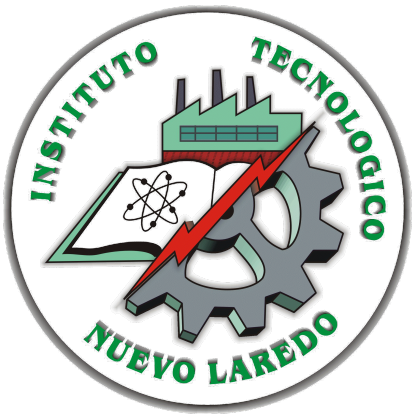
INSTITUTO TECNOLÓGICO DE NUEVO LAREDO

INGENIERÍA EN SISTEMAS COMPUTACIONALES



Proyecto Final: Recomendador de Anime con Python

Alumno: Gianfranco Martínez Cisneros

No. Control: 18100196

Materia: Programación Multiparadigma

Docente: Ing. Juan Manuel Ahumada Vázquez

Objetivo

El objetivo de este proyecto es desarrollar un **recomendador de anime** que pueda analizar una lista de animes y sugerir nuevos basándose en los gustos previos del usuario. Utilizando **Python** y aplicando los paradigmas de **Programación Orientada a Objetos (OOP)**, **Programación Funcional** y **Programación Reactiva/Concurrente**, se crea un sistema eficiente y altamente modular que maneja la inmutabilidad, el procesamiento concurrente y funciones puras para una experiencia de usuario fluida y rápida.

1. Descripción del Proyecto

El proyecto utiliza el **dataset 'Anime Recommendations Database'** para almacenar y procesar los datos de animes. Los usuarios pueden interactuar con la aplicación a través de una interfaz gráfica (Tkinter), donde pueden aplicar filtros y obtener recomendaciones personalizadas. Además, las recomendaciones se procesan de manera **asíncrona** para garantizar que la interfaz permanezca responsiva.

Interfaz Gráfica:

La aplicación presenta una interfaz profesional y clara para el usuario, con las siguientes funcionalidades:

- **Mostrar animes recomendados** basados en el perfil de usuario.
- **Filtrar animes** por género, rating, o por un rating mínimo.
- **Obtener recomendaciones avanzadas**, que son procesadas asíncronamente para no bloquear la UI.
- **Manejo eficiente de eventos** a través de funciones que actualizan el estado de los botones y la lista de animes de forma concurrente.

---

2. Estructura del Proyecto

La estructura del proyecto es la siguiente:

```
FinalMulti/
├── main.py                      # Punto de entrada con Tkinter
├── data/
│   ├── anime.csv
│   └── rating.csv
├── models/
│   ├── __init__.py
│   ├── anime.py
│   ├── user_rating.py
│   ├── base_recommender.py
│   └── recommender.py
├── logic/
│   ├── __init__.py
│   ├── filters.py
│   ├── async_tasks.py
│   ├── aggregator.py
│   └── utils.py
├── ui/
│   └── interface.py
├── .env
├── requirements.txt
└── README.md
```

Descripción de las Carpetas:

- `data/`: Contiene el dataset 'Anime Recommendations Database', que se utiliza para cargar los datos.
- `models/`: Define las clases `Anime` que representan los datos de un anime.
- `ui/`: Contiene el código para la interfaz gráfica, usando Tkinter.
- `async_tasks.py`: Define las tareas asíncronas para las recomendaciones avanzadas.
- `main.py`: Es el script principal que ejecuta la aplicación.

---

3. Explicación del Código

3.1. Clase `Anime`

La clase `Anime` está diseñada con un buen nivel de encapsulamiento, usando atributos privados y propiedades de solo lectura que permiten acceder a la información sin riesgo de modificar el estado interno directamente. Esto asegura que los datos se mantengan consistentes y controlados.

La separación del string `genre` en una lista de géneros en el constructor es una forma eficiente de preparar los datos para posteriores operaciones, facilitando filtrados y búsquedas.

El método `has_genre` proporciona una funcionalidad clara y simple para validar si un anime pertenece a un género dado, usando una expresión generadora para hacer la comparación de manera eficiente y case-insensitive.

