Angela Yereth Burbano Valdivieso

Ingeniería Electrónica

UPTC

Guía de Persistencia de Datos

Esta guía está dirigida a estudiantes de ingeniería electrónica con fundamentos en estructuras de datos. Se busca proporcionar un enfoque práctico y reflexivo sobre la importancia de almacenar datos de forma permanente, utilizando archivos `.txt`, `.json` y `.csv`, que son ampliamente usados en sistemas embebidos, registro de sensores, monitoreo y otras aplicaciones reales.

Problema incial: *Supón que necesitas registrar datos de sensores en un sistema embebido, o anotar tus observaciones diarias durante un experimento de laboratorio. Esos datos no pueden quedar en memoria volátil. Necesitas guardarlos, procesarlos y reutilizarlos. La persistencia de datos es esencial para llevar historiales, hacer estadísticas, generar reportes y validar comportamientos en el tiempo.*

Parte 1: Archivos de Texto Plano (.txt)

Introducción a TXT

Un archivo `.txt` es un archivo de texto plano que contiene únicamente caracteres codificados (normalmente en ASCII o UTF-8) sin ningún tipo de formato (negritas, colores, imágenes, etc.). Es ampliamente utilizado para guardar información de forma sencilla, legible y fácilmente manipulable por humanos o por programas.

En Python, los archivos `.txt` permiten leer, escribir o modificar datos mediante funciones integradas que simplifican enormemente la tarea de persistencia de datos. Esto lo convierte en una herramienta básica para registrar eventos, guardar resultados de sensores, o almacenar logs de programas de control electrónico.Son ideales para registros simples como logs, notas, instrucciones o comentarios.

Sintaxis básica

Cuando se trabaja con archivos en Python, la función principal que se utiliza es:

```
open(nombre_archivo, modo)
```

El segundo parámetro (modo) indica **cómo se abrirá el archivo**. Los modos más comunes son:

Modo "r" — Lectura

Abre un archivo existente para leerlo. Si el archivo no existe, se lanza una excepción `FileNotFoundError`.

Por ejemplo:

```
with open("datos.txt", "r", encoding="utf-8") as archivo:
    contenido = archivo.read()
    print(contenido)
```

En el anterior código:

`with open("datos.txt", "r", encoding="utf-8") as archivo:` Abre el archivo `datos.txt` en modo de lectura `"r"` y `encoding="utf-8"` especifica la codificación de caracteres que se utilizará para el archivo. `utf-8` es una codificación estándar y muy recomendada que puede manejar una amplia gama de caracteres de diferentes idiomas.

Además, el uso de `with` asegura que el archivo se cierre automáticamente al terminar.

`with ... as ...:` Es una construcción en Python conocida como el "gestor de contexto" `context manager`. Su principal beneficio en el manejo de archivos es que garantiza que el archivo se cerrará correctamente después de que el bloque de código dentro del with se haya ejecutado, incluso si ocurren errores.

Después de que el bloque `with` termina (ya sea normalmente o debido a una excepción), el gestor de contexto se encarga automáticamente de cerrar el archivo, incluso si no se llamó explícitamente a `f.close()`.

`contenido = archivo.read()` Lee todo el contenido del archivo como una única cadena de texto y lo guarda en la variable contenido.

`print(contenido)` Imprime en pantalla el contenido leído.

Modo "w" — Escritura (sobrescribe)

Crea un archivo nuevo o sobrescribe el archivo existente. Si ya existía, borra su contenido.

Por ejemplo:

```
with open("registro.txt", "w") as f:
    f.write("Sensor activado.\nVoltaje = 3.3V")
```

Como el resultado de la función `open("registro.txt", "w")` es un objeto que representa el archivo abierto.

`as f:` asigna este objeto de archivo a la variable llamada `f`.

A partir de este punto, dentro del bloque de código indentado debajo de la línea `with open(...) as f:`, se puede utilizar la variable `f` para interactuar con el archivo, tal como se hace con `f.write(...)` para escribir datos en el archivo.

