TRABAJO PRÁCTICO INTEGRADOR

PROGRAMACIÓN 1

3 DE NOVIEMBRE DE 2025

ALUMNOS: FABRIZIO SIMÓN Y FRANCO RIOS
PROFESOR: CINTHIA RIGONI / RAMIRO HUALPA

Índice página

1. 📜 Introducción 3
2. 🧠 Desarrollo Teórico (Conceptos Aplicados) 4
   - 2.1. Estructuras de Datos: Listas y Diccionarios 4
   - 2.2. Modularización: Funciones y Módulos 4
   - 2.3. Manejo de Archivos: Lectura de CSV 5
   - 2.4. Técnicas de Procesamiento de Datos 5
   - 2.5. Validación de Entradas de Usuario 6
3. 🚀 Desarrollo del Sistema (Nuestra Solución) 6
   - 3.1. Arquitectura del Código 6
   - 3.2. Descripción de los Módulos 7
     - main.py (Controlador)
     - funciones.py (Lógica de Negocio)
     - validaciones.py (Utilidades)
   - 3.3. Decisiones Clave de Diseño 7
4. 💡 Conclusiones personales 8
   - 4.1. Conclusión (Franco Rios) 8
   - 4.2. Conclusión (Fabrizio Simon) 8
5. 🔗 Referencias 9

1. Introducción

El presente informe documenta en detalle el proceso de diseño, desarrollo, prueba e implementación del Trabajo Práctico Integrador (TPI) correspondiente a la materia Programación 1. Este proyecto representa un hito fundamental en la carrera de la Tecnicatura Universitaria en Programación, ya que constituye la primera oportunidad para sintetizar y aplicar la totalidad de los conceptos algorítmicos y estructurales aprendidos durante la cursada. El objetivo no era solo escribir código funcional, sino enfrentarnos a un problema práctico de gestión de datos, simulando un requerimiento del mundo real.

La consigna central fue la creación de un sistema de software robusto en Python, operado por consola, capaz de administrar un conjunto de información sobre los países del mundo. Los requerimientos funcionales incluían la ingesta de datos desde un archivo de formato CSV (Valores Separados por Coma), su almacenamiento en memoria utilizando las estructuras de datos más eficientes para este caso (listas y diccionarios), y la provisión de una interfaz de usuario interactiva (CLI) para realizar un conjunto completo de operaciones de consulta y procesamiento.

El resultado es una aplicación modular y escalable que permite al usuario cargar el set de datos, realizar búsquedas por coincidencia parcial de nombre (con normalización de texto), filtrar países por continente, rangos de población y superficie, ordenar los resultados dinámicamente según diversos criterios y, finalmente, calcular y mostrar un panel de estadísticas descriptivas sobre la información. Este documento no solo describirá la solución técnica final y la arquitectura de tres módulos implementada, sino también los fundamentos teóricos, las decisiones de diseño tomadas y las lecciones aprendidas durante el ciclo de vida del desarrollo.

2. Desarrollo Teórico (Conceptos Aplicados)

En esta sección, se describen los fundamentos teóricos de Python que fueron necesarios para la construcción del sistema, explicando cómo cada concepto contribuyó a la solución final.

2.1. Estructuras de Datos: Listas y Diccionarios

El núcleo del programa reside en la correcta elección de las estructuras de datos para almacenar la información.

- Listas (List): Son colecciones ordenadas, mutables y heterogéneas. En nuestro proyecto, la lista principal datos_paises (definida en main.py) se utilizó como el contenedor principal. Esta estructura fue ideal porque nos permite almacenar una colección de N países, donde cada país es un elemento de la lista, facilitando la iteración sobre el conjunto de datos.
- Diccionarios (Dict): Son colecciones desordenadas (en versiones antiguas de Python), mutables y mapeadas por clave-valor. Elegimos representar cada país individual como un diccionario (ej: {'nombre': 'Argentina', 'poblacion': 45376763, ...}). Esta decisión fue crucial para la legibilidad del código. Permite un acceso semántico y claro a los datos (ej: pais['poblacion']) en lugar de usar índices numéricos (ej: pais[1]), lo cual haría el código frágil y difícil de mantener si el orden de las columnas del CSV cambiara.

La combinación de una lista de diccionarios nos brindó la flexibilidad de las listas (para almacenar un número variable de países) y la claridad de los diccionarios (para representar cada país de forma estructurada).

2.2. Modularización: Funciones y Módulos

Para evitar un único archivo extenso y difícil de manejar (conocido como "código espagueti"), aplicamos el principio de modularización.