El método `__str__` devuelve una representación legible que facilita la visualización rápida del objeto, útil para depuración o despliegue en la interfaz.

En conjunto, el diseño de esta clase garantiza que cada instancia representa un anime con sus atributos correctamente protegidos y expuestos, facilitando su uso en funciones puras y en procesos concurrentes sin riesgos de efectos secundarios indeseados.

```
class Anime:
    def __init__(self, anime_id: int, name: str, genre: str, type_: str, episodes:
int, rating: float, members: int):
        self._anime_id = anime_id
        self._name = name
        # Separamos la cadena genre en lista de géneros
        self._genres = genre.split(', ') if genre else []
        self._type = type_
        self._episodes = episodes
        self._rating = rating
        self._members = members

    # Encapsulamiento con propiedades que respetan el dataset
    @property
    def anime_id(self):
        return self._anime_id

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

    @property
    def genres(self):
        return self._genres

    @property
    def type(self):
        return self._type

    @property
```

```
    def episodes(self):
        return self._episodes

    @property
    def rating(self):
        return self._rating

    @property
    def members(self):
        return self._members

    def __str__(self):
        return f"{self._name} ({self._type}) - {', '.join(self._genres)} - Rating:
{self._rating}"

    def has_genre(self, genre: str) -> bool:
        """Verifica si el anime contiene un género específico."""
        return genre.lower() in (g.lower() for g in self._genres)
```

- `name`: Nombre del anime.
- `rating`: Rating promedio del anime.
- `genres`: Lista de géneros asociados al anime.
- `year`: Año de estreno del anime.

3.2 Funciones Funcionales

En este apartado, se implementan funciones puras para el procesamiento de listas de objetos `Anime`. Estas funciones utilizan conceptos fundamentales de programación funcional, como funciones de orden superior (`filter`, `map`, `reduce`), inmutabilidad y ausencia de efectos secundarios.

Filtrar por rating mínimo

```
def filtrar_por_rating_minimo(animes: List[Anime], umbral: float) -> List[Anime]:
    return list(filter(lambda a: a.rating >= umbral, animes))
```

- Esta función recibe dos parámetros: una lista de objetos `Anime` y un valor `umbral` de tipo `float`.
- Utiliza la función de orden superior `filter` junto con una función lambda que verifica si el `rating` de cada anime es mayor o igual al umbral:

  ```
  lambda a: a.rating >= umbral
  ```

- `filter` devuelve un iterador con los elementos que cumplen la condición, y `list()` lo convierte en una lista.
- Al no modificar la lista original y devolver una nueva, la función respeta el principio de inmutabilidad.
- Esta función es **pura**, ya que su salida depende únicamente de sus argumentos y no tiene efectos secundarios.

Filtrar por género

```
def filtrar_por_genero(animes: List[Anime], genero: str) -> List[Anime]:
    return list(filter(lambda a: genero.lower() in [g.lower() for g in a.genres],
animes))
```

- Filtra la lista de animes para incluir solo aquellos que contienen el género especificado.
- La función lambda convierte todos los géneros del anime a minúsculas para hacer la búsqueda **insensible a mayúsculas/minúsculas**:

  ```
  [g.lower() for g in a.genres]
  ```

- Luego, verifica si el género buscado (`genero.lower()`) está dentro de esa lista.
- Se emplea `filter` para filtrar los animes que cumplen la condición.

- Devuelve una nueva lista, sin modificar la original, manteniendo la pureza funcional.

Ordenar por rating

```
def ordenar_por_rating(animes: List[Anime], descendente: bool = True) ->
List[Anime]:
    return sorted(animes, key=lambda a: a.rating, reverse=descendente)
```

- Ordena los animes según su atributo `rating`.
- Usa la función integrada `sorted` que devuelve una nueva lista ordenada (sin alterar la original).
- La clave para la ordenación es el `rating` extraído por la función lambda:

  ```
  key=lambda a: a.rating
  ```

- El parámetro `descendente` controla si la lista queda en orden descendente (`True` por defecto) o ascendente.
- Esta función también es pura, sin efectos secundarios.