Comparandolo con el ejemplo anterior se puede observar que no necesariamente debe utilizarse `encoding="utf-8"` como argumento, es una práctica recomendada por la amplia gama de carateres que maneja.

Modo "a" — Agregar (append)

Crea un archivo nuevo o sobrescribe el archivo existente. Si ya existía, borra su contenido.

Por ejemplo:

```
with open("datos.txt", "a") as dt:
    dt.write("Nueva línea de datos registrada.\n")
```

`"a"` es el modo append, se abre el archivo para agregar contenido al final sin borrar nada.

Y `df.write(...)` agrega esa línea al final del archivo.

Modo "x" — Creación exclusiva

Crea un nuevo archivo, pero si ya existe lanza un error `FileExistsError`.

Por ejemplo:

```
try:
    with open("datos_nuevo.txt", "x", encoding="utf-8") as dn:
        dn.write("Creando archivo nuevo exclusivamente.")
except FileExistsError:
    print("El archivo ya existe.")
```

`try:` intenta ejecutar el bloque de código que le sigue. Si ocurre una excepción (un error) durante la ejecución de este bloque, Python buscará un bloque `except` que pueda manejar esa excepción.

`"x"` Es el modo de apertura exclusiva para creación. Tiene el siguiente comportamiento:

- Si el archivo `"datos_nuevo.txt"` no existe, se creará un nuevo archivo vacío y se abrirá para escritura con el uso de `dn.write(...)` escribe una línea en ese nuevo archivo.
- Si el archivo `"datos_nuevo.txt"` ya existe, la función `open()` lanzará una excepción de tipo `FileExistsError` y pasará al `except` del código, donde imprimirá `"El archivo ya existe."`

Leer línea por línea

La lectura línea por línea es un principio muy importante, especialmente cuando trabajamos con archivos largos (por ejemplo, un log de miles de líneas, *un log es un archivo que contiene un registro cronológico de lo que ocurre dentro de un sistema operativo, una aplicación de software, un dispositivo de red o cualquier otro sistema electrónico.*).

Pero debe usarse con cuidad, pues leer todo el archivo de una vez puede ocupar mucha memoria.

En su forma más básica, se puede usar un bucle for sobre el archivo abierto:

```
with open("registro.txt", "r") as r:
    for linea in fr:
        print(linea.strip()) #strip elimina saltos de linea al final
```

`for linea in fr:` recorre el archivo línea por línea, lo cual es eficiente para archivos grandes.

`linea.strip()` elimina espacios o saltos de línea `\n` al principio y al final de cada línea.

`print(...)` muestra la línea limpia.

Esto es especialmente útil si estás procesando datos provenientes de sensores, por ejemplo: una línea por cada muestra tomada por un sensor de humedad.

Manejo de errores

En programación real, los errores pueden ocurrir por muchas razones, como que el archivo no exista, que no tengas permiso de lectura/escritura o que se interrumpe la conexión con el medio de almacenamiento.

Por eso, es buena práctica utilizar bloques `try-except` para manejar errores de forma controlada, como en ejemplo de la aplicación del modo `"x"`

Otro ejemplo puede ser este:

```
try:
    with open("datos_sensor.txt", "r", encoding="utf-8") as ds:
        for linea in ds:
            print(linea.strip())
except FileNotFoundError:
    print("El archivo no fue encontrado.")
except PermissionError:
    print("No tienes permiso para abrir este archivo.")
```

```
except Exception as e:
    print(f"Ocurrió un error inesperado: {e}")

```

En este caso más amplio, lo primero que se va a ejecutar es el `try:` intenta abrir y leer el archivo. Donde `for linea in archivo:` recorre línea por línea el contenido y `print(linea.strip())` imprime cada línea, quitando el salto de línea final.

