**Homework entregable Henry - EcoWatch**

# **Resumen**

El siguiente trabajo funciona como una herramienta integral para el procesamiento, validación y análisis de registros ambientales obtenidos desde sensores o sistemas IoT, almacenados en archivos de texto con formato CSV. Está diseñado para automatizar la carga de datos, detectar errores comunes (campos vacíos, formatos inválidos, etc.), organizar los datos válidos en memoria temporal (caché) y generar reportes útiles tanto en formato tabular como visual (gráficos).

El sistema permite al usuario:

* Cargar fácilmente múltiples archivos de datos desde un directorio de entrada.
* Validar y filtrar automáticamente registros inválidos, manteniendo trazabilidad de errores.
* Generar reportes personalizables en función del tipo de análisis requerido.
* Exportar los resultados en formatos estándar como .csv o .xlsx.
* Visualizar los datos mediante gráficos automáticos si los reportes lo permiten.
* Extender fácilmente la funcionalidad gracias a su diseño modular, reutilizable y basado en patrones de diseño como factory y decorator.

Este enfoque modular permite una rápida adaptación a distintos entornos de monitoreo ambiental, proyectos educativos, tareas de mantenimiento predictivo o soluciones industriales que requieran análisis automatizado de condiciones ambientales como temperatura, humedad y concentración de CO₂.

## **Componentes Principales**

### **main.py**

Interfaz principal, donde se maneja la experiencia de usuario mediante menús de selección de opciones. Permite al usuario:

* Ver los archivos disponibles en la carpeta /*entradas*
* Seleccionar los archivos a procesar
* Ejecutar validaciones, mostrar estadísticas básicas y elegir reportes personalizados
* Elegir exportar los reportes a formato .csv o .xlsx en la subcarpeta */salidas*

### **LogsManager**

Clase que se encarga de:

* Listar todos los archivos válidos (.csv, .json) de una carpeta
* Mostrar los archivos disponibles para su procesamiento al usuario
* Permitir la selección por índice en base al menú indicado

Atributos principales

* base\_path: Ruta completa al directorio donde se encuentran los archivos (por defecto, /entradas)
* extensions: Tupla con las extensiones de archivo válidas (por defecto, (".csv", ".json"))
* files: Lista de archivos disponibles que cumplen con las extensiones válidas

### **Log**

Clase encargada de:

* Leer archivos CSV y validar los registros (timestamp, campos vacíos).
* Almacenar logs válidos en un objeto **RegistroCache**.
* Guardar errores en *errores\_logs.csv* con su justificación.
* Mostrar resumen general (cantidad total, válidos, inválidos, top 3 errores).
* Generar reportes personalizados a partir de una **fábrica de reportes**.
* Exportar resultados a .csv o .xlsx.
* Graficar reportes (si así lo requiere el usuario) y guardar como .png.

### **Decoradores**

Contiene funciones decoradoras reutilizables para facilitar el debugging y el benchmarking de funciones críticas del flujo de datos. Estos decoradores están aplicados, por ejemplo, a **leer\_logs()** y **generar\_reporte\_personalizado()** para facilitar debugging y análisis de rendimiento.

Atributos principales

No posee atributos de instancia. Todas las funciones son **@staticmethod**

Decoradores disponibles

* **log\_ejecucion(func)**: imprime mensajes antes y después de la ejecución de una función.
* **medir\_tiempo(func)**: mide el tiempo de ejecución de una función e imprime el resultado.

### **RegistroCache**

Clase encargada de la gestión de un sistema de caché temporal en memoria, que mantiene los registros dentro de una ventana de tiempo configurable (en minutos). Cada registro debe contener un campo 'timestamp' en formato ISO 8601 (ej. '2025-05-29T14:30:00').

Atributos:

* ventana: Duración de la ventana de tiempo en la que se consideran válidos los registros.
* cache: Estructura de datos tipo deque (cola doble) para almacenar los registros ordenados por tiempo de llegada.

