UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Integrador - Programación 1

Gestión de Datos de Países en Python

Alumno:

Camilo Illanes.

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación 1

Docente Titular

Cinthia Rigoni

1 de Noviembre de 2025

Tabla de contenido

Introducción

Propósito del Informe

Este informe tiene como propósito explicar de forma clara los conceptos de programación que se aplicaron en el desarrollo del proyecto “Gestión de datos de Países”.
A lo largo del documento se presentan las principales estructuras de datos, los mecanismos de control de flujo, las técnicas de procesamiento y las operaciones de manejo de archivos que hacen posible el funcionamiento del programa.

Cada sección incluye una breve explicación teórica y luego muestra cómo se aplica en el código, usando fragmentos reales para ejemplificarlo.
El objetivo final es dejar bien documentada la lógica interna del sistema y las decisiones técnicas que se tomaron para construir una aplicación sólida, clara y fácil de mantener.

Descripción General del Proyecto

El proyecto es una aplicación de consola desarrollada en Python que permite a los usuarios interactuar con un conjunto de datos sobre países, cargados desde un archivo CSV. La aplicación ofrece un menú interactivo con las siguientes funcionalidades:

- Búsqueda de países por nombre.
- Filtrado de países por continente, población y superficie.
- Ordenamiento de los datos según diversos criterios (nombre, población, superficie).
- Cálculo y visualización de estadísticas básicas sobre el conjunto de datos.
- Agregar nuevos países al sistema con validación exhaustiva de datos.
- Editar información de países existentes, con opción de modificar campos individuales o todos a la vez.
- Eliminar países de la base de datos con sistema de doble confirmación.
- Persistencia automática de cambios en el archivo CSV.

El sistema está diseñado de forma modular, separando la lógica de la interfaz de usuario, las operaciones sobre los datos y la lectura de archivos, lo cual facilita su mantenimiento y escalabilidad.

Capítulo 1: Estructuras de Datos Fundamentales

Las estructuras de datos son componentes esenciales en la programación que permiten organizar, gestionar y almacenar datos de manera eficiente. La elección de la estructura de datos adecuada es crucial para el rendimiento y la claridad del código. En este proyecto, las listas y los diccionarios de Python son los pilares sobre los que se construye toda la lógica de manejo de datos.

1.1. Listas

Definición Teórica

Una lista en Python es una estructura de datos que representa una colección ordenada y mutable de elementos. "Ordenada" significa que los elementos mantienen un orden específico y se puede acceder a ellos mediante un índice numérico (empezando desde 0). "Mutable" implica que su contenido puede ser modificado después de su creación; es posible añadir, eliminar o cambiar elementos. Las listas pueden contener elementos de diferentes tipos de datos (enteros, cadenas, otros objetos, e incluso otras listas).

Características Principales

- **Indexación:** Acceso a elementos mediante su posición (***paises[0]***).
- **Mutabilidad:** Permite la modificación de sus elementos (***paises[0] = 'nuevo_valor'***).
- **Dinamismo:** Pueden crecer y decrecer en tamaño mediante operaciones como ***append()*** (añadir al final) o ***pop()*** (eliminar).
- **Variedad de tipos:** Pueden contener elementos de distintos tipos de datos.

Aplicación en el Proyecto

En el proyecto, la lista es la estructura de datos principal para almacenar el conjunto completo de países cargados desde el archivo CSV. La variable ***paises***, inicializada en ***main.py*** y utilizada en casi todas las funciones, es una lista.

Ejemplo de creación y carga *(lectura.py):*

```python
# Leer csv
def leer_csv(ruta_archivo):
    paises = []

    try:
        # Si no encuentra el archivo en la ruta
        if not os.path.exists(ruta_archivo):
            raise FileNotFoundError(f"No se encontró el archivo: {ruta_archivo}")

        with open(ruta_archivo, encoding='utf-8') as archivo:
            lector = csv.DictReader(archivo)

            for numero_fila, fila in enumerate(lector, start=2):  # start=2 porque la fila 1 es el encabezado
                try:
                    pais = {
                        "nombre": fila["nombre"].strip(),
                        "poblacion": int(fila["poblacion"]),
                        "superficie": int(fila["superficie"]),
                        "continente": fila["continente"].strip()
                    }
                    paises.append(pais)
                except KeyError as e:
                    print(f"\nAVISO: Falta el campo {e} en la fila {numero_fila}")
                except ValueError as e:
                    print(f"\nAVISO: Valor inválido en la fila {numero_fila} - {e}")

        print(f"Se cargaron {len(paises)} países correctamente")

    except (FileNotFoundError, PermissionError) as e:
        print(f"\nAVISO: {e}")
    except Exception as e:
        print(f"\nAVISO: {e}")

    return paises
```

En este fragmento, ***paises = []*** crea una lista vacía. Dentro del bucle, cada país (representado como un diccionario) es añadido a esta lista mediante el método ***paises.append(pais)***. El resultado es una lista donde cada elemento es un diccionario que contiene la información de un país. Esta lista ***paises*** es luego pasada como argumento a las funciones de filtrado, ordenamiento y estadísticas, que iteran sobre ella para realizar sus operaciones.

Ejemplo de iteración *(de filtros.py)*:

```python
# Busca países por coincidencia parcial o exacta
def buscar_pais(paises, nombre):
    try:
        if not isinstance(paises, list):
            raise TypeError("El parámetro 'paises' debe ser una lista")
        if not isinstance(nombre, str):
            raise TypeError("El parámetro 'nombre' debe ser un string")

        # Se utiliza una "list comprehension" para iterar sobre la lista y filtrar
        resultados = [pais for pais in paises if quitar_tildes(nombre.lower()) in quitar_tildes(pais["nombre"].lower())]
        return resultados

    except KeyError:
        print("\nAVISO: Algunos países no tienen el campo 'nombre'")
        return []
    except (TypeError, AttributeError) as e:
        print(f"\nAVISO: {e}")
        return []
```

Este uso demuestra cómo la lista ***paises*** es recorrida para generar una nueva lista (***resultados***) que contiene sólo los elementos que cumplen una condición específica. La elección de una lista es ideal aquí por su capacidad para almacenar una colección de tamaño variable de objetos (países) y por la eficiencia con la que se puede iterar sobre sus elementos.

1.2. Diccionarios

Definición Teórica

Un diccionario en Python es una estructura de datos que almacena una colección no ordenada de elementos en un formato de ***clave: valor***. A diferencia de las listas, que se indexan por posición numérica, los diccionarios se indexan mediante claves únicas. Son mutables y extremadamente eficientes para la recuperación de datos cuando la clave es conocida.

Estructura Clave-Valor

- **Clave (Key):** Debe ser un objeto inmutable (como una cadena, número o tupla). Cada clave en un diccionario debe ser única.
- **Valor (Value):** Puede ser cualquier objeto de Python (un número, una cadena, una lista, otro diccionario, etc.).

Esta estructura es ideal para representar objetos del mundo real que tienen propiedades asociadas, como un país, que tiene un nombre, una población, una superficie, etc.

Aplicación en el Proyecto

Los diccionarios son utilizados para representar a cada país individual. Cada fila del archivo CSV se convierte en un diccionario donde los encabezados de las columnas (***"nombre", "poblacion", etc.***) actúan como claves y los datos de la fila actúan como sus valores correspondientes.