Si por algún motivo no es posible completar el `try`, se pueden exponer 3 maneras de atuar frente al error

`except FileNotFoundError:` captura error si el archivo no existe.

`except PermissionError:` captura error si no tienes permiso para abrir el archivo.

`except Exception as e:` captura cualquier otro error y lo muestra con su mensaje

Buenas prácticas

Como se mencionaba anteriormente, se puede compilar código sin muchos elementos, sin embargo, con el fin de enrriquecer nuestro trabajo y plantear soluciones óptimas, se recomiendan las siguientes prácticas:

- Usar `with` para asegurar que el archivo se cierre correctamente.
- Validar existencia del archivo antes de lectura.
- Usar `strip()` para limpiar saltos de línea.
- Para el caso de uso de sensores, incluye fecha y hora para cada lectura.

Por ejemplo:

```
from datetime import datetime

def registrar_humedad(humedad):
    ahora = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    linea = f"{ahora} - Humedad: {humedad}%\n"
    with open("humedad_log.txt", "a", encoding="utf-8") as rh:
        rh.write(linea)
```

`from datetime import datetime` permite usar la hora y fecha actual.

`ahora = datetime.now().strftime(...)` obtiene la hora y la formatea en formato legible (ej. 2025-04-10 20:30:00).

`linea = f"..."` crea una cadena con la hora y el dato de humedad.

`open("humedad_log.txt", "a", ...)`: abre el archivo en modo agregar.

`rh.write(linea)` escribe la línea en el archivo con el dato de humedad y la hora.

Desafíos

A manera e reto, propón un programa para dar solución a los siguientes planteamientos.

1. Implementar un buscador de palabras clave.
2. Registrar logs de errores de un sistema.

Parte 2: Archivos JSON (.json)

Introducción a JSON

Un archivo .json (JavaScript Object Notation) es un formato de texto ligero para almacenar y transportar datos estructurados, similar a los diccionarios en Python. Es ampliamente utilizado para la transmisión de datos entre cliente y servidor.

Estructura

La estructura de un JSON, es parecida a un diccionario, a diferencia de un archivo plano, ofrece más organización ideal para manejo de muchas variables. Como en el siguiente ejemplo

```
{
  "nombre": "Lucía",
  "edad": 25,
  "activo": true,
  "habilidades": ["Python", "HTML", "CSS"],
  "direccion": {
    "ciudad": "Bogotá",
    "pais": "Colombia"
  }
}
```

Sintaxis básica

Python tiene una librería básica incorporada llamada `json` que permite manipular fácilmente este tipo de archivos. El programa a continunación muestra como convertir un diccionario a JSON.

Además, se utilizan tres modos ya mencionados, `"r" "w" "a"`

```
import json

# Convertir de diccionario a JSON y guardar
diccionario = {
    "nombre": "Laura",
    "edad": 25
}

with open("archivo.json", "w", encoding="utf-8") as archivo:
    json.dump(diccionario, archivo)

# Leer y convertir de JSON a diccionario
with open("archivo.json", "r", encoding="utf-8") as archivo:
    datos = json.load(archivo)
    print(datos["nombre"])

```

La primera parte del código nos muestra como convertir un diccionario a un archivo JSON

```
diccionario = {
    "nombre": "Laura",
    "edad": 25
}
```

Se define un diccionario Python llamado diccionario. Este diccionario tiene dos pares clave-valor: La clave `"nombre"` con el valor `"Laura"` (un string). La clave `"edad"` con el valor `25` (un entero)

```
with open("archivo.json", "w", encoding="utf-8") as archivo:
    json.dump(diccionario, archivo)
```

Utiliza los modos vistos anteriormente, en este caso abre el archivo llamado `"archivo.json"` en modo escritura (`"w"`)

`json.dump(diccionario, archivo`) Es la función clave del módulo json para convertir un objeto Python (en este caso, el diccionario "diccionario") a su representación en formato JSON y escribirlo en el archivo referenciado por la variable archivo.

