libro: ")
            author = input("Ingresa nombre del autor: ")
            publication_year = int(input("Ingresa el ano de publicacion: "))
            genre = input("Ingresa el genero del libro: ")
            quantity = int(input("Ingresa la cantidad de libros: "))
            is_digital = input("Es un libro digital? (s/n): ").lower() == 's'
            if is_digital:
                file_format = input("Ingresa el formato del archivo: ")
                book = DigitalBook(book_id, title, author, publication_year,
genre, quantity, file_format)
            else:
                book = Book(book_id, title, author, publication_year, genre,
quantity)
            library.add_book(book)
        elif choice == 2:
            library.display_books()
```

```
        elif choice == 3:
            member_id = int(input("Ingresa el ID del miembro: "))
            name = input("Ingresa el nombre del miembro: ")
            member = Member(member_id, name)
            library.add_member(member)
        elif choice == 4:
            member_id = int(input("Ingresa el ID del miembro: "))
            book_id = int(input("Ingresa el ID del libro: "))
            library.issue_book(book_id, member_id)
        elif choice == 5:
            member_id = int(input("Ingresa el ID del miembro: "))
            book_id = int(input("Ingresa el ID del libro: "))
            library.return_book(book_id, member_id)
        elif choice == 6:
            library.display_members()
        elif choice == 7:
            member_id = int(input("Ingresa el ID del miembro: "))
            library.search_member(member_id)
        elif choice == 8:
            library.save_library_to_file("library.json")
            library.save_members_to_file("members.json")
            print("Saliendo del programa\n")
            break
        else:
            print("Esta no es una opcion valida!!!\n")
```

biblioteca_web.py

Este archivo permite llevar el sistema a la web mediante Flask. Se implementan rutas y funciones que replican la lógica de **main()** del script anterior, pero adaptadas al contexto de una aplicación web.

Conclusión

La práctica permitió construir un sistema completo de gestión de biblioteca, que incluye tanto la lógica orientada a objetos como su representación en una interfaz web. El uso de la Programación Orientada a Objetos permitió estructurar el código de forma modular y reutilizable, facilitando su mantenimiento y escalabilidad.