Ejemplo de creación (de lectura.py):

```python
for numero_fila, fila in enumerate(lector, start=2):  # start=2 porque la fila 1 es el encabezado
    try:
        pais = {
            "nombre": fila["nombre"].strip(),
            "poblacion": int(fila["poblacion"]),
            "superficie": int(fila["superficie"]),
            "continente": fila["continente"].strip()
        }
        paises.append(pais)
    except KeyError as e:
        print(f"\nAVISO: Falta el campo {e} en la fila {numero_fila}")
    except ValueError as e:
        print(f"\nAVISO: Valor inválido en la fila {numero_fila} - {e}")
```

Este código muestra cómo se construye un diccionario ***pais*** para cada fila leída. Las claves son cadenas de texto (***"nombre", "poblacion"***) y los valores se extraen del objeto ***fila*** proporcionado por ***csv.DictReader***.

Ejemplo de acceso a datos (de ordenar.py):

```python
# Ordena países por una clave dada (nombre, población, superficie)
def ordenar_paises(paises, clave, descendente=False):
    try:
        # Verificar que paises sea una lista
        if not isinstance(paises, list):
            raise TypeError("El parámetro 'paises' debe ser una lista")

        # Verificar que la lista no esté vacía
        if not paises:
            print("Advertencia: La lista de países está vacía")
            return []

        # Verificar que clave sea un string
        if not isinstance(clave, str):
            raise TypeError("El parámetro 'clave' debe ser un string")

        # Verificar que clave no esté vacía
        if not clave.strip():
            raise ValueError("El parámetro 'clave' no puede estar vacío")

        # Verificar que descendente sea booleano
        if not isinstance(descendente, bool):
            raise TypeError("El parámetro 'descendente' debe ser True o False")

        # Verificar que la clave existe en todos los países
        claves_validas = {"nombre", "poblacion", "superficie", "continente"}
        if clave not in claves_validas:
            raise ValueError(f"Clave inválida. Use una de: {', '.join(claves_validas)}")
```

```python
        # Verificar que todos los países tengan la clave
        for i, pais in enumerate(paises):
            if not isinstance(pais, dict):
                raise TypeError(f"El elemento en la posición {i} no es un diccionario")

            if clave not in pais:
                raise KeyError(f"El país en la posición {i} no tiene la clave '{clave}'")

        # Ordenar los países
        paises_ordenados = sorted(paises, key=lambda x: x[clave], reverse=descendente)

        return paises_ordenados

    except (TypeError, ValueError, KeyError, Exception) as e:
        print(f"\nAviso: {e}")
        return []
```

Aquí, la función ***lambda x: x[clave]*** accede al valor asociado a la ***clave*** proporcionada (por ejemplo, ***"poblacion"***) dentro de cada diccionario x (que representa un país). El acceso ***x['poblacion']*** es una operación muy rápida, lo que hace que los diccionarios sean perfectos para esta tarea. Si se hubiera usado una lista de listas, el acceso habría sido por índice (***x[1]***), lo cual es menos legible y más propenso a errores si el orden de los datos cambia.

Capítulo 2: Modularidad y Control de Flujo

La modularidad se refiere a la práctica de dividir un programa en módulos o componentes separados e intercambiables, mientras que el control de flujo se refiere al orden en que se ejecutan las instrucciones del programa. Ambos son cruciales para crear software legible, mantenible y robusto.

2.1. Funciones

Definición y Propósito

Una función es un bloque de código reutilizable que realiza una tarea específica. Las funciones ayudan a organizar el código, evitar la repetición (principio DRY - Don't Repeat Yourself) y hacer que los programas sean más modulares. Una función bien diseñada toma ciertas entradas (parámetros), realiza una serie de operaciones y, opcionalmente, devuelve un resultado.

Parámetros y Valores de Retorno

- **Parámetros (o argumentos):** Son los valores que se le pasan a una función para que pueda operar. Permiten que una misma función se comporte de manera diferente según la entrada que reciba.
- **Valores de Retorno:** Una función puede devolver un valor al código que la llamó usando la palabra clave ***return***. Si no se especifica un ***return***, la función devuelve ***None*** por defecto.

Aplicación en el Proyecto

El proyecto está fuertemente basado en funciones, lo que refleja una buena práctica de diseño modular. Cada archivo ***(lectura.py, filtros.py, ordenar.py, estadistica.py, menu.py, etc.)*** contiene funciones con responsabilidades bien definidas.

Ejemplo de una función de procesamiento (de filtros.py):

```python
#Filtra paises por continente
def filtrar_por_continente(paises, continente):
    try:
        if not isinstance(paises, list):
            raise TypeError("El parámetro 'paises' debe ser una lista")
        if not isinstance(continente, str):
            raise TypeError("El parámetro 'continente' debe ser un string")

        return [pais for pais in paises if quitar_tildes(pais["continente"].lower()) == quitar_tildes(continente.lower())]

    except KeyError:
        print("\nAVISO: Algunos países no tienen el campo 'continente'")
        return []

    except (TypeError, AttributeError) as e:
        print(f"\nAVISO: {e}")
        return []
```

Esta función ***filtrar_por_continente()*** es un excelente ejemplo:

- **Tiene una única responsabilidad:** Filtrar una lista de países por un continente específico.
- **Acepta parámetros:** ***paises*** (la lista sobre la que operar) y ***continente*** (el criterio de filtrado). Esto la hace genérica y reutilizable.
- **Devuelve un valor:** Retorna una nueva lista con los resultados del filtro, o una lista vacía si no hay coincidencias o si ocurre un error.
- **Incluye manejo de errores:** Utiliza un bloque ***try-except*** para gestionar posibles problemas, como un tipo de dato incorrecto o un diccionario sin la clave esperada.

2.2. Estructuras Condicionales

Definición Teórica

Las estructuras condicionales permiten que un programa ejecute diferentes bloques de código según si se cumple o no una determinada condición booleana (***True*** o ***False***). Son la base para la toma de decisiones y la lógica en la programación.

La Sentencia if-else

La forma más común de condicional. El bloque de código dentro del ***if*** se ejecuta si la condición es verdadera. Opcionalmente, se pueden añadir bloques ***elif*** (else if) para

comprobar condiciones adicionales, y un bloque ***else*** que se ejecuta si ninguna de las condiciones anteriores es verdadera.

La Sentencia match-case

Introducida en Python 3.10, la sentencia ***match-case*** es una alternativa más estructurada y legible al ***if-elif-else*** para casos donde se compara una variable con múltiples valores posibles. Evalúa una expresión y ejecuta el bloque de código del primer ***case*** que coincida.

Aplicación en el Proyecto

Los condicionales son omnipresentes en el proyecto para la validación de entradas, el control de la lógica del menú y la presentación de resultados.

Ejemplo de match-case para el menú principal (de main.py):

```python
def main():
    # Cargar datos del archivo CSV
    paises = leer_csv("csv/paises_mundo.csv")

    if not paises:
        print("No se pudieron cargar datos. Revisa el archivo CSV.")
        return

    # Bucle principal del menú
    while True:
        try:
            # Mostrar menú y obtener opción
            opcion = mostrar_menu()
            opcion_int = int(opcion) if opcion.isdigit() else 0

            # Ejecutar opción seleccionada
            match opcion_int:
                case 1:
                    opcion_buscar_pais(paises)

                case 2:
                    opcion_filtrar_continente(paises)

                case 3:
                    opcion_filtrar_poblacion(paises)

                case 4:
                    opcion_filtrar_superficie(paises)

                case 5:
                    opcion_ordenar_paises(paises)
```

```
            case 6:
                opcion_mostrar_estadisticas(paises)

            case 7:
                print("\n¡Gracias por usar el programa!")
                print("Saliendo ... ")
                break

            case _:
                print('\nLa opción ingresada no es válida')

        except ValueError as e:
            print(f"Error: {e}")
        except KeyboardInterrupt:
            print("\n\nPrograma interrumpido por el usuario (Ctrl+C)")
            print("Saliendo ... ")
            break
        except Exception as e:
            print(f"Error inesperado: {e}")
```

El ***match*** evalúa el valor de ***opcion_int***. Si es ***1***, se ejecuta el ***case 1*** y se llama a ***opcion_buscar_pais***. Si el valor no coincide con ninguno de los casos numéricos, se ejecuta el caso por defecto (***case*** _), informando al usuario de una opción inválida. Esta estructura es mucho más limpia que una cadena larga de ***if-elif-elif***.