Esta segunda parte nos permite leer el archivo creado anteriormente

```
with open("archivo.json", "r", encoding="utf-8") as archivo:
    datos = json.load(archivo)
    print(datos["nombre"])
```

`with open("archivo.json", "r", encoding="utf-8") as archivo:` Abre el mismo archivo `"archivo.json"` que se creó anteriormente, pero esta vez en modo lectura (`"r"`).

También se especifica la codificación `"utf-8"` para leer el contenido correctamente. El objeto del archivo se asigna a la variable archivo.

`datos = json.load(archivo)` Es otra función clave del módulo json. Lee el contenido del archivo JSON referenciado por archivo y lo convierte de nuevo a un objeto Python (en este caso, un diccionario). El diccionario resultante se guarda en la variable datos.

`print(datos["nombre"])` Accede al valor asociado con la clave `"nombre"` dentro del diccionario datos y lo imprime en la consola. En este caso, la salida será `"Laura"` .

Buenas prácticas

- Validar estructura antes de guardar.
- Usar `indent` para mejorar legibilidad.
- Manejar excepciones con `try/except` . en general las buenas prácticas no son algo exclusivo de la permanecnia de datos, si no de la programación en general.

Retos

1. Leer un archivo JSON con múltiples registros.
2. Agregar nueva información y volver a guardar el archivo.

Parte 3: Archivos CSV (.csv)

Introducción a CSV

Un archivo CSV (Comma Separated Values) es un archivo de texto plano donde los datos están separados por comas y tiene formato tabular, donde cada línea representa una fila. Es muy usado en hojas de cálculo y exportaciones de bases de datos.

Estructura base de CSV (como hoja de cálculo)

```
Nombre,Edad,Correo
Lucía,25,lucia@correo.com
Pedro,30,pedro@correo.com
```

Cada línea representa una fila. Cada valor está separado por comas (,) — o punto y coma (;) en algunos países.

Sintaxis básica

Crear CSV

```
import csv
with open("datos.csv", "w", newline="") as dt:
    writer = csv.writer(dt)
    writer.writerow(["Hora", "Voltaje"])
    writer.writerow(["10:00", 3.3])
```

Como ya hemos visto, la primera parte abre un archivo llamado `"datos.csv"` en modo escritura (`"w"`)

`newline=""` es un argumento importante cuando se trabaja con archivos CSV en Python. Ayuda a evitar la inserción de líneas en blanco adicionales entre las filas escritas en el archivo CSV.

Diferentes sistemas operativos utilizan diferentes convenciones para los finales de línea, y `newline=""` asegura que el módulo csv controle el manejo de las nuevas líneas de manera consistente.

`writer = csv.writer(dt)` Crea un objeto writer utilizando la función `csv.writer` (). Este objeto writer es el que se utilizará para escribir datos en el archivo CSV referenciado por la variable dt. El `csv.writer()` se encarga de formatear los datos proporcionados para que se escriban correctamente en el formato CSV (separados por comas por defecto).

`writer.writerow(["Hora", "Voltaje"])` Utiliza el método `writerow()` del objeto writer para escribir una fila en el archivo CSV. Se le pasa una lista `["Hora", "Voltaje"]` . Esta lista representa la primera fila del archivo CSV, que actuará como la cabecera o los nombres de las columnas. En el archivo `"datos.csv"` , esta línea se traducirá a:

```
Hora,Voltaje
```

`writer.writerow(["10:00", 3.3])` Nuevamente, se utiliza el método `writerow()` para escribir otra fila en el archivo CSV. Se le pasa la lista `["10:00", 3.3]` . Esta lista representa una fila de datos, donde `"10:00"` es el valor para la columna `"Hora"` y `3.3` es el valor para la columna `"Voltaje"` . En el archivo `"datos.csv"` , esta línea se añadirá como:

```
10:00,3.3
```

Leer CSV

```
with open("datos.csv", "r") as dt:
    reader = csv.reader()
    for fila in reader:
        print(fila)
```

`reader = csv.reader(dt)` Crea un objeto reader utilizando la función `csv.reader()` . Este objeto reader se encarga de iterar sobre las líneas del archivo CSV referenciado por la variable `dt` . Por defecto, el `csv.reader()` asume que los valores dentro de

cada fila están separados por comas. Cada vez que se itera sobre el reader, devuelve una lista de strings, donde cada string representa un valor en la fila.

