Parcial 2 - programación 1

Gestión de Datos de autos en Python: filtros, ordenamientos y estadísticas

Alumnos - Grupo n° 6

Avalo, Pablo – Suárez, Ismael.

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación 1 - 2025

Docente

Ramiro, Hualpa

5 de noviembre de 2025

Tabla de contenido

Trabajo Integrador Programación 1 -2025

Introducción

En el marco de la asignatura Programación 1, desarrollamos una aplicación de consola en Python para la consulta y administración de datos sobre autos. Nuestra solución integra un menú principal y dos modos de operación —local (archivo CSV) y remoto (API REST)— que nos permitieron ejercitar la persistencia en archivos y el consumo de servicios HTTP. Implementamos búsquedas, filtros, ordenamientos, estadísticas y un CRUD para altas, bajas y modificaciones (ABM). Priorizamos una arquitectura modular con funciones en español y documentamos el código mediante docstrings estilo Google, favoreciendo la legibilidad y la mantenibilidad. Si bien el desarrollo responde a los objetivos iniciales del cursado, dejamos una base sólida para incorporar validaciones, pruebas automatizadas y mejoras evolutivas en futuras iteraciones.

PARCIAL PROGRAMACIÓN N°02

Objetivos

Desarrollar una aplicación en Python que permita gestionar información sobre un tema a elección de los alumnos (en nuestro caso optamos por autos), aplicando listas, diccionarios, funciones, estructuras condicionales y repetitivas, ordenamientos y estadísticas. El sistema debe ser capaz de leer datos desde un archivo CSV, realizar consultas y generar indicadores clave a partir del dataset. El objetivo principal es afianzar el uso de estructuras de datos, modularización con funciones y técnicas de filtrado/ordenamiento, aplicando los conceptos aprendidos en Programación 1.

Consignas generales

• Lenguaje: Python 3.x+

• Estructuras: listas, diccionarios, funciones.

• Archivos: lectura desde CSV.

• Código claro, comentado y modularizado (una función = una responsabilidad).

• Validaciones de entradas y manejo básico de errores.

• Trabajo en equipos de 2 personas.

Dominio (dataset de autos)

Cada país estará representado con los siguientes datos:

• Marca (string)

• año del vehiculo (int) • Modelo (string)

• Combustible (string)

• Transmisión (string)

Ejemplo de registro CSV:

Marca, Modelo, Año, Combustible, Transmisión

1. Nissan March | Año: 2014 | Combustible: nafta | Transmisión: manual

2. Audi Sedan | Año: 2016 | Combustible: Híbrido | Transmisión: Manual

3. Chevrolet Sedan | Año: 2019 | Combustible: Híbrido | Transmisión: Automática

PARCIAL PROGRAMACIÓN N°02

Requerimientos técnicos

1) Diseño (previo al código)

Explicar en un informe teórico los conceptos aplicados:

o Listas

o Diccionarios

o Funciones

o Condicionales

o Ordenamientos

o Estadísticas básicas

o Archivos CSV • Definir el flujo de operaciones principales en un diagrama

o esquema.

2) Funcionalidades mínimas del sistema

El programa debe ofrecer un menú de opciones en consola que permita:

• Buscar un auto por su marca o modelo **(coincidencia parcial o exacta).**

• Filtrar vehículos por: o Transmisión o Año o Tipo de combustible.

• Ordenar vehículos por:

o Marca

o Modelo

o año (de 1900 a 2100)

o Tipo de combustible (nafta, diésel, híbrido, eléctrico)

o Transmisión

• **Mostrar estadísticas:**

o Cantidad de modelos disponibles por marca

o Cantidad de autos por tipo de combustible

o Cantidad de autos por transmisión

3) Validaciones

• Controlar errores de formato en el CSV.

• Evitar fallos al ingresar filtros inválidos o búsquedas sin resultados.

• Mensajes claros de éxito/error.

Entregables (obligatorios)

1. Carpeta digital

• Marco teórico con fuentes bibliográficas.

• Código Python funcional, modular y comentado.

• Capturas de pantalla de ejecución de ejemplos.

• Conclusiones grupales sobre los aprendizajes.

2. Repositorio en GitHub

Debe incluir:

• Proyecto completo en Python.

• README.md con:

o Descripción del programa.

o Instrucciones de uso.

o Ejemplos de entradas y salidas.

o Participación de los integrantes.

• Archivo CSV con el dataset base.

3. Video tutorial (10–15 minutos)

• Explicación del problema planteado.

• Presentación de la estructura de datos utilizada.

• Demostración del programa funcionando.

• Reflexión final sobre el desarrollo del proyecto.

Criterios de evaluación

• Correcta funcionalidad (búsquedas, filtros, ordenamientos, estadísticas).

• Uso correcto de estructuras de datos (listas y diccionarios).

• Uso de API y recursividad.

• Calidad del código (modularización, legibilidad, comentarios).