Obtener rating promedio

```
def obtener_rating_promedio(animes: List[Anime]) -> float:
    if not animes:
        return 0.0
    total = reduce(lambda acc, a: acc + a.rating, animes, 0.0)
    return total / len(animes)
```

- Calcula el promedio de los ratings de la lista.
- Primero, verifica si la lista está vacía para evitar división por cero y devuelve `0.0` en ese caso.
- Emplea la función `reduce` para acumular la suma total de ratings:

  ```
  reduce(lambda acc, a: acc + a.rating, animes, 0.0)
  ```

- Divide el total acumulado por la cantidad de animes para obtener el promedio.
- La función es pura y sin efectos secundarios, cumpliendo con los principios funcionales.

Resumen del cumplimiento funcional

- Las funciones son **funciones puras**: su salida depende únicamente de sus parámetros y no modifican datos externos ni la entrada.
- Usan funciones de orden superior (`filter`, `sorted`, `reduce`) para transformar y procesar datos.
- Mantienen la **inmutabilidad** al devolver nuevas listas o valores sin alterar las listas originales.
- Aplican un estilo declarativo, legible y modular, facilitando el mantenimiento y escalabilidad del código.

3.3 Funciones Asíncronas

En esta sección, introducimos una función que emplea **programación asíncrona** para simular un procesamiento que toma tiempo, como podría ser una llamada a una API externa o un cálculo complejo, sin bloquear el flujo de ejecución principal.

Análisis de la función `recomendar_lenta`

```
async def recomendar_lenta(animes: List[Anime]) -> str:
    await asyncio.sleep(2)  # Simula demora externa
    if not animes:
        return "No se encontraron recomendaciones avanzadas."

    mensaje = "Recomendación avanzada basada en tus gustos:\n"
    for anime in animes[:3]:
        mensaje += f"- Tal vez te guste también '{anime.name}' (Rating:
{anime.rating:.1f})\n"
    return mensaje
```

- **Declaración asíncrona:** Se usa la palabra clave `async` para definir una función asíncrona que retorna una coroutine, lo que permite ejecutar operaciones concurrentes sin bloquear el hilo principal.
- **Simulación de espera no bloqueante:**

  ```
  await asyncio.sleep(2)
  ```

  Aquí se simula una demora de 2 segundos (por ejemplo, la latencia de una llamada externa o proceso intensivo). `await` libera el control mientras espera, permitiendo que otras tareas se ejecuten concurrentemente.
- **Manejo de datos:**
  - Si la lista `animes` está vacía, se retorna inmediatamente un mensaje que indica que no hay recomendaciones.
  - De lo contrario, se construye un mensaje con las primeras 3 recomendaciones basadas en los animes recibidos, mostrando su nombre y rating con formato.
- **Retorno:** La función retorna un mensaje tipo `str` que puede ser mostrado al usuario o procesado en un pipeline más amplio.

Cumplimiento de requisitos Reactivos/Concurrentes

- **Concurrencia:** El uso de `async` y `await` permite que esta función sea integrada dentro de un entorno asincrónico con `asyncio`, favoreciendo la concurrencia y el procesamiento eficiente de múltiples tareas.
- **No bloqueo:** La simulación de la demora con `await asyncio.sleep(2)` no bloquea el hilo principal ni el event loop, lo que permite que el programa responda a otros eventos o solicitudes mientras

espera.

- **Manejo de eventos:** Esta función puede ser usada como parte de un sistema reactivo donde se manejen eventos de usuario o respuestas de red de forma asíncrona.
- **Extensibilidad:** En una versión real, el `asyncio.sleep` puede ser reemplazado por llamadas asíncronas reales a APIs o bases de datos, manteniendo la misma estructura.

3.4 Funciones Asíncronas en la Interfaz

Cuando desarrollamos una **interfaz gráfica (GUI)**, como con Tkinter, es fundamental que las operaciones que puedan tomar tiempo no bloqueen el hilo principal encargado de la interfaz. Si el hilo principal queda bloqueado, la UI se "congela" y deja de responder a interacciones del usuario.