### **Reportes**

#### **ReporteBase**

Clase base abstracta para todos los reportes. Define la interfaz que deben implementar los reportes concretos.

Métodos obligatorios

* **generar\_reporte(logs\_validos)**: debe ser implementado por cualquier subclase para generar el reporte correspondiente.

#### **ReporteFactory**

Implementa el patrón de diseño *Factory*. Devuelve una instancia del reporte solicitado a partir de un identificador de texto. Permite generar reportes como:

* Estado de sala
* Alertas críticas
* Temperatura promedio

Cada reporte es una clase distinta que sigue una interfaz común (**generar\_reporte**).

Atributos principales

No tiene atributos de instancia, funciona como método estático.

Método clave

* **crear\_reporte(tipo: str)**: devuelve una instancia del tipo de reporte solicitado (**estado\_sala**, **alertas\_criticas**, **reporte\_temperatura\_promedio**)

#### **ReporteAlertasCriticas**

Reporte que identifica logs con CO₂ > 1000 o temperatura > 30°C como alertas críticas.

Atributos principales

No posee atributos propios en el constructor. Se comporta como clase funcional a partir de los datos pasados en ejecución.

Método clave

* **generar\_reporte(logs\_validos: list)**: genera un DataFrame con los registros críticos filtrados.

#### **ReporteEstadoSala**

Reporte que calcula la temperatura y humedad promedio por sala.

Atributos principales

No posee atributos propios. Utiliza directamente los datos entregados.

Método clave

* **generar\_reporte(logs\_validos: list):** procesa los datos y retorna las estadísticas agrupadas por sala.

#### **ReporteTemperaturaPromedio**

Reporte que calcula la temperatura promedio por sala.

Atributos principales

No posee atributos definidos internamente, trabaja sobre el input parametrizado datos.

Método clave

* **generar\_reporte(datos: list):** devuelve un DataFrame con la temperatura media por sala.

## **Ejecución - Ciclo de vida del programa**

1. Se inicia el script main.py.
2. Se muestran los archivos disponibles en la carpeta de entrada.
3. El usuario selecciona uno o varios archivos.
4. Por cada archivo:
   1. Se leen y validan los registros.
   2. Se almacenan registros válidos en caché.
   3. Se guardan registros inválidos con errores en *errores\_logs.csv*.
   4. Se muestra un resumen de validación.
5. El usuario elige si desea generar un reporte.
6. Se le permite elegir el tipo de reporte, exportar el archivo y/o visualizar un gráfico.
7. El proceso puede repetirse con otros archivos disponibles.

## **Requisitos**

Versión de Python

* Python 3.8 o superior

Dependencias de Python

El proyecto requiere las siguientes bibliotecas para su ejecución y funcionamiento completo. Todas están disponibles dentro del archivo de requirements.txt y se pueden instalar por medio de pip install -r requirements.txt

Listado:

* + pandas – procesamiento y análisis de datos tabulares
  + matplotlib – generación de gráficos y visualizaciones
  + numpy – soporte para operaciones numéricas
  + tabulate – visualización de datos tabulares en consola
  + python-dotenv – carga de variables de entorno desde archivos .env
  + pillow – soporte para imágenes (requerido por matplotlib)
  + python-dateutil – manejo avanzado de fechas y tiempos
  + pytz – soporte para zonas horarias
  + tzdata – base de datos de zonas horarias (complemento para pytz)
  + six – compatibilidad entre Python 2 y 3 (utilizado por dependencias)
  + cycler – combinación de estilos en gráficos (requerido por matplotlib)
  + kiwisolver – resolución de ecuaciones simbólicas (usado en matplotlib)
  + pyparsing – análisis de expresiones (usado por matplotlib y otras)
  + packaging – manejo de versiones y dependencias
  + fonttools – manipulación de fuentes (requerido por matplotlib)
  + contourpy – generación eficiente de contornos en gráficos (matplotlib)