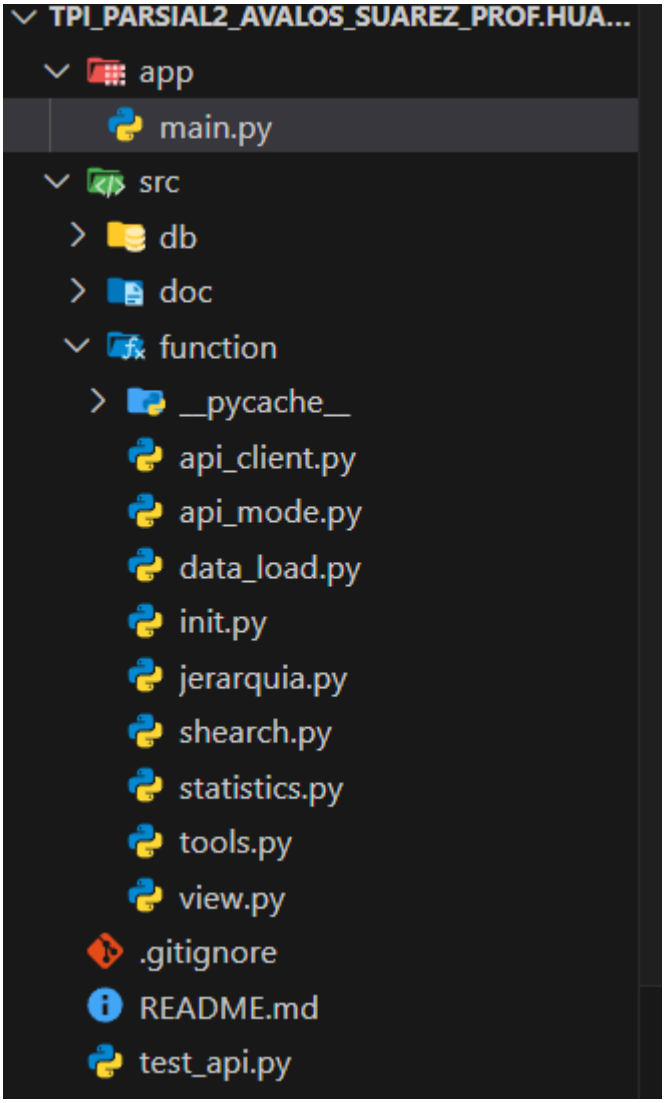
• Documentación (README claro, informe teórico coherente).

• Presentación en video (tiempo adecuado, explicación técnica, participación equitativa).

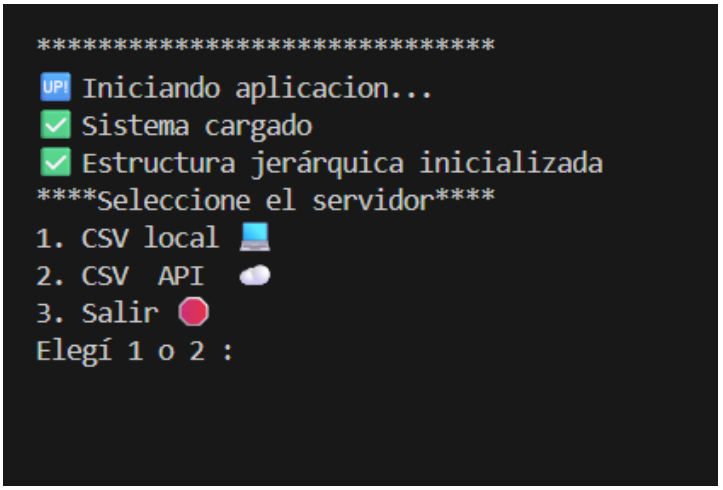
• Entrega completa en GitHub con código, informe y CSV

1-Funcionalidades mínimas del sistema

Estructura:



Selección de servidor



1-Menú

```
**********GESTIÓN DE AUTOS**********
1.  Buscar auto por marca o modelo
2.  Filtrar por tipo de combustible
3.  Filtrar por rango de año
4.  Filtrar por transmisión
5.  Ordenar autos
6.  Mostrar estadisticas
7.  Agregar un auto
8.  Editar un auto
9.  Borrar auto
10. Cambiar modo de servidor
11. Salir
Ingrese una opcion 1-11:
```

2-Validaciones

```python
def buscar_auto_recursivo(autos, busqueda, indice=0, encontrados=None):
    """Busca autos de forma recursiva por marca o modelo.

    Implementación recursiva para cumplir con los requisitos del proyecto.

    Args:
        autos (list[dict]): Lista de autos donde buscar.
        busqueda (str): Texto a buscar en Marca o Modelo.
        indice (int): Índice actual en la lista (para recursión).
        encontrados (list[dict]): Lista acumulativa de resultados.

    Returns:
        list[dict]: Lista de autos que coinciden con la búsqueda.
    """
    if encontrados is None:
        encontrados = []

    # Caso base: se recorrió toda la lista
    if indice >= len(autos):
        return encontrados

    # Procesar el elemento actual
    auto_actual = autos[indice]
    busqueda_norm = normalizar(busqueda)
    marca_norm = normalizar(auto_actual.get("Marca", ""))
    modelo_norm = normalizar(auto_actual.get("Modelo", ""))

    if busqueda_norm in marca_norm or busqueda_norm in modelo_norm:
        encontrados.append(auto_actual)

    # Llamada recursiva para el siguiente elemento
    return buscar_auto_recursivo(autos, busqueda, indice + 1, encontrados)
```