Análisis del código

```
import asyncio
import threading

def run_async_task(coro):
    """Ejecuta una corrutina asyncio en un thread separado para no bloquear
Tkinter."""
    loop = asyncio.new_event_loop()

    def start_loop():
        asyncio.set_event_loop(loop)
        loop.run_until_complete(coro)

    threading.Thread(target=start_loop, daemon=True).start()
```

- **Contexto:** Tkinter no es compatible con `asyncio` de forma nativa porque ambos requieren controlar el loop principal de eventos. Tkinter maneja su propio loop de eventos para la UI, y `asyncio` requiere el suyo para la ejecución de coroutines asíncronas.
- **Solución propuesta:** Ejecutar el loop de `asyncio` en un **thread separado** para evitar bloquear el hilo principal de Tkinter.
- `asyncio.new_event_loop()`**:** Se crea un nuevo event loop exclusivo para la tarea asíncrona, evitando interferir con otros loops o el loop principal de la UI.
- **Función** `start_loop`**:**
  - Se asocia el nuevo event loop al thread con `asyncio.set_event_loop(loop)`.
  - Se ejecuta la coroutine completa con `loop.run_until_complete(coro)`, bloqueando el thread pero sin afectar la UI principal.
- **Creación del thread:**
  - Se inicia un `threading.Thread` que ejecuta la función `start_loop`.
  - El thread es **daemon**, lo que significa que no bloqueará la terminación del programa.
  - Así, la coroutine se ejecuta en segundo plano sin congelar la interfaz.

Relación con los requisitos del proyecto

- **Concurrencia y reactividad:** Esta solución demuestra manejo avanzado de concurrencia en la interfaz gráfica, permitiendo ejecutar funciones asíncronas sin bloquear la UI ni perder capacidad de respuesta.

- **Integración con programación reactiva:** El uso combinado de `asyncio` y `threading` permite que la UI reciba resultados asincrónicos (como recomendaciones avanzadas) y actualice la pantalla de forma reactiva.
- **Experiencia de usuario:** Al mantener la interfaz fluida, se mejora la interacción, evitando que la aplicación "se cuelgue" o deje de responder ante tareas prolongadas.

---

4. Ejecución Principal

La ejecución principal de la aplicación carga los animes desde el dataset y permite al usuario interactuar con la interfaz para filtrar y obtener recomendaciones avanzadas. Además, las recomendaciones se procesan en segundo plano.

```
if __name__ == "__main__":
    # Cargar datos de animes
    animes = cargar_animes_desde_csv("anime_recommendations.csv")

    app = AnimeApp(animes)
    app.run()  # Inicia la interfaz gráfica
```

---

5. Conclusiones

Este proyecto representa una integración práctica y efectiva de varios paradigmas de programación dentro de un solo sistema desarrollado en Python, demostrando cómo combinar técnicas modernas para lograr una aplicación robusta y eficiente.

- **Programación Orientada a Objetos (POO):** Se empleó para modelar claramente las entidades del dominio, en este caso los animes, encapsulando atributos y comportamientos relacionados en la clase `Anime`. Este enfoque facilita la reutilización, mantenimiento y extensión del código, además de permitir una representación natural y ordenada de los datos del dominio.

- **Programación Funcional:** Las funciones puras y de alto orden, como filtros, mapeos y reducciones, fueron utilizadas para procesar y manipular listas de animes de manera declarativa y eficiente. Esto resultó en un código más legible, modular y fácil de testear, donde las transformaciones de datos se realizan sin efectos secundarios y de forma concisa.

- **Programación Reactiva y Concurrente:** La integración de funciones asíncronas y la ejecución de tareas en segundo plano garantizan que la interfaz gráfica permanezca responsiva y fluida, incluso al ejecutar procesos que simulan cálculos complejos o esperas externas. Esto mejora significativamente la experiencia del usuario y demuestra un manejo adecuado de la concurrencia en aplicaciones de escritorio con Tkinter.

En conjunto, estos paradigmas no solo permiten cumplir con los requisitos funcionales del recomendador de animes, sino que también generan un código organizado, escalable y preparado para futuras mejoras, como integración con APIs externas o ampliación de funcionalidades. Además, este proyecto sirve como una excelente base educativa para entender cómo distintos estilos de programación pueden coexistir y complementarse en aplicaciones reales.

Links de Videos

- Demo de la App
- Presentacion Tecnica

Link del Repositorio Github

- Final-Multi