`for fila in reader:` Es un bucle for que itera sobre cada fila del archivo CSV a través del objeto reader. En cada iteración, la variable fila contendrá una lista de strings representando los valores de la fila actual.

Dentro del bucle for, `print(fila)` imprime el valor de la variable fila. Como fila es una lista de strings (cada elemento es un valor de la fila CSV), esta línea imprimirá cada fila del archivo CSV como una lista en la consola.

CSV con diccionarios

```
import csv

with open("datos.csv", "w", newline="") as dt:
    campos = ["Tiempo", "Corriente"]
    writer = csv.DictWriter(dt, fieldnames=campos)
    writer.writeheader()
    writer.writerow({"Tiempo": "12:00", "Corriente": 0.5})
```

`campos = ["Tiempo", "Corriente"]` Se define una lista llamada campos. Esta lista contiene los nombres de las columnas que se utilizarán como encabezado en el archivo CSV.

`writer = csv.DictWriter(dt, fieldnames=campos)` Crea un objeto writer utilizando la función `csv.DictWriter()`.

`dt` Es el objeto del archivo CSV abierto en modo escritura.

EL argumento `fieldnames=campos` es crucial. Le indica al DictWriter cuáles son los nombres de las columnas y el orden en que deben aparecer en el archivo CSV. Se utiliza la lista campos definida anteriormente.

`writer.writeheader()` usa el método del objeto writer, y escribe la fila de encabezado en el archivo CSV. Utiliza los nombres de los campos proporcionados en el argumento fieldnames al crear el DictWriter. En este caso, escribirá la línea:

```
Tiempo,Corriente
```

`writer.writerow({"Tiempo": "12:00", "Corriente": 0.5})` escribe una fila de datos en el archivo CSV.

A diferencia de `csv.writer().writerow()`, `csv.DictWriter().writerow()` espera un diccionario como argumento.

Las claves del diccionario `"Tiempo" , "Corriente"` deben coincidir con los nombres de los campos especificados en fieldnames.

Los valores del diccionario `"12:00" , 0.5` son los datos que se escribirán en las columnas correspondientes de la fila.

Después de ejecutar este código, se creará (o sobrescribirá) un archivo llamado `"datos.csv"` con el siguiente contenido:

```
Tiempo,Corriente
12:00,0.5
```

Leer CSV con diccionarios

```
import csv

with open("datos.csv", "r") as f:
    reader = csv.DictReader(f)
    for fila in reader:
        print(fila["Tiempo"], fila["Corriente"])
```

`reader = csv.DictReader(f)` Crea un objeto reader utilizando la función `csv.DictReader()`. Esta función está diseñada para leer archivos CSV y tratar cada fila como un diccionario.

`f` Es el objeto del archivo CSV abierto en modo lectura.

Cómo funciona `DictReader`: La primera fila del archivo CSV se utiliza automáticamente como la cabecera o los nombres de las claves del diccionario. Las filas subsiguientes se leen como diccionarios, donde las claves son los nombres de las columnas de la cabecera y los valores son los datos correspondientes en esa fila.

`for fila in reader:` Es un bucle for que itera sobre cada fila del archivo CSV a través del objeto reader (que es un DictReader). En cada iteración, la variable fila contendrá un diccionario representando la fila actual. Las claves de este diccionario serán los nombres de las columnas de la cabecera del CSV.