Funcionamiento

```
*********Estadísticas generales*********
▫ 🚗 Auto más antiguo: Honda Sedan (2010)
▫ 🚗 Auto más nuevo: Ford Crossover (2024)
▫ 🗓️ Año promedio: 2016

*********Cantidad de autos por marca*********
      - Audi: 9
      - BMW: 8
      - Chevrolet: 14
      - Ford: 10
      - Honda: 11
      - Hyundai: 11
      - Kia: 12
      - Mazda: 13
      - Mercedes-Benz: 5
      - Nissan: 11
      - Peugeot: 7
      - Renault: 10
      - Subaru: 10
      - Toyota: 7
      - Volkswagen: 13

*********Cantidad de autos por tipo de combustible*********
      - Diesel: 29
      - Eléctrico: 37
      - Híbrido: 41
      - Nafta: 43
      - nafta: 1

*********Cantidad de autos por transmisión*********
      - Automática: 73
      - Manual: 77
      - manual: 1
**************************************************
```

- Se realiza la búsqueda de la estadística de los vehículos cargados en el csv

```python
def contar_autos_por_marca_recursivo(autos, indice=0, conteo=None):
    """Cuenta autos por marca de forma recursiva.

    Implementación recursiva para cumplir con los requisitos del proyecto.

    Args:
        autos (list[dict]): Lista de autos.
        indice (int): Índice actual en la lista (para recursión).
        conteo (dict): Diccionario acumulativo de conteos.

    Returns:
        dict: Diccionario con marcas como claves y cantidad de autos como valores.
    """
    if conteo is None:
        conteo = {}

    # Caso base: se recorrió toda la lista
    if indice >= len(autos):
        return conteo

    # Procesar el elemento actual
    auto_actual = autos[indice]
    marca = auto_actual.get("Marca", "Desconocida")
    conteo[marca] = conteo.get(marca, 0) + 1

    # Llamada recursiva para el siguiente elemento
    return contar_autos_por_marca_recursivo(autos, indice + 1, conteo)
```

```python
def contar_autos_por_combustible_recursivo(autos, indice=0, conteo=None):
    """Cuenta autos por tipo de combustible de forma recursiva.

    Implementación recursiva para cumplir con los requisitos del proyecto.

    Args:
        autos (list[dict]): Lista de autos.
        indice (int): Índice actual en la lista (para recursión).
        conteo (dict): Diccionario acumulativo de conteos.

    Returns:
        dict: Diccionario con tipos de combustible como claves y cantidad de autos como valores.
    """
    if conteo is None:
        conteo = {}

    # Caso base: se recorrió toda la lista
    if indice >= len(autos):
        return conteo

    # Procesar el elemento actual
    auto_actual = autos[indice]
    combustible = auto_actual.get("TipoCombustible", "Desconocido")
    conteo[combustible] = conteo.get(combustible, 0) + 1

    # Llamada recursiva para el siguiente elemento
    return contar_autos_por_combustible_recursivo(autos, indice + 1, conteo)
```

```python
def sumar_años_recursivo(autos, indice=0):
    """Suma los años de los autos de forma recursiva.

    Implementación recursiva para cumplir con los requisitos del proyecto.

    Args:
        autos (list[dict]): Lista de autos.
        indice (int): Índice actual en la lista (para recursión).

    Returns:
        int: Suma total de años.
    """
    # Caso base: se recorrió toda la lista
    if indice >= len(autos):
        return 0

    # Procesar el elemento actual
    año_actual = autos[indice].get("Año", 0)

    # Llamada recursiva para el siguiente elemento y sumar
    return año_actual + sumar_años_recursivo(autos, indice + 1)
```

Conclusión

Como equipo, integramos los conceptos vistos en clase en una solución coherente y funcional. Leímos y persistimos datos en CSV, replicamos las mismas operaciones contra una API remota y unificamos la experiencia en una interfaz de consola simple. Este recorrido nos permitió consolidar el manejo de tipos de datos, control de flujo, funciones y colecciones, además de normalizar entradas y separar responsabilidades (vista, lógica y utilidades). Como próximos pasos, proponemos elevar la calidad con pruebas unitarias y automatizadas, validaciones más estrictas (límites y formatos), manejo de errores más detallado, registro de eventos (logging) y empaquetado para distribución. El mayor desafío fue manejar los errores en los filtros y las búsquedas, lo que nos permitió mejorar la validación de entradas y modularización del código. Consideramos que el trabajo cumplió los objetivos de aprendizaje y nos deja una base firme para seguir avanzando

Referencias

-Tecnicatura Universitaria en Programación.

Programación 1-2025. Unidad 5 Universidad Tecnológica Nacional.

-Video Presentación YouTube:

https://youtu.be/k9_z9b7sKdY