Ejemplo de if para validación y presentación (de opciones_menu.py):

```
def opcion_buscar_pais(paises):
    # Opcion 1: Búsqueda parcial con nombre
    while True:
        try:
            nombre = input('Ingrese el nombre del país: ').strip()

            if not nombre:
                raise ValueError('El nombre no puede estar vacío')
            elif any(caracter.isdigit() for caracter in nombre):
                raise TypeError('El nombre no puede contener números')

            resultados = buscar_pais(paises, nombre)

            if resultados:
                mostrar_con_paginacion(resultados, titulo=f'Países que coinciden con "{nombre}"')
            else:
                print('No se encontraron coincidencias.')

        except (ValueError, TypeError) as e:
            print(f'\nAVISO: {e}. Intente nuevamente.\n')
            continue

        if not preguntar_si_no('\n¿Buscar otro país? (s/n): '):
            break
```

Aquí, ***if resultados:*** es una forma común en Python de comprobar si una lista (o cualquier colección) tiene elementos. Si ***resultados*** contiene al menos un país, la condición es ***True*** y se muestran los resultados. De lo contrario, el bloque ***else*** se ejecuta para informar al usuario.

Capítulo 3: Procesamiento y Análisis de Datos

Una vez que los datos están cargados y estructurados, el siguiente paso es procesarlos para extraer información útil. En este proyecto, esto se manifiesta a través de operaciones de ordenamiento y el cálculo de estadísticas básicas.

3.1. Algoritmos de Ordenamiento

Concepto de Ordenamiento

El ordenamiento es el proceso de organizar los elementos de una colección (como una lista) en un orden específico, ya sea ascendente o descendente. Es una operación fundamental en la informática, ya que los datos ordenados son mucho más fáciles de consultar y analizar por los humanos y por otros algoritmos.

El Método sorted() en Python

Python proporciona una función incorporada, ***sorted()***, que toma un iterable (como una lista) y devuelve una nueva lista con los elementos ordenados. Es una función muy potente que puede ser personalizada con dos argumentos clave:

- **key**: Una función que se aplica a cada elemento antes de la comparación. Permite ordenar estructuras complejas (como una lista de diccionarios) basándose en uno de sus atributos.
- **reverse**: Un valor booleano. Si se establece en ***True***, el ordenamiento se realiza de forma descendente.

Internamente, ***sorted()*** utiliza un algoritmo llamado Timsort, que es una mezcla híbrida y eficiente de los algoritmos ***Merge Sort*** e ***Insertion Sort***, optimizado para muchos tipos de datos del mundo real.

Aplicación en el Proyecto

El ordenamiento se implementa en la función ***ordenar_paises()*** del módulo ***ordenar.py***, que es llamada por ***opcion_ordenar_paises()***.

Ejemplo de la función de ordenamiento (de ordenar.py):

```python
# Ordena países por una clave dada (nombre, población, superficie)
def ordenar_paises(paises, clave, descendente=False):
    try:
        # Verificar que paises sea una lista
        if not isinstance(paises, list):
            raise TypeError("El parámetro 'paises' debe ser una lista")

        # Verificar que la lista no esté vacía
        if not paises:
            print("Advertencia: La lista de países está vacía")
            return []

        # Verificar que clave sea un string
        if not isinstance(clave, str):
            raise TypeError("El parámetro 'clave' debe ser un string")

        # Verificar que clave no esté vacía
        if not clave.strip():
            raise ValueError("El parámetro 'clave' no puede estar vacío")

        # Verificar que descendente sea booleano
        if not isinstance(descendente, bool):
            raise TypeError("El parámetro 'descendente' debe ser True o False")

        # Verificar que la clave existe en todos los países
        claves_validas = {"nombre", "poblacion", "superficie", "continente"}
        if clave not in claves_validas:
            raise ValueError(f"Clave inválida. Use una de: {', '.join(claves_validas)}")
```

```python
        # Verificar que todos los países tengan la clave
        for i, pais in enumerate(paises):
            if not isinstance(pais, dict):
                raise TypeError(f"El elemento en la posición {i} no es un diccionario")

            if clave not in pais:
                raise KeyError(f"El país en la posición {i} no tiene la clave '{clave}'")

        # Ordenar los países
        paises_ordenados = sorted(paises, key=lambda x: x[clave], reverse=descendente)

        return paises_ordenados

    except (TypeError, ValueError, KeyError, Exception) as e:
        print(f"\nAviso: {e}")
        return []
```

El núcleo de esta función es la línea: ***paises_ordenados = sorted(paises, key=lambda x: x[clave], reverse=descendente)***

- **paises**: Es la lista de diccionarios que se va a ordenar.
- **key=lambda x: x[clave]**: Aquí es donde ocurre la magia. Por cada diccionario ***x*** en la lista ***paises***, la función ***lambda*** extrae el valor asociado a la ***clave*** especificada (que puede ser ***"nombre"***, ***"poblacion"*** o ***"superficie"***). ***sorted()*** utiliza estos valores extraídos para realizar las comparaciones y ordenar los diccionarios originales.

- **reverse=descendente**: Este parámetro controla si el orden es ascendente (***False***) o descendente (***True***), según la entrada del usuario.

Esta implementación es robusta, flexible y eficiente, delegando la complejidad del algoritmo de ordenamiento a la función ***sorted()*** optimizada de Python.

3.2. Estadísticas Básicas

Relevancia del Análisis Estadístico

El análisis estadístico permite resumir un gran conjunto de datos en unas pocas cifras clave o métricas. Estas métricas, como los promedios, máximos y mínimos, proporcionan una visión general rápida y comprensible de las características principales de los datos sin necesidad de examinar cada registro individualmente.

Métricas Utilizadas

En el proyecto se calculan las siguientes estadísticas:

- **Máximo y Mínimo:** Identificar los países con la mayor y menor población.
- **Promedio (Media Aritmética):** Calcular la población y superficie promedio de todos los países.
- **Frecuencia:** Contar cuántos países pertenecen a cada continente.

Aplicación en el Proyecto

Toda la lógica estadística está encapsulada en la función ***mostrar_estadisticas()*** del módulo ***estadistica.py***.