`print(fila["Tiempo"], fila["Corriente"])` Dentro del bucle for, esta línea accede a los valores del diccionario fila utilizando las claves `"Tiempo" y "Corriente"`.

- `fila["Tiempo"]` Obtiene el valor de la columna "Tiempo" para la fila actual.
- `fila["Corriente"]` Obtiene el valor de la columna "Corriente" para la fila actual.

`print(...)` Imprime estos dos valores separados por un espacio en la consola. Si el archivo `"datos.csv"` contiene el siguiente contenido (como se creó en el ejemplo anterior):

```
Tiempo,Corriente
12:00,0.5
```

La salida del código sería:

```
12:00 0.5
```

Buenas prácticas

- Usar `newline=""` al escribir.
- Preferir `csv.DictReader` y `csv.DictWriter` con encabezados: Estas clases permiten acceder a los datos utilizando los nombres de las columnas como claves de diccionario, lo que hace que el código sea más legible y fácil de entender. En lugar de depender de los índices de las listas, trabajas con nombres significativos.
- Ser consistente con el delimitador y el carácter de comillas.

Retos

1. Guardar registros de consumo de energía en un CSV.
2. Leer CSV y calcular promedios.
3. Reescribir registros con nuevos datos.

Parte 4: Ejemplo de articulación 1

Ahora vamos a evidenciar cómo aplicar esto a un proyecto. Por ejemplo, una pequeña corporación bancaria requiere un sistema donde pueda Realizarle un CRUD *(Create, Read, Update, Delete)* al usuario, asignarle un nombre, un número de teléfono y un número de usuario *(único)*, así mismo se debe poder asignarle al usuario una cuenta de ahorros, o cuenta corriente.

Tambien queremos que haya persistencia de datos, en `.txt`, `.json` y `.csv`

Cómo comenzar a programar?

En primer lugar debemos analizar lo que nos pidieron, para ello deberíamos preguntarnos, ¿qué requerimientos tiene nuestro programa?

Según la actividad planteada, inicialmente se busca un programa que me permita crear:

1. Un usuario

Ese usuario debe tener relacionado:

a. Un nombre

b. Un número de teléfono

c. Un número de usuario (único)

Y se tiene que poder anclar al usuario una cuenta de ahorro o cuenta corriente.

2. Cuenta de ahorros

Partamos del hecho que toda cuenta debe tener un número de cuenta y, para que se catalogue como de ahorros, debe tener un interés mensual que se le aplica al saldo de la cuenta, y un aforo máximo de gasto. Por tanto debe tener relacionado:

a. Un número de cuenta

b. Una taza de interés

c. Un aforo máximo

3. Cuenta corriente

Ella no tiene una taza de interés, solo un tope de gasto, lo denominado sobregiro, por tanto, necesita relación con:

a. Un número de cuenta

b. Un monto de sobregiro

Pero, este programa tambien pide, no solo crear al usuario, si no, **MODIFICARLO** y **ELIMINARLO**

Una vez identificado lo que debemos hacer, pasamos al cómo hacerlo.

Pensémos en que para crear, modificar o eliminar algo en programación, ese algo debe existir, por ello, nuestro paso de salida es crear el usuario, y sus ***Atributos*** (Todos aquellos elementos asociados al usuario) esto indica que podemos crear un diccionario con estas características y repetir ese proceso para la cuenta de ahorros y la corriente.

