U4P09-Gestión de videojuegos

1. Descripción

El objetivo de está práctica es crear un programa en Python que gestione un **catálogo de videojuegos** desde consola, aplicando todas las colecciones vistas en clase.

El proyecto se desarrollará en **equipos de 4 personas,** en **varias etapas,** y el código fuente y las diferentes versiones se gestionarán usando Git y GitHub.

2. Formato de entrega

Será propuesto en clase por el profesor.

3. Trabajo a realizar

3.1. Etapas de trabajo

La práctica se desarrollará en varias etapas y cada una de ella se deberán cumplir los siguientes requisitos. Cada etapa tiene que generar un ejecutable y será una versión del programa.

3.1.1. Etapa 1: Listas

- Crea una lista con los títulos de tus 5 videojuegos favoritos.
- Añade nuevas videojuegos al final de la lista.
- Inserta un videojuego en una posición específica.
- Elimina un videojuego de la lista.
- Recorre la lista e imprime los títulos numerados.
- Muestra la lista ordenada alfabéticamente.

3.1.2. Etapa 2: Tuplas

- Representa cada videojuego como una tupla con la forma: `(título, año, género)`.
- Guarda varias videojuegos en una lista de tuplas.
- Recorre la lista e imprime los datos con un formato claro.
  - Ejemplo: `'The Legend of Zelda (1986) Rol'`.

3.1.3. Etapa 3: Cadenas

- Pide al usuario que escriba el nombre de un videojuego.
- Busca si existe en tu lista de tuplas (ignora mayúsculas y espacios extras).
- Si existe, muestra sus datos con formato: `"Título: The Legend of Zelda | Género: Rol | Año: 1986"`.

- Si no existe, muestra un mensaje adecuado.
- Practica métodos de cadenas (`.lower()`, `.upper()`, `.replace()`, `` `.split()``). .

3.1.4. Etapa 4: Conjuntos

- Crea un conjunto con todos los géneros de tus videojuegos.
- Añade nuevos géneros y elimina otros.
- Pregunta al usuario su género favorito y verifica si está en tu conjunto.
- Crea otro conjunto con los géneros de un amigo y encuentra:
  - Unión: todos los géneros entre ambos.
  - Intersección: géneros en común.
  - Diferencia : géneros que tú tienes y tu amigo no.

3.1.5. Etapa 5: Diccionarios

- Transforma tu colección en un diccionario donde:
  - La clave sea un identificador único (título **normalizado**).
  - El valor sea otro diccionario con datos de la videojuego.

```
{
 "titulo": str,
 "anio": int,
 "genero": set
}
```

- Implementa operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre el catálogo usando la clave.
- Implementa funciones de búsqueda:
  - Exacta por título.
  - Parcial (contenga un fragmento en el título).
  - Por género o rango de años.
- Calcula estadísticas:
  - Número total de videojuegos.
  - Conteo por género.

3.1.6. Etapa 6 : Proyecto final

Crea un **menú interactivo** que permita:

- Listar, buscar, filtrar, añadir, actualizar y eliminar videojuegos.
- Mostrar estadísticas generales.

Usa funciones para organizar el código.

Muestra siempre la información en un formato claro y atractivo.

El proyecto final debe tener la siguiente estructura.

```
gestor-vieojuegos/
├─ main.py                # Programa principal con el menú
├─ src/
│  └─ gestor/
│     ├─ etapas/
│     │  ├─ listas.py       # Ejercicios de la etapa 1
│     │  ├─ tuplas.py       # Ejercicios de la etapa 2
│     │  ├─ cadenas.py      # Ejercicios de la etapa 3
│     │  ├─ conjuntos.py    # Ejercicios de la etapa 4
│     │  └─ diccionarios.py # Ejercicios de la etapa 5
│     │
│     ├─ catalogo.py        # Funciones reales del CRUD de películas
│     ├─ busquedas.py       # Funciones de búsqueda y filtrado
│     ├─ estadisticas.py    # Cálculos de estadísticas
│     ├─ utils_texto.py     # Funciones auxiliares (normalizar texto, etc.)
│     └─ __init__.py
└─ README.md               # Descripción del proyecto y autores
```

- Los ficheros dentro de `etapas/` son el resultado de las etapas intermedias.
- Los módulos fuera de `etapas/` son los que se usan en **main.py** para construir el programa final.
- El README debe indicar:
  - El reparto de roles y tareas.
  - Cómo ejecutar el programa.

3.2. Trabajo en equipo con Git y GitHub

3.2.1. Organización del equipo

Grupo de **4 personas** con estos roles:

- **Integrador**: crea el repositorio, revisa código y aprueba **Pull Requests** y **merges** a `main`.
- **Dev A**: ejemplo: etapa 1, 3 y 6 (funciones de búsqueda).
- **Dev B**: ejemplo: etapas 2, 3, y 6 (funciones CRUD).
- **Dev C (UX/Consola)**: etapa 4 y 6 (menú y formato de salida).

Cada miembro trabaja en **su parte**, y el integrador revisa y aprueba los cambios.

Podéis hacer el reparto como consideréis (que sea equilibrado).

Podéis crear un repositorio de **GitHub con colaboradores** que puedan hacer **Pull Request** directamente si hacer fork del proyecto.

3.2.2. Estructura de ramas

- **main** → rama principal (protegida, no se toca directamente).
- Ramas de trabajo: una por tarea, por ejemplo:
  - **feat/etapa-1-listas**
  - **feat/etapa-3-cadenas**
  - **feat/menu-final**

3.2.3. Flujo de trabajo (GitHub Flow simplificado)

1. Cada miembro trabaja en su rama propia.
2. Hace **commits claros** y **push** al repositorio.
3. Abre un **Pull Request** y lo revisa otro compañero.
4. El **Integrador** aprueba y hace **merge a main**.

Recomendaciones

- Un responsable por archivo para evitar conflictos.
- Commits pequeños** con mensajes claros (`feat:`, `fix:`, `refactor:`).A
- Antes de hacer merge probar el código antes.

3.2.4. Propuesta del Plantilla para Pull Request

```
### Qué hace
- Implementa la búsqueda de videojuegos por título.

### Cómo probar
1. Ejecuta: python main.py
2. Elige la opción 2 del menú.
3. Escribe un título (por ejemplo, "Matrix").
4. Comprueba que muestra los datos correctos.

### Archivos modificados
- src/gestor/busquedas.py
```

3.3. Corrección

- **Organización del equip**o: 2 puntos.
- **Repositorio de GitHub**: Reame, Ramas, Pull Request, … : 2 puntos
- **Código de las etapas:** 1 punto.
- **Código del proyecto final:** 5 puntos.