Ejemplo de cálculos estadísticos (de estadistica.py):

```python
# Muestra estadísticas básicas de los países
def mostrar_estadisticas(paises):
    try:
        # Validación inicial
        if not paises:
            print('No hay datos cargados.')
            return

        if not isinstance(paises, list):
            raise TypeError('El parámetro "paises" debe ser una lista')

        # Cálculo de país con mayor y menor población
        pais_mayor_pob = max(paises, key=lambda x: x['poblacion'])
        pais_menor_pob = min(paises, key=lambda x: x['poblacion'])

        # Cálculo de promedios
        promedio_pob = sum(pais['poblacion'] for pais in paises) / len(paises)
        promedio_sup = sum(pais['superficie'] for pais in paises) / len(paises)

        # Mostrar estadísticas básicas
        print('\n==== ESTADÍSTICAS ==== ')
        print(f'- País con mayor población  : {pais_mayor_pob["nombre"]} ({pais_mayor_pob["poblacion"]:,})')
        print(f'- País con menor población  : {pais_menor_pob["nombre"]} ({pais_menor_pob["poblacion"]:,})')
        print(f'- Promedio de población     : {promedio_pob:,.0f}')
        print(f'- Promedio de superficie    : {promedio_sup:,.0f}')
        # Cantidad de países por continente
        continentes = {}
        for pais in paises:
            cont = pais['continente']
            continentes[cont] = continentes.get(cont, 0) + 1

        print('\nCantidad de países por continente:')
        for cont, cant in continentes.items():
            print(f'  - {cont}: {cant}')

    except KeyError as e:
        print(f'\nAVISO: Algunos países no tienen el campo {e}')
    except (TypeError, ValueError) as e:
        print(f'\nAVISO: {e}')
    except ZeroDivisionError:
        print('\nAVISO: No hay países para calcular promedios')
    except Exception as e:
        print(f'\nAVISO: {e}')
```

Este código demuestra el uso de funciones incorporadas de Python para realizar cálculos estadísticos de manera concisa:

- **max(paises, key=lambda x: x['poblacion'])**: Encuentra el diccionario (país) completo que tiene el valor máximo para la clave ***'poblacion'***. ***min()*** funciona de manera análoga.
- **sum(pais['poblacion'] for pais in paises)**: Utiliza una expresión generadora para sumar las poblaciones de todos los países.

- **len(paises)**: Obtiene el número total de países, que se utiliza como divisor para calcular el promedio.
- El bucle final para contar países por continente es un patrón común para calcular frecuencias: utiliza un diccionario (***continentes***) para almacenar los conteos, empleando el método ***.get(clave, 0)*** para manejar la primera vez que se encuentra un continente.

Capítulo 4: Manejo de Archivos y Datos Externos

Los programas a menudo necesitan interactuar con datos almacenados fuera del propio código, en archivos. La capacidad de leer y escribir en diferentes formatos de archivo es una habilidad esencial en el desarrollo de software.

4.1. Archivos CSV

Definición del Formato

CSV es un formato de archivo de texto plano que se utiliza para almacenar datos tabulares (es decir, datos organizados en una tabla con filas y columnas). Cada línea del archivo corresponde a una fila de la tabla, y las columnas se separan por un delimitador, comúnmente una coma (,). La primera línea suele ser un encabezado que nombra cada columna. Es un formato muy popular para el intercambio de datos debido a su simplicidad y compatibilidad con una amplia gama de software, como hojas de cálculo y bases de datos.

El archivo ***paises_mundo.csv*** del proyecto sigue esta estructura, utilizando la coma como delimitador.

El Módulo csv de Python

Python incluye un módulo en su biblioteca estándar llamado ***csv*** que simplifica enormemente el trabajo con archivos CSV. Proporciona herramientas para leer y escribir datos en este formato sin tener que manejar manualmente la separación de comas y las comillas.

Una de las clases más útiles de este módulo es ***csv.DictReader***. En lugar de devolver cada fila como una lista de cadenas, ***DictReader*** la devuelve como un diccionario, utilizando la primera fila del archivo como las claves. Esto hace que el código sea mucho más legible y robusto, ya que se accede a los datos por nombre de columna en lugar de por posición.

Aplicación en el Proyecto: Lectura y Parseo

La lectura del archivo CSV es la primera operación que realiza el programa y está contenida en la función ***leer_csv()*** del módulo ***lectura.py***.

Ejemplo de lectura de CSV (de lectura.py):

```python
# Leer csv
def leer_csv(ruta_archivo):
    paises = []

    try:
        # Si no encuentra el archivo en la ruta
        if not os.path.exists(ruta_archivo):
            raise FileNotFoundError(f"No se encontró el archivo: {ruta_archivo}")

        with open(ruta_archivo, encoding='utf-8') as archivo:
            lector = csv.DictReader(archivo)

            for numero_fila, fila in enumerate(lector, start=2):  # start=2 porque la fila 1 es el encabezado
                try:
                    pais = {
                        "nombre": fila["nombre"].strip(),
                        "poblacion": int(fila["poblacion"]),
                        "superficie": int(fila["superficie"]),
                        "continente": fila["continente"].strip()
                    }
                    paises.append(pais)
                except KeyError as e:
                    print(f"\nAVISO: Falta el campo {e} en la fila {numero_fila}")
                except ValueError as e:
                    print(f"\nAVISO: Valor inválido en la fila {numero_fila} - {e}")

        print(f"Se cargaron {len(paises)} países correctamente")

    except (FileNotFoundError, PermissionError) as e:
        print(f"\nAVISO: {e}")
    except Exception as e:
        print(f"\nAVISO: {e}")

    return paises
```

Análisis del proceso:

- **Apertura del archivo:** ***with open(...)*** abre el archivo de forma segura, garantizando que se cierre automáticamente al finalizar. ***encoding='utf-8'*** es importante para manejar correctamente caracteres especiales y tildes.
- **Creación del lector:** ***lector = csv.DictReader(archivo)*** crea un objeto ***DictReader*** que interpretará el archivo. Automáticamente usará la primera fila como encabezado para las claves de los diccionarios que genere.

- **Iteración y parseo:** El bucle ***for fila in lector:*** itera sobre cada línea del CSV (excepto el encabezado). En cada iteración, ***fila*** es un diccionario donde las claves son los nombres de las columnas (***"nombre"***, ***"poblacion"***, etc.) y los valores son cadenas de texto.
- **Conversión de tipos:** El código convierte explícitamente los valores de población y superficie a enteros usando ***int()***, ya que los datos leídos de un archivo de texto son siempre cadenas. También utiliza ***.strip()*** para eliminar posibles espacios en blanco al principio o al final de las cadenas.
- **Manejo de errores:** El código está envuelto en bloques ***try-except*** para manejar problemas comunes como que el archivo no exista (***FileNotFoundError***) o que un valor numérico no pueda ser convertido (***ValueError***).

Escritura de Archivos CSV

El módulo ***csv*** también proporciona herramientas para escribir datos en archivos CSV. La clase ***csv.DictWriter*** es el complemento de ***DictReader*** para operaciones de escritura.

Modos de apertura de archivos:

- **'r' (read):** Lectura. El archivo debe existir.
- **'w' (write):** Escritura. Crea un archivo nuevo o sobrescribe uno existente.
- **'a' (append):** Añadir al final. Crea el archivo si no existe.

El proyecto utiliza dos estrategias de escritura:

1. Modo append para agregar nuevos países sin modificar los existentes.

```
with open(ruta_archivo, mode='a', encoding='utf-8', newline='') as archivo:
    campos = ['nombre', 'poblacion', 'superficie', 'continente']
    escribir = csv.DictWriter(archivo, fieldnames=campos)
    escribir.writerow(pais)
return True
```

2. Modo write para reescribir completamente el archivo al editar o eliminar.

```
with open(ruta_archivo, mode='w', newline='', encoding='utf-8') as archivo:
    campos = ['nombre', 'poblacion', 'superficie', 'continente']
    escribir = csv.DictWriter(archivo, fieldnames=campos)
    escribir.writeheader()
    escribir.writerows(paises)
return True
```

Capítulo 5: Flujo de Operaciones del Programa

5.1. Descripción del Flujo Principal

El flujo de operaciones del programa sigue un patrón de "Bucle de Lectura-Evaluación-Impresión" (REPL - Read-Eval-Print Loop), común en aplicaciones de consola interactivas.