Aspi escribimos el código

```
import json
import csv
from datetime import datetime
```

En esta primer parte importamos `json`, `csv` y `datetime` para la fecha, este último con el fin de organizar de mejor manera la información que vamos a almacenar en nuestros documentos.

```
usuarios = {}  #usuarios sería el Diccionario que almacena todos los usuarios del banco.
               #La clave es el id_usuario y el valor es otro diccionario con los datos del usuario y sus cuentas

#A continuacióm vamos a crear una función para registrar eventos.
def log_evento(mensaje):
    fecha_hora = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

```
    linea = f"[{fecha_hora}] {mensaje}\n"
    with open("registro.txt", "a", encoding="utf-8") as archivo:
        archivo.write(linea)
#Esta función agrega una línea con la hora actual y un mensaje a un archivo llamado registro.txt.
#Sirve como bitácora (log) del sistema

#Formulamos la función registrar un nuevo usuario
def registrar_usuario():
    id_usuario = input("Ingrese ID del usuario: ") #Pide al usuario que ingrese un ID
    if id_usuario in usuarios:
        print("El usuario ya existe.")
        return #Si el ID ya existe, muestra un mensaje de error y para el proceso de registro
    nombre = input("Ingrese nombre del usuario: ")  #Si no, contpinua con el ingreso de datos del usuario
    telefono = input("Ingrese teléfono del usuario: ")

    #Luego crea un nuevo diccionario con la info del usuario y lo guarda en el diccionario usuarios
    usuario = {
        "id_usuario": id_usuario,
        "nombre": nombre,
        "telefono": telefono,
        "cuenta_ahorros": None,
        "cuenta_corriente": None
    }

    usuarios[id_usuario] = usuario
    log_evento(f"Usuario registrado: {nombre}, Tel: {telefono}, ID: {id_usuario}")
    print("Usuario registrado.") #Muestra un mensaje que indica el registro exitoso


#Creamos la función cuenta de ahorros
def crear_cuenta_ahorros():
    id_usuario = input("Ingrese ID del usuario: ") #Pide el ID del usuario
    if id_usuario not in usuarios: #Si el user no existe se acaba el proceso
        print("Usuario no encontrado.")
        return
    if usuarios[id_usuario]["cuenta_ahorros"]: #Si user ya tiene cuenta se termina el proceso
        print("El usuario ya tiene cuenta de ahorros.")
        return

    #Si user existe y no tiene cuenta de ahorros, se solicita los datos: número de cuenta, interés y aforo máxim
    numero = input("Ingrese número de cuenta: ")
    interes = float(input("Ingrese interés (%): "))
    aforo = float(input("Ingrese aforo máximo: "))

    cuenta = {
        "numero_cuenta": numero,
        "interes": interes,
        "aforo_maximo": aforo
    }

    #Los datos se guardan en el campo "cuenta_ahorros" dentro del diccionario del usuario
    usuarios[id_usuario]["cuenta_ahorros"] = cuenta
    log_evento(f"Cuenta Ahorros creada para {id_usuario}")
    print("Cuenta de ahorros creada.")

#Hacemos la función cuenta corriente
def crear_cuenta_corriente():
    id_usuario = input("Ingrese ID del usuario: ") #Pide el ID del usuario
    if id_usuario not in usuarios: #Si el user no existe se acaba el proceso
        print("Usuario no encontrado.")
        return
    if usuarios[id_usuario]["cuenta_corriente"]: #Si user ya tiene cuenta se termina el proceso
        print("El usuario ya tiene cuenta corriente.")
        return

```

```
    #Si user existe y no tiene cuenta corriente, se solicita los datos: número de cuenta, y sobregiro
    numero = input("Ingrese número de cuenta: ")
    sobregiro = float(input("Ingrese sobregiro máximo: "))

    cuenta = {
        "numero_cuenta": numero,
        "sobregiro_maximo": sobregiro
    }

    #Los datos se guardan en el campo "cuenta_corriente" dentro del diccionario del usuario
    usuarios[id_usuario]["cuenta_corriente"] = cuenta
    log_evento(f"Cuenta Corriente creada para {id_usuario}")
    print("Cuenta corriente creada.")