- Funciones: Permiten encapsular bloques de lógica reutilizables, evitar la repetición de código (principio DRY - Don't Repeat Yourself) y mejorar la legibilidad general del programa. Cada función tiene una única responsabilidad (ej: filtrar_por_continente solo filtra).
- Módulos: Son archivos .py que agrupan funciones relacionadas. En nuestro TPI, separamos el código en tres módulos (main.py, funciones.py, validaciones.py). Esta "separación de responsabilidades" es clave: main.py maneja la interacción con el usuario, funciones.py maneja la lógica de datos y validaciones.py maneja la limpieza de entradas. Esto resulta en alta cohesión (cada módulo hace una sola cosa bien) y bajo acoplamiento (los módulos no dependen excesivamente entre sí).

2.3. Manejo de Archivos: Lectura de CSV

Para cargar los datos, se utilizó la lectura de archivos de texto plano, específicamente el formato CSV (Comma Separated Values).

- Bloque with open(...): Se utilizó la sintaxis with open(...) as archivo: para la apertura del archivo. Esta es la práctica recomendada en Python, ya que gestiona automáticamente el cierre del archivo (invocando archivo.close()) incluso si ocurren errores durante la lectura, previniendo fugas de recursos.
- Manejo de Excepciones: Se implementó un manejo de errores robusto. Un bloque try...except FileNotFoundError externo envuelve la operación de apertura para capturar el error crítico de que el archivo paises.csv no exista. Además, un try...except ValueError anidado se utiliza dentro del bucle de lectura para omitir líneas individuales con formato incorrecto (ej: un valor no numérico en el campo de población), permitiendo que el programa continúe cargando el resto de los datos válidos.

2.4. Técnicas de Procesamiento de Datos

Una vez con los datos en memoria, aplicamos técnicas modernas de Python para procesarlos.

- Filtrado (List Comprehensions): Para las búsquedas y filtros (por rango y continente), se utilizaron "list comprehensions" (comprensión de listas). Esta es una característica de Python que permite crear nuevas listas de forma concisa, legible y eficiente a partir de una existente (ej: [pais for pais in paises if pais['poblacion'] > 1000000]).
- Ordenamiento (Función sorted()): Para ordenar, se utilizó la función nativa sorted(), la cual es altamente optimizada (implementa el algoritmo Timsort). Se le proveyó una función lambda en el argumento key (ej: key=lambda p: p['poblacion']) para indicarle al algoritmo por cuál de los campos del diccionario debía ordenar. El argumento reverse=True/False permitió controlar la dirección ascendente o descendente.

2.5. Validación de Entradas de Usuario

Un pilar fundamental de la aplicación es la robustez frente al usuario. Nunca se debe confiar en los datos de entrada.

Se implementaron funciones de validación (ej: validar_entero, validar_string) que utilizan un bucle while True para "atrapar" al usuario. Este bucle asegura que el programa no continúe hasta que el usuario ingrese un dato del tipo y formato esperado. Se utilizó try...except ValueError para capturar errores de conversión (ej: si el usuario escribe "cien" en lugar de "100"), mejorando la experiencia de usuario (UX) al evitar que el programa colapse por una entrada inválida.

3. Desarrollo del Sistema (Nuestra Solución)

Basados en los conceptos anteriores, se diseñó la siguiente arquitectura.

3.1. Arquitectura del Código

El sistema se divide en tres módulos, siguiendo el principio de separación de responsabilidades:

1. main.py (Capa de Presentación / Controlador): Es el punto de entrada. Su única responsabilidad es mostrar el menú, capturar la opción del usuario y orquestar las llamadas a las funciones correspondientes del módulo funciones. No contiene lógica de negocio; es el "director de orquesta".
2. funciones.py (Capa de Lógica / Modelo): Contiene el "cerebro" de la aplicación. Aquí residen todas las funciones que manipulan los datos (cargar, buscar, filtrar, ordenar, calcular estadísticas) y la función que se encarga de formatear e imprimir los datos (mostrar_paises).
3. validaciones.py (Capa de Utilidad): Contiene funciones auxiliares y genéricas, reutilizables en cualquier proyecto, que validan las entradas del usuario (números, rangos, strings) para asegurar que sean correctas antes de ser procesadas.