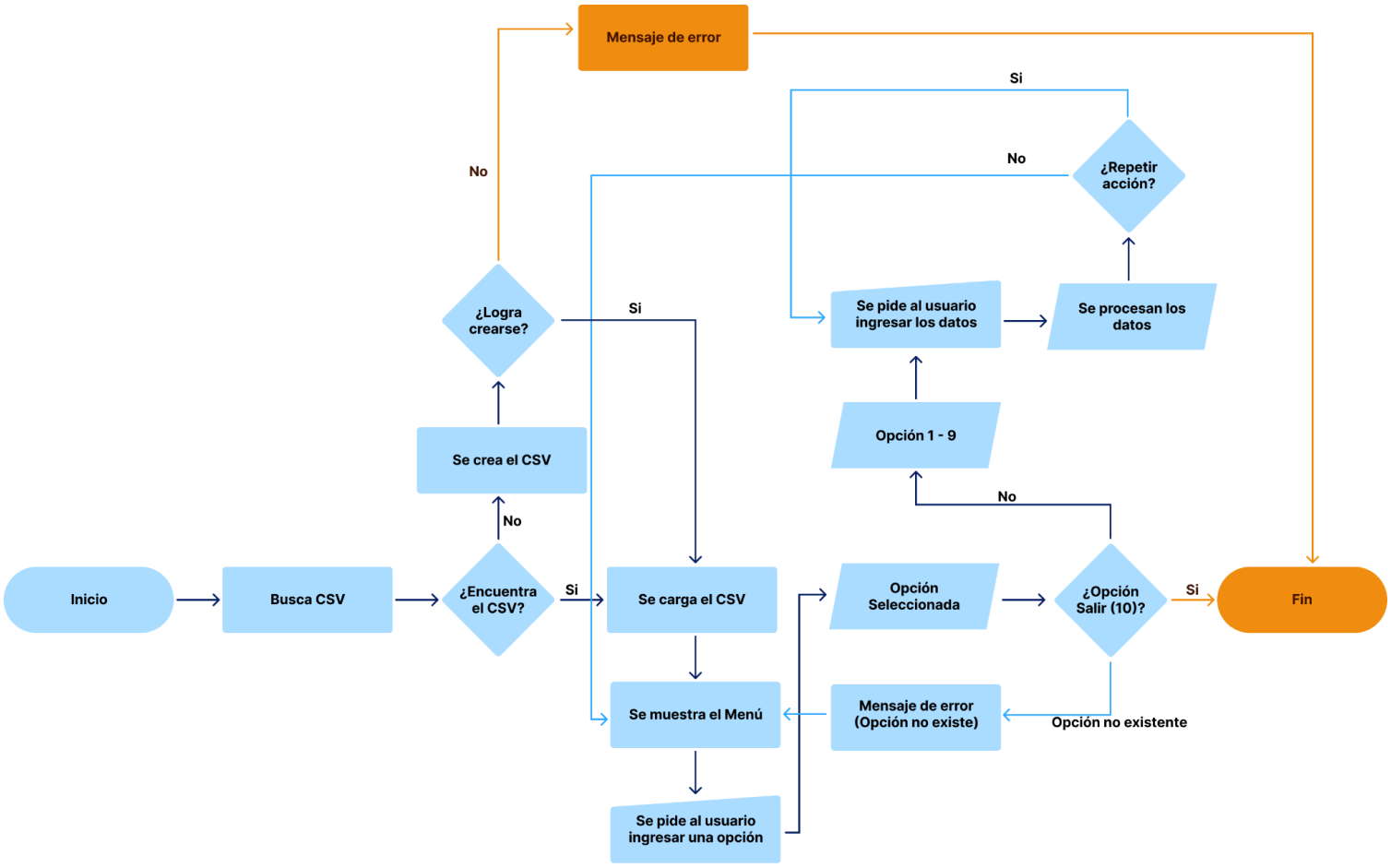
1. **Inicialización:** Al ejecutar ***main.py***, la función ***main()*** se pone en marcha. Su primera tarea es llamar a ***leer_csv()*** para cargar los datos del archivo ***paises_mundo.csv*** en la lista ***paises***. Si la carga falla, el programa termina con un mensaje de error.
2. **Bucle Principal:** El programa entra en un bucle infinito ***(while True:)***. Este bucle mantiene la aplicación en ejecución hasta que el usuario decide salir.
3. **Mostrar Menú (Imprimir):** En cada iteración del bucle, se llama a ***mostrar_menu()***, que imprime en la consola las opciones disponibles para el usuario.
4. **Capturar Entrada (Leer):** La función ***mostrar_menu()*** también captura la entrada del teclado del usuario (***input()***) y la devuelve a la función ***main()***.
5. **Procesar Opción (Evaluar):** La entrada del usuario se convierte a un número entero. Una estructura ***match-case*** evalúa este número y determina qué acción realizar.
6. **Ejecutar Acción:** Se llama a la función correspondiente del módulo ***opciones_menu.py*** (ej: ***opcion_buscar_pais()***, ***opcion_ordenar_paises()***, etc.), pasándole la lista ***paises*** como argumento.
7. **Operaciones de Modificación (Opcional):** Si el usuario elige una opción que modifica datos (agregar, editar o eliminar), se ejecutan pasos adicionales:
    a. Validación exhaustiva de la entrada del usuario.
    b. Búsqueda del registro (para editar/eliminar).
    c. Confirmación explícita antes de aplicar cambios.
    d. Modificación de la lista en memoria.
    e. Persistencia en el archivo CSV.
    f. Reversión automática si falla el guardado.
    g. Retroalimentación al usuario sobre el resultado.
8. **Lógica de la Opción:** La función específica maneja la interacción con el usuario (pide más datos, como un nombre de país o un rango de población), llama a las funciones de

procesamiento de datos necesarias (***buscar_pais()***, ***ordenar_paises()***, etc.) y muestra los resultados, a menudo utilizando el paginador ***mostrar_con_paginacion()***.

9. **Retorno al Bucle:** Una vez que la función de la opción termina, el control vuelve al inicio del ***while True:*** y el menú se muestra de nuevo.

10. **Condición de Salida:** Si el usuario elige la opción "Salir" (caso 7), la sentencia ***break*** rompe el bucle ***while True:***, y el programa finaliza su ejecución mostrando un mensaje de despedida.

5.2 Diagrama de Flujo



Capítulo 6: Operaciones CRUD y Persistencia de Datos

6.1. Operaciones de Creación

Definición Teórica

CRUD es un acrónimo que representa las cuatro operaciones básicas de persistencia de datos: ***Create*** (Crear), ***Read*** (Leer), ***Update*** (Actualizar) y ***Delete*** (Eliminar). Estas operaciones constituyen la base de cualquier sistema de gestión de bases de datos o aplicación que maneje información persistente.

La operación de creación permite agregar nuevos registros al conjunto de datos, expandiendo la información almacenada. En el contexto de este proyecto, significa agregar un nuevo país con toda su información asociada (nombre, población, superficie y continente) tanto a la estructura de datos en memoria como al archivo CSV persistente.

Validación de Datos de Entrada

Antes de agregar un nuevo registro, es crítico validar que los datos cumplan con los requisitos del sistema:

- **Validación de tipos:** Asegurar que los campos numéricos sean efectivamente números.
- **Validación de rango:** Verificar que los valores estén dentro de rangos lógicos (ej: población > 0).
- **Validación de unicidad:** Evitar duplicados verificando que el registro no exista ya**.**
- **Validación de formato:** Comprobar que los datos cumplan con el formato esperado.

