Proyecto: Analizador de Datos con Python y Flet

Este documento describe una estructura sugerida y consideraciones para desarrollar una aplicación de escritorio y web utilizando Python y Flet para el análisis básico de datasets en formato CSV.

1. Objetivos de la Aplicación

- **Carga de Archivos:** Permitir al usuario cargar archivos CSV.
- **Análisis Básico:**
  - Mostrar el número total de registros (filas) y columnas.
  - Listar los encabezados (nombres de columnas) del dataset.
- **Consultas Personalizadas:**
  - Permitir al usuario ingresar código o comandos para consultar los datos cargados.
  - Mostrar los resultados de las consultas en formato de tabla.
- **Visualización:**
  - Generar gráficos básicos a partir de los datos o resultados de consultas.
- **Plataformas:** Funcionar como aplicación de escritorio y aplicación web.

2. Librerías Propuestas y Consideraciones

Aquí revisamos las librerías que mencionaste y algunas sugerencias:

- **Interfaz de Usuario (UI):**
  - `flet`: Framework principal para construir la UI tanto para web como para escritorio.
  - `flet-desktop` (o herramientas equivalentes como `PyInstaller`, `cx_Freeze`): Para empaquetar la aplicación Flet como un ejecutable de escritorio. Flet mismo maneja gran parte de esto.
- **Manipulación y Análisis de Datos:**
  - `pandas`: Esencial para la manipulación de DataFrames. Su función `read_csv` es robusta. Es la opción más común y con gran integración.
  - `numpy`: Dependencia fundamental de Pandas y útil para operaciones numéricas si se requieren directamente.
  - `polars`: Alternativa moderna y muy rápida a Pandas. Podrías considerarla si el rendimiento con datasets muy grandes es una preocupación primordial desde el inicio. Para empezar, Pandas suele ser más que suficiente y tiene una curva de aprendizaje más suave para muchos. **Sugerencia:** Comienza con Pandas; si el rendimiento se convierte en un cuello de botella, explora Polars. Usar ambos podría ser redundante al principio.
- **Visualización de Datos:**

    - matplotlib: Base para muchas librerías de gráficos en Python. Potente y personalizable.
    - seaborn: Construida sobre Matplotlib, facilita la creación de gráficos estadísticos más atractivos y complejos con menos código. Excelente combinación con Pandas.
- **Presentación de Tablas (UI):**
    - Flet provee su propio control DataTable (flet.DataTable) que es ideal para mostrar datos tabulares directamente en la interfaz gráfica.
    - tabulate: Útil para crear tablas en formato de texto (por ejemplo, para logs en consola o representaciones textuales simples). Probablemente no sea necesaria para la visualización principal en la UI si usas flet.DataTable.
- **Motor de Consultas (Importante para "ingresar código de consultas"):**
    - **Sugerencia:** duckdb. Esta librería es una base de datos analítica en proceso que puede ejecutar consultas SQL directamente sobre DataFrames de Pandas (y Polars). Es increíblemente rápida y fácil de integrar. Permitir que los usuarios escriban SQL es más seguro y estructurado que intentar ejecutar código Python arbitrario.
- **Calidad de Código y Testing:**
    - pytest: Framework de testing robusto y popular.
    - black: Formateador de código automático para mantener un estilo consistente.
    - flake8: Linter para verificar errores de estilo y calidad de código (PEP8).
    - isort: Organiza automáticamente los imports.