3.2. Descripción de los Módulos

- **main.py:** Gestiona el bucle principal del menú (while True). Mantiene el estado de la aplicación en la variable datos_paises. Implementa una guarda crucial: verifica si datos_paises está vacía antes de permitir al usuario ejecutar opciones que requieren datos (de la 2 a la 8), forzándolo a cargar datos primero.
- **funciones.py**: Es el módulo más denso. cargar_datos() procesa el CSV. Funciones como buscar_por_nombre() y filtrar_por_continente() demuestran el uso de validar_texto (del módulo de validaciones) para normalizar tanto los datos del diccionario como la entrada del usuario, permitiendo búsquedas insensibles a mayúsculas o acentos.
- **validaciones.py:** Provee **validar_entero(), validar_opcion_menu(), validar_rango_numerico()** y **validar_string().** Estas funciones encapsulan la lógica de pedir datos al usuario (input()) y validarlos, devolviendo un valor "limpio" y seguro a la capa de lógica.

3.3. Decisiones Clave de Diseño

Durante el desarrollo, se tomaron varias decisiones de diseño importantes:

1. **Normalización de Texto**: Se decidió implementar validar_texto() para convertir todas las búsquedas y los datos de texto a minúsculas y sin acentos. Esto mejora drásticamente la experiencia del usuario (UX), ya que si busca "America", "américa" o "AMERICA", el filtro filtrar_por_continente funcionará igualmente.
2. **Inmutabilidad de Datos**: Las funciones de filtrado y ordenamiento (**filtrar_por_rango**, **ordenar_paises**, etc.) no modifican la lista original **datos_paises**. En su lugar, crean y devuelven una nueva lista (resultados, **paises_ordenados**) con los datos procesados. Esto es una buena práctica que previene efectos secundarios inesperados y permite al usuario realizar nuevos filtros sobre la lista original intacta.
3. **Manejo de Errores en CSV:** Se decidió conscientemente que un error en una sola línea del CSV no debía detener la carga completa del archivo. El try...except interno en **cargar_datos** permite que el programa reporte el error en esa línea y continúe con las siguientes, asegurando que la mayor cantidad de datos válidos esté disponible para el usuario.

4. Conclusiones Personales

4.1. Conclusión (Franco Rios)

Este trabajo práctico me permitió integrar por primera vez todos los conceptos vistos en la materia. Personalmente, lo que más me costó fue diseñar correctamente la separación en módulos, decidiendo qué función pertenecía a qué archivo. Al principio, tendía a poner demasiada lógica en main.py. Sin embargo, al refactorizar el código para mover la lógica de negocio a funciones.py y las validaciones a validaciones.py, entendí el verdadero valor de la modularización, el bajo acoplamiento y la alta cohesión.

La elección de una lista de diccionarios fue un acierto clave; facilitó enormemente la implementación de las funciones de filtrado y ordenamiento, especialmente al usar funciones lambda con sorted(). Estoy satisfecho con la robustez general del programa, en particular con el módulo de validaciones, que evita que la aplicación falle por entradas inesperadas del usuario.

4.2. Conclusión (Fabrizio Simón)

La realización de este TPI fue un desafío que me ayudó a unificar mi comprensión sobre el procesamiento y la manipulación de datos. Lo que más destaco del proyecto es la implementación de las funciones de filtrado y búsqueda. Desarrollar la lógica para la búsqueda insensible a mayúsculas y acentos (usando la función validar_texto) me hizo pensar mucho en la experiencia de usuario (UX); comprendí que no basta con que el programa funcione, sino que debe ser intuitivo y "perdonar" pequeños errores del usuario, como omitir un acento.

La función de estadísticas (mostrar_estadisticas) también fue muy satisfactoria, ya que pude aplicar funciones de agregación como max, min y sum directamente sobre la estructura de datos que diseñamos, demostrando su flexibilidad. Este proyecto me dio confianza en mi capacidad para tomar un conjunto de datos "crudo" y convertirlo en información útil y consultable.

Referencias

Documentación Oficial (Fuente Principal):

- Python Software Foundation. (2025). Tutorial de Python: Estructuras de Datos (Listas, Diccionarios). Recuperado de: https://docs.python.org/3/tutorial/datastructures.html
- Python Software Foundation. (2025). Tutorial de Python: Módulos. Recuperado de: https://docs.python.org/3/tutorial/modules.html
- Python Software Foundation. (2025). Tutorial de Python: Entrada y Salida (Archivos). Recuperado de: https://docs.python.org/3/tutorial/inputoutput.html

Libros y Material de Cátedra:

- Apuntes de Cátedra de Programación 1. (2025). Tecnicatura Universitaria en Programación. UTN-FRM.

Recursos Web Adicionales:

- Real Python. (s.f.). Reading and Writing CSV Files in Python. Recuperado de: https://realpython.com/python-csv/
- W3Schools. (s.f.). Python Dictionaries. Recuperado de: https://www.w3schools.com/python/python_dictionaries.asp