Aplicación en el Proyecto

La funcionalidad de agregar países está implementada en la función ***opcion_agregar_pais()*** del módulo ***opciones_menu.py***, que coordina la interacción con el usuario, y en ***agregar_pais()*** del módulo ***edicion.py***, que maneja la persistencia.

Ejemplo de validación y creación (opciones_menu.py):

```
nombre = input('Ingrese nombre del país: ').strip().lower()

if not nombre:
    raise ValueError('El nombre no puede estar vacío')

if any(caracter.isdigit() for caracter in nombre):
    raise TypeError('El nombre no puede contener números')

if any(quitar_tildes(pais['nombre'].lower()) == quitar_tildes(nombre) for pais in paises):
    raise ValueError(f'El país "{nombre.capitalize()}" ya existe!')

try:
    poblacion = int(input('Ingrese la población del país: ').strip())

    if poblacion ≤ 0:
        raise ValueError('La población debe ser mayor a 0')

except ValueError as e:
    if 'invalid literal' in str(e):
        raise ValueError('La población debe ser un número entero')
    raise

try:
    superficie = int(input('Ingrese la superficie del país (km²): ').strip())

    if superficie ≤ 0:
        raise ValueError('La superficie debe ser mayor a 0')

except ValueError as e:
    if 'invalid literal' in str(e):
        raise ValueError('La superficie debe ser un número entero')
    raise
```

Este fragmento demuestra múltiples capas de validación:

- **Validación de vacío**: ***if not nombre*** verifica que el campo no esté en blanco.
- **Validación de contenido**: ***any(caracter.isdigit() for caracter in nombre)*** rechaza nombres con números.
- **Validación de duplicados**: La expresión generadora recorre todos los países existentes, normalizando tanto el nombre ingresado como los almacenados mediante ***quitar_tildes()*** para comparación insensible a tildes y mayúsculas.
- **Validación numérica**: El bloque ***try-except*** captura errores de conversión a entero y valida que el valor sea positivo.

Ejemplo de persistencia (edicion.py):

```
def agregar_pais(paises, pais, ruta_archivo):
    try:
        if not isinstance(paises, list):
            raise TypeError('El elemento no es una lista')
        if not isinstance(pais, dict):
            raise TypeError('El país debe ser un diccionario')
        if guardar_pais(ruta_archivo, pais):
            paises.append(pais)
            return True
        return False

    except (ValueError, TypeError) as e:
        print(f"\nAVISO: {e}")
        return False
```

```
def guardar_pais(ruta_archivo, pais):
    try:
        if not os.path.exists(ruta_archivo):
            raise FileNotFoundError(f"No se encontró el archivo: {ruta_archivo}")

        with open(ruta_archivo, mode='a', encoding='utf-8', newline='') as archivo:
            campos = ['nombre', 'poblacion', 'superficie', 'continente']
            escribir = csv.DictWriter(archivo, fieldnames=campos)
            escribir.writerow(pais)
        return True

    except (FileNotFoundError, PermissionError, OSError) as e:
        print(f"\nAVISO: {e}")
        return False

    except Exception as e:
        print(f"\nAVISO: Error al guardar en CSV - {e}")
        return False
```

La función ***guardar_pais()*** utiliza ***csv.DictWriter*** en modo append ***('a')*** para agregar el nuevo registro al final del archivo sin modificar los existentes. Solo si la escritura en disco es exitosa se agrega el país a la lista en memoria, asegurando consistencia entre ambas representaciones.

Confirmación del Usuario

Antes de ejecutar la operación, el sistema muestra un resumen completo de los datos que se van a agregar:

```
print('\n' + '-'*50)
print('Resumen del país que vas a agregar:')
print('-'*50)
print(f'  Nombre      : {nombre.title()}')
print(f'  Población   : {poblacion:,} habitantes')
print(f'  Superficie  : {superficie:,} km²')
print(f'  Continente  : {continentes[indice]}')
print('-'*50 + '\n')
```

```
if not preguntar_si_no('¿Confirma que desea agregar este país? (s/n): '):
    print('\nOperación cancelada.')
    break
```

Este patrón de "resumen y confirmación" es una práctica de UX que previene errores y da al usuario la oportunidad de revisar antes de confirmar.

6.2. Operaciones de Actualización

Definición Teórica

La operación de actualización permite modificar los valores de un registro existente sin eliminarlo. Es fundamental en sistemas donde los datos pueden cambiar con el tiempo (por ejemplo, la población de un país que se actualiza con nuevos censos).

Existen dos estrategias principales de actualización:

- **Actualización completa**: Reemplazar todos los campos del registro.
- **Actualización parcial**: Modificar solo los campos específicos que necesitan cambios.

El proyecto implementa ambas estrategias, dando al usuario la flexibilidad de elegir.

Búsqueda Exacta con Normalización

Para editar un registro, primero debe localizarse. El proyecto utiliza búsqueda exacta pero con normalización de texto, permitiendo flexibilidad en la entrada del usuario.

Ejemplo de búsqueda exacta (filtros.py):

```python
def buscar_exacto(paises, nombre):
    try:
        if not isinstance(paises, list):
            raise TypeError("El parámetro 'paises' debe ser una lista")
        if not isinstance(nombre, str):
            raise TypeError("El parámetro 'nombre' debe ser un string")

        # Buscar coincidencia exacta
        for pais in paises:
            if quitar_tildes(nombre.lower().strip()) == quitar_tildes(pais["nombre"].lower().strip()):
                return pais
        return None

    except KeyError:
        print("\nAVISO: Algunos países no tienen el campo 'nombre'")
        return None
    except (TypeError, AttributeError) as e:
        print(f"\nAVISO: {e}")
        return None
```

La función ***quitar_tildes()*** del módulo ***utilidades.py*** normaliza los caracteres con diacríticos, permitiendo que una búsqueda de ***"mexico"*** encuentre ***"México"***:

```python
def quitar_tildes(texto):
    return ''.join(
        c for c in unicodedata.normalize('NFD', texto)
        if unicodedata.category(c) != 'Mn'
    )
```

Aplicación en el Proyecto

La función ***opcion_editar_pais()*** coordina todo el proceso de edición, desde la búsqueda hasta la persistencia.

Ejemplo de selección de campos a editar (opciones_menu.py):

```
print('\n¿Qué campos deseas editar?\n')
print('  1) Nombre\n  2) Población\n  3) Superficie\n  4) Continente\n  5) Todo\n  6) Cancelar')

try:
    opcion_editar = int(input('\nOpción: ').strip())
except ValueError:
    print('\nDebe ingresar un número válido.')
    continue

# Procesar la opción elegida
match opcion_editar:
    case 1:
        nuevo_nombre = solicitar_nuevo_nombre(paises, pais_encontrado)
        if editar_campo_pais(paises, indice_pais, 'nombre', nuevo_nombre, ruta_archivo):
            print('\nNombre actualizado correctamente.')
    case 2:
        nueva_poblacion = solicitar_valor_numerico('\nNueva población: ')
        if preguntar_si_no('¿Confirma el cambio? (s/n):'):
            if editar_campo_pais(paises, indice_pais, 'poblacion', nueva_poblacion, ruta_archivo):
                print('\nPoblación actualizada correctamente.')
    case 3:
        nueva_superficie = solicitar_valor_numerico('\nNueva superficie (km²): ')
        if preguntar_si_no('¿Confirma el cambio? (s/n):'):
            if editar_campo_pais(paises, indice_pais, 'superficie', nueva_superficie, ruta_archivo):
                print('\nSuperficie actualizada correctamente.')
    case 4:
        nuevo_continente = solicitar_nuevo_continente()
        if preguntar_si_no('¿Confirma el cambio? (s/n):'):
            if editar_campo_pais(paises, indice_pais, 'continente', nuevo_continente, ruta_archivo):
                print('\nContinente actualizado correctamente.')
```

El uso de ***match-case*** para el menú de edición hace que el código sea mucho más legible que una cadena de ***if-elif***. Cada caso maneja una opción diferente, solicitando los datos necesarios y confirmando antes de aplicar cambios.