Librerías Resumidas (Recomendadas para empezar):

1. flet (para UI web y escritorio)
2. pandas (para manipulación de datos)
3. numpy (dependencia de pandas, útil por sí misma)
4. matplotlib (para gráficos)
5. seaborn (para gráficos estadísticos avanzados)
6. duckdb (para ejecutar consultas SQL sobre DataFrames)
7. pytest (para testing)
8. black, flake8, isort (para calidad de código)

3. Estructura de Directorios Sugerida

Una estructura modular facilitará el mantenimiento y la escalabilidad:

```
analizador_datos_flet/
│
├── app/                  # Lógica principal de la aplicación Flet
│   ├── __init__.py
│   ├── main.py           # Punto de entrada de la aplicación Flet, define las vistas principales
```

```
│   ├── navigation.py        # (Opcional) Manejo de la navegación entre vistas/rutas
│   │
│   ├── views/               # Módulos para cada vista/página de la aplicación
│   │   ├── __init__.py
│   │   ├── file_upload_view.py # Lógica y UI para la carga de archivos
│   │   ├── data_display_view.py # Lógica y UI para mostrar datos, encabezados, conteos
│   │   └── query_view.py      # Lógica y UI para la entrada de consultas y visualización de resultados
│   │
│   ├── controls/            # (Opcional) Controles Flet personalizados y reutilizables
│   │   ├── __init__.py
│   │   ├── data_table_custom.py # Ejemplo: un DataTable con funcionalidades extendidas
│   │   └── plot_container.py    # Ejemplo: un contenedor para mostrar gráficos Matplotlib/Seaborn
│   │
│   └── assets/              # Archivos estáticos (íconos, fuentes personalizadas, etc.)
│       └── icon.png
│
├── core/                    # Lógica de negocio y procesamiento de datos (independiente de Flet)
│   ├── __init__.py
│   ├── data_loader.py       # Funciones para cargar y validar archivos CSV (usando Pandas)
│   ├── data_analyzer.py     # Funciones para análisis básico (conteo, encabezados)
│   ├── query_engine.py      # Funciones para ejecutar consultas (usando DuckDB sobre Pandas
DataFrame)
│   └── plot_generator.py    # Funciones para generar gráficos (usando Matplotlib/Seaborn)
│
├── tests/                   # Pruebas unitarias y de integración con Pytest
│   ├── __init__.py
│   ├── core/                # Pruebas para la lógica de negocio
│   │   ├── __init__.py
│   │   ├── test_data_loader.py
│   │   └── test_query_engine.py
│   └── app/                 # Pruebas para componentes de la UI (pueden ser más complejas)
│       ├── __init__.py
│       └── # test_file_upload_view.py (ejemplo)
│
├── data/                    # Directorio para almacenar datasets de ejemplo o cargados por el usuario
│                        # (Asegúrate de añadirlo a .gitignore si no quieres versionar los datos)
│
├── .gitignore               # Especifica archivos y directorios a ignorar por Git
├── pyproject.toml           # Recomendado para gestionar dependencias y configuración del proyecto
(e.g., con Poetry o PDM)
├── README.md                # Descripción del proyecto, instrucciones de instalación y uso
└── requirements.txt         # Alternativa a pyproject.toml para listar dependencias (pip freeze >
requirements.txt)
```

4. Flujo de Trabajo y Componentes Clave

1. app/main.py:

- Inicializa la aplicación Flet (flet.app).
- Define la estructura principal de la UI (ej. layout con un panel de navegación y un área de contenido).
- Maneja el enrutamiento o cambio entre las diferentes vistas (file_upload_view, data_display_view, query_view).

2. app/views/:
   - file_upload_view.py:
     - Contendrá un control FilePicker de Flet para que el usuario seleccione archivos CSV.
     - Al seleccionar un archivo, llamará a funciones en core/data_loader.py.
     - Actualizará el estado de la aplicación para indicar que un archivo ha sido cargado.
   - data_display_view.py:
     - Se activará cuando un dataset esté cargado.
     - Utilizará funciones de core/data_analyzer.py para obtener conteos y encabezados.
     - Mostrará esta información usando controles de Flet (ej. Text, DataTable).
   - query_view.py:
     - Proporcionará un campo de texto (TextField) para que el usuario ingrese consultas SQL (o la sintaxis que decidas).
     - Al ejecutar la consulta, llamará a core/query_engine.py.
     - Mostrará los resultados en un flet.DataTable.
     - Podría incluir opciones para generar gráficos a partir de los resultados, llamando a core/plot_generator.py y mostrando el gráfico en un flet.Image (si guardas el plot como imagen) o un control de Matplotlib integrado con Flet si existe una buena solución (Flet tiene matplotlib.FletPlotAgg para esto).
3. core/:
   - data_loader.py:
     - load_csv(file_path): Lee un CSV usando pd.read_csv(), maneja errores básicos de carga. Retorna un DataFrame.
   - data_analyzer.py:
     - get_row_count(df): Retorna len(df).
     - get_column_count(df): Retorna len(df.columns).
     - get_headers(df): Retorna df.columns.tolist().
   - query_engine.py:
     - execute_query(df, query_string):
       - Crea una conexión DuckDB en memoria.
       - Registra el DataFrame de Pandas con DuckDB (duckdb.register('my_table', df)).