Ejemplo de edición completa con resumen (de opciones_menu.py):

```
case 5:
    print('\n═══ Ingrese los Nuevos Datos ═══')
    nuevo_nombre = solicitar_nuevo_nombre(paises, pais_encontrado)
    nueva_poblacion = solicitar_valor_numerico('Nueva población: ')
    nueva_superficie = solicitar_valor_numerico('Nueva superficie (km²): ')
    nuevo_continente = solicitar_nuevo_continente()

    # Mostrar resumen
    print('\nResumen de cambios:')
    print('-'*70)
    print(f'  Nombre      : {pais_encontrado["nombre"]} ⟶ {nuevo_nombre}')
    print(f'  Población   : {pais_encontrado["poblacion"]:,} ⟶ {nueva_poblacion:,}')
    print(f'  Superficie  : {pais_encontrado["superficie"]:,} ⟶ {nueva_superficie:,} km²')
    print(f'  Continente  : {pais_encontrado["continente"]} ⟶ {nuevo_continente}')

    if preguntar_si_no('\n¿Confirma los cambios? (s/n): '):
        paises[indice_pais].update({
            'nombre': nuevo_nombre,
            'poblacion': nueva_poblacion,
            'superficie': nueva_superficie,
            'continente': nuevo_continente
        })
        if guardar_paises(ruta_archivo, paises):
            print('\n¡País actualizado correctamente!')
        else:
            print('\nNo se pudieron guardar los cambios.')
    else:
        print('\nCambios cancelados.')
```

El resumen visual con el formato "***valor_anterior --> valor_nuevo***" permite al usuario ver claramente qué va a cambiar antes de confirmar. El método ***.update()*** de los diccionarios actualiza múltiples campos simultáneamente.

Persistencia de Cambios

A diferencia de agregar un nuevo registro (que usa modo append), editar requiere reescribir todo el archivo CSV:

Ejemplo de reescritura completa (edicion.py)

```
def guardar_paises(ruta_archivo, paises):
    try:
        if not isinstance(paises, list):
            raise TypeError("El parámetro 'paises' debe ser una lista")

        directorio = os.path.dirname(ruta_archivo)
        if directorio and not os.path.exists(directorio):
            os.makedirs(directorio)

        with open(ruta_archivo, mode='w', newline='', encoding='utf-8') as archivo:
            campos = ['nombre', 'poblacion', 'superficie', 'continente']
            escribir = csv.DictWriter(archivo, fieldnames=campos)
            escribir.writeheader()
            escribir.writerows(paises)
        return True

    except (PermissionError, OSError) as e:
        print(f"\nAVISO: {e}")
        return False

    except Exception as e:
        print(f"\nAVISO: Error al guardar cambios en CSV - {e}")
        return False
```

El modo ***'w'*** (write) sobrescribe completamente el archivo. ***writeheader()*** escribe la fila de encabezados y ***writerows()*** escribe todos los registros de la lista ***paises*** de una sola vez.

Reversión Automática en Caso de Error

La función ***editar_campo_pais()*** implementa un patrón de reversión para garantizar la integridad de los datos:

```python
def editar_campo_pais(paises, indice, campo, nuevo_valor, ruta_archivo):
    try:
        if not (0 ≤ indice < len(paises)):
            raise ValueError(f"Índice {indice} fuera de rango")

        if campo not in ['nombre', 'poblacion', 'superficie', 'continente']:
            raise ValueError(f"Campo '{campo}' no válido")

        if campo in ['poblacion', 'superficie']:
            if not isinstance(nuevo_valor, int) or nuevo_valor ≤ 0:
                raise ValueError(f"El campo '{campo}' debe ser un número entero positivo")
        elif campo in ['nombre', 'continente']:
            if not isinstance(nuevo_valor, str) or not nuevo_valor.strip():
                raise ValueError(f"El campo '{campo}' debe ser un texto válido")

        # Guardar valor anterior
        valor_anterior = paises[indice][campo]

        paises[indice][campo] = nuevo_valor

        if guardar_paises(ruta_archivo, paises):
            print(f"\nCampo '{campo}' actualizado correctamente\n")
            print(f"  Anterior: {valor_anterior}")
            print(f"  Nuevo   : {nuevo_valor}")
            return True
        else:
            # Si falla al guardar, revertir el cambio en memoria
            paises[indice][campo] = valor_anterior
            print("\nNo se pudieron guardar los cambios en el archivo")
            return False

    except (ValueError, TypeError) as e:
        print(f"\nAVISO: {e}")
        return False

    except Exception as e:
        print(f"\nAVISO: Error inesperado - {e}")
        return False
```

Si la escritura al archivo falla (por ejemplo, por falta de permisos), el cambio en la lista en memoria se revierte restaurando el ***valor_anterior***, manteniendo la consistencia.

6.3. Operaciones de Eliminación

Definición Teórica

La operación de eliminación permite remover permanentemente un registro del conjunto de datos. Es la operación más crítica del CRUD porque es irreversible, por lo que requiere confirmaciones explícitas y manejo cuidadoso de errores.

Sistema de Doble Confirmación

Dada la naturaleza irreversible de la eliminación, el proyecto implementa un sistema de doble confirmación:

1. **Primera confirmación**: Después de mostrar los datos del país encontrado.
2. **Segunda confirmación**: Con una advertencia explícita sobre la permanencia de la acción.

Ejemplo de doble confirmación (opciones_menu.py):

```
print('\n═══ Eliminar Pais ═══\n')
pais_encontrado, indice_pais = buscar_y_seleccionar_pais(paises, 'eliminar')

if pais_encontrado is None:
    if not preguntar_si_no('\n¿Deseas intentar con otro nombre? (s/n): '):
        break
    continue

print(f'\nAVISO: Estás a punto de eliminar a "{pais_encontrado["nombre"]}" de forma permanente.')

if preguntar_si_no('¿Estás realmente seguro de que deseas continuar? (s/n): '):
    pais_a_eliminar = paises.pop(indice_pais)
    if guardar_paises(ruta_archivo, paises):
        print(f'\nEl país "{pais_a_eliminar["nombre"]}" ha sido eliminado exitosamente.')
        print(f'Total de países restantes: {len(paises)}')
    else:
        paises.insert(indice_pais, pais_a_eliminar)
        print('\nAVISO: No se pudieron guardar los cambios. La eliminación ha sido revertida.')

else:
    print('\nOperación de eliminación cancelada.')
```

Reversión Automática

Similar al patrón usado en edición, la eliminación también implementa reversión:

- Antes de guardar, se utiliza ***paises.pop(indice_pais)*** que elimina el elemento de la lista y lo devuelve.
- Si ***guardar_paises()*** falla, se restaura el país eliminado con ***paises.insert(indice_pais, pais_a_eliminar)***, que lo coloca exactamente en su posición original.
- Esto garantiza que una falla en el guardado no resulte en inconsistencia entre la memoria y el archivo.