        - Ejecuta la query_string SQL (ej. SELECT * FROM my_table WHERE ...).
        - Retorna el resultado como un nuevo DataFrame de Pandas.
    - plot_generator.py:
        - Funciones como generate_histogram(df, column_name), generate_scatterplot(df, x_col, y_col).
        - Usarán Matplotlib/Seaborn para crear el gráfico.
        - Pueden retornar el objeto Figure de Matplotlib, o guardar el gráfico en un buffer de bytes (ej. io.BytesIO) para ser mostrado en Flet. Flet puede integrar directamente figuras de Matplotlib usando flet.matplotlib_chart.MatplotlibChart.

5. Sostenibilidad y Mantenimiento

- **Separación de Responsabilidades:** La división entre app (UI) y core (lógica) es crucial. La lógica de negocio en core no debe depender de Flet, lo que facilita su prueba y reutilización.
- **Modularidad:** Dividir las vistas y la lógica del core en archivos más pequeños y enfocados.
- **Gestión de Estado:** Decide cómo manejarás el estado de la aplicación (ej. el DataFrame cargado actualmente). Flet tiene mecanismos para el manejo de estado que deberás explorar. Puede ser tan simple como pasar datos entre funciones/métodos o usar patrones más avanzados si la app crece.
- **Pruebas Automatizadas:** Escribe pruebas para tu lógica en core usando pytest. Esto te dará confianza para refactorizar y añadir nuevas funcionalidades.
- **Control de Versiones:** Usa Git desde el inicio.
- **Entorno Virtual:** Usa un entorno virtual (como venv o los gestionados por Poetry/PDM) para aislar las dependencias de tu proyecto.
- **Documentación y Comentarios:** Comenta tu código, especialmente las partes complejas. Un buen README.md es esencial.
- **Manejo de Errores:** Implementa un buen manejo de errores (try-except bloques) tanto en la carga de datos como en la ejecución de consultas, y muestra mensajes amigables al usuario en la UI.

6. Pasos Siguientes Sugeridos

1. **Configura tu Entorno:**
    - Crea la estructura de directorios.
    - Inicializa Git (git init).
    - Configura tu gestor de dependencias (pyproject.toml con Poetry o PDM, o un requirements.txt y venv).
    - Instala las librerías base.
2. **Crea la Aplicación Flet Mínima:**

    - En app/main.py, crea una ventana simple con Flet.
3. **Implementa la Carga de Archivos:**
    - Crea file_upload_view.py y core/data_loader.py.
    - Logra cargar un CSV y almacenar el DataFrame (quizás en una variable global simple al inicio, o un objeto de estado más estructurado).
4. **Muestra Información Básica:**
    - Crea data_display_view.py y core/data_analyzer.py.
    - Muestra el número de filas, columnas y los encabezados del DataFrame cargado.
5. **Implementa el Motor de Consultas:**
    - Crea query_view.py y core/query_engine.py usando DuckDB.
    - Permite al usuario ejecutar consultas SQL y ver los resultados en una tabla Flet.
6. **Añade Visualizaciones:**
    - Integra core/plot_generator.py y muestra gráficos en la UI.
7. **Refina y Añade Pruebas:**
    - Mejora la UI, maneja errores, escribe pruebas.