Búsqueda y Selección de País

La función auxiliar ***buscar_y_seleccionar_pais()*** encapsula todo el proceso de búsqueda, validación y confirmación, reutilizándose tanto en edición como en eliminación:

Ejemplo de función auxiliar (auxiliares.py):

```python
def buscar_y_seleccionar_pais(paises, accion='editar'):
    nombre_buscar = input(f'Ingresa el nombre del país a {accion}: ').strip()

    if not nombre_buscar:
        print('\nEl nombre no puede estar vacío.')
        return None, None

    pais_encontrado = buscar_exacto(paises, nombre_buscar)

    if pais_encontrado is None:
        print(f'\nNo se encontró ningún país con el nombre "{nombre_buscar}"')
        print('El nombre debe ser exacto (puedes omitir tildes y mayúsculas)')
        return None, None

    print(f'\nPaís encontrado: {pais_encontrado["nombre"]}')
    print('-' * 70)
    print(f'  Población  : {pais_encontrado["poblacion"]:,} habitantes')
    print(f'  Superficie : {pais_encontrado["superficie"]:,} km²')
    print(f'  Continente : {pais_encontrado["continente"]}')
    print('-' * 70)

    if not preguntar_si_no(f'\n¿Es este el país que deseas {accion}? (s/n): '):
        print('\nBúsqueda cancelada.')
        return None, None

    indice_pais = next((i for i, pais in enumerate(paises) if pais['nombre'] == pais_encontrado['nombre']), None)

    if indice_pais is None:
        print('\AVISO: no se encontró el país en la base de datos interna.')
        return None, None

    return pais_encontrado, indice_pais
```

Esta función:

- Acepta un parámetro ***accion*** que personaliza los mensajes ("editar" o "eliminar").
- Muestra toda la información del país encontrado antes de pedir confirmación.
- Retorna una tupla con el país encontrado y su índice en la lista, o ***(None, None)*** si el usuario cancela.
- Utiliza una expresión generadora con ***next()*** para encontrar eficientemente el índice.

6.4. Funciones Auxiliares de Validación

El módulo ***auxiliares.py*** contiene funciones reutilizables que encapsulan patrones comunes de validación, siguiendo el principio ***DRY***.

Validación de Valores Numéricos

```python
def solicitar_valor_numerico(mensaje_prompt):
    while True:
        try:
            valor = int(input(mensaje_prompt).strip())
            if valor ≤ 0:
                print('El valor debe ser un número positivo.')
                continue
            return valor
        except ValueError:
            print('Debes ingresar un número entero válido.')
```

Esta función permanece en un bucle hasta que el usuario ingrese un valor válido, manejando tanto errores de conversión como valores fuera de rango.

Validación de Nombres Únicos

```python
def solicitar_nuevo_nombre(paises, pais_actual):
    while True:
        nuevo_nombre = input('\nNuevo nombre: ').strip()
        if not nuevo_nombre:
            print('El nombre no puede estar vacío.')
            continue
        if any(c.isdigit() for c in nuevo_nombre):
            print('El nombre no puede contener números.')
            continue

        # Verificar si ya existe otro país con ese nombre
        pais_duplicado = buscar_exacto(paises, nuevo_nombre)
        if pais_duplicado and pais_duplicado['nombre'] ≠ pais_actual['nombre']:
            print(f'Ya existe un país con el nombre "{pais_duplicado["nombre"]}".')
            continue

        return nuevo_nombre.title()
```

Esta función se fija si hay otro país con el mismo nombre, pero deja pasar el actual para que no salte error si solo se cambian otros datos.

Selección de Continente

```python
def solicitar_nuevo_continente():
    continentes = ['África', 'América del Norte', 'América del Sur', 'Asia', 'Europa', 'Oceanía']

    print('\nContinentes disponibles:')
    for i, cont in enumerate(continentes, 1):
        print(f'  {i}) {cont}')

    while True:
        try:
            opcion = int(input('\nSeleccione el número: ').strip())
            if 1 ≤ opcion ≤ len(continentes):
                return continentes[opcion - 1]
            else:
                print(f'Opción inválida. Debe elegir entre 1 y {len(continentes)}.')
        except ValueError:
            print('Debes ingresar un número válido.')
```

En lugar de permitir entrada de texto libre, esta función presenta un menú numerado, reduciendo errores de tipeo y estandarizando los valores de continente.

6.5. Integración con el Flujo del Programa

Las operaciones ***CRUD*** se integran perfectamente en el bucle principal del programa a través del ***match-case*** en ***main.py***:

```python
# Ejecutar opción seleccionada
match opcion_int:
    case 1:
        opcion_buscar_pais(paises)

    case 2:
        opcion_filtrar_continente(paises)

    case 3:
        opcion_filtrar_poblacion(paises)

    case 4:
        opcion_filtrar_superficie(paises)

    case 5:
        opcion_ordenar_paises(paises)

    case 6:
        opcion_mostrar_estadisticas(paises)

    case 7:
        opcion_agregar_pais(paises, ruta)

    case 8:
        opcion_editar_pais(paises, ruta)

    case 9:
        opcion_eliminar_pais(paises, ruta)

    case 10:
        print('\n¡Gracias por usar el programa!')
        print('Saliendo ... ')
        break

    case _:
        print('\nLa opción ingresada no es válida')
```

Las funciones de modificación (casos 7, 8 y 9) reciben tanto la lista ***paises*** como la ***ruta*** del archivo CSV, permitiéndoles modificar ambas representaciones de los datos.

Conclusión

La ***"Gestión de Datos de Países"*** muestra claramente cómo los conceptos básicos de programación en Python pueden combinarse para crear una aplicación funcional, sólida y bien organizada. En este proyecto, se usan listas para guardar todos los países y diccionarios para representar cada uno de ellos. Esto permite tener una base de datos en memoria que es simple de entender y muy eficiente.

El código está dividido en funciones modulares, cada una con una tarea bien definida. Esto no solo mejora la lectura y el mantenimiento, sino que también permite reutilizar código y trabajar de forma más ordenada.

El control de flujo, manejado con estructuras como el ***match-case***, hace que la navegación por el menú sea clara y lógica. Por otro lado, las funciones de ordenamiento y estadísticas aprovechan las herramientas integradas de Python para procesar y resumir la información de manera rápida y precisa.

Además, el hecho de que el programa pueda leer y procesar datos desde un archivo CSV lo vuelve flexible y totalmente independiente de la fuente de datos.

La implementación de un sistema CRUD completo (crear, leer, actualizar y eliminar) eleva el proyecto de una simple consulta a un verdadero sistema de gestión de información. El conjunto de validaciones, mensajes de confirmación y mecanismos de reversión automática muestra una clara preocupación por la experiencia del usuario y por mantener la integridad de los datos.

El patrón usado: guardar el valor anterior, modificar, guardar y volver atrás si algo falla, es una práctica muy profesional que evita errores y mantiene la coherencia entre la memoria y el archivo.

Las funciones auxiliares incluidas en el módulo ***auxiliares.py*** reflejan el principio ***DRY*** ("Don't Repeat Yourself"), al reunir en un solo lugar todas las validaciones comunes. Esto evita repetir código y facilita el mantenimiento.

Por último, la normalización del texto, que hace que las búsquedas no dependan de tildes o mayúsculas, muestra atención a los pequeños detalles que mejoran la usabilidad, sobre todo en un entorno hispanohablante.

En conjunto, este proyecto no solo cumple con lo que se propuso, sino que también demuestra buenas prácticas de programación, desde la organización del código hasta la creación de una interfaz de consola simple pero interactiva.

Referencias

Apuntes y ejemplos vistos en clase

W3Schools. *Python Tutorial* :

- https://www.w3schools.com/python/

Real Python. *Working With Files in Python*:

- https://realpython.com/

Python.org. (2024). *Documentación oficial de Python 3*:

- **https://docs.python.org/es/3/**

Link al vídeo explicativo

- video_presentacion_tpi_programacion.mp4