#Creamos la función guardar en JSON
def guardar_datos_json(archivo="datos_banco.json"):
    data = list(usuarios.values()) #Convierte el diccionario usuarios a una lista de diccionarios (valores)
    with open(archivo, "w", encoding="utf-8") as f:
        json.dump(data, f, indent=4)
        #Escribe la lista de diccionarios en el archivo JSON proporcionado
    print(f"Datos guardados en {archivo}")

#Función cargar desde JSON
def cargar_datos_json(archivo="datos_banco.json"): #Carga el archivo llamado datos_banco de formato JSON
    try:
        with open(archivo, "r", encoding="utf-8") as f:
            data = json.load(f)
            #El archivo se abre en modo lectura ("r") y se carga
        usuarios.clear()
        #Elimina todos los elementos del diccionario usuarios, el nuevo archivo sobreescribe todo
        for u in data:
            usuarios[u["id_usuario"]] = u
            #Cada diccionario en la lista data se convierte en un usuario y se agrega al diccionario usuarios
        print(f"Datos cargados desde {archivo}")
    except FileNotFoundError: #Termina la operación si no se se encuentra el archivo JSON a subir
        print("Archivo no encontrado.")
    except Exception as e: #Si ocurre algún problema se termina el proceso
        print(f"Error al cargar datos: {e}")


#Función para guardar datos en un csv
def guardar_csv():
    with open("usuarios.csv", "w", newline='', encoding="utf-8") as f:
    # Crea el archivo usuario.csv, ahí solo estpa el id, nombre y teléfono
        writer = csv.writer(f)
        writer.writerow(["id_usuario", "nombre", "telefono"])
        for u in usuarios.values():
            writer.writerow([u["id_usuario"], u["nombre"], u["telefono"]])
    with open("cuentas_ahorros.csv", "w", newline='', encoding="utf-8") as f:
    # Crea el archivo cuentas_ahorros.csv, una línea por cada cuenta de ahorros existente
        writer = csv.writer(f)
        writer.writerow(["id_usuario", "numero_cuenta", "interes", "aforo_maximo"])
        for u in usuarios.values():
            ca = u.get("cuenta_ahorros")
            if ca:
                writer.writerow([u["id_usuario"], ca["numero_cuenta"], ca["interes"], ca["aforo_maximo"]])
    with open("cuentas_corriente.csv", "w", newline='', encoding="utf-8") as f:
    #Crea cuantas_corriente.csv, una línea por cada cuenta corriente existente
        writer = csv.writer(f)
        writer.writerow(["id_usuario", "numero_cuenta", "sobregiro_maximo"])
        for u in usuarios.values():
            cc = u.get("cuenta_corriente")
            if cc:
                writer.writerow([u["id_usuario"], cc["numero_cuenta"], cc["sobregiro_maximo"]])
    print("Datos guardados en archivos CSV.")
```

```
137
138 #Función para mostrar el menú de opciones para el operador del programa
139 def menu():
140     while True:
141         print("\n--- Menú ---")
142         print("1. Registrar usuario")
143         print("2. Crear cuenta de ahorros")
144         print("3. Crear cuenta corriente")
145         print("4. Guardar en JSON")
146         print("5. Guardar en CSV")
147         print("6. Cargar datos desde JSON")
148         print("7. Salir")
149         opcion = input("Seleccione una opción: ")
150
151         #Para cada Opción se llama a alguna de las funciones previamente creadas
152         if opcion == "1":
153             registrar_usuario()
154         elif opcion == "2":
155             crear_cuenta_ahorros()
156         elif opcion == "3":
157             crear_cuenta_corriente()
158         elif opcion == "4":
159             guardar_datos_json()
160         elif opcion == "5":
161             guardar_csv()
162         elif opcion == "6":
163             cargar_datos_json()
164         elif opcion == "7":
165             print("Saliendo del programa...")
166             break
167         else:
168             print("Opción inválida.")
169
170 #Con esta linea se ejecuta el programa solo si el archivo corre directamente y no si se importa desde otro módul
171 if __name__ == "__main__":
```

De esta manera hemos completado el ejercicio, pero, qué sucedería si por un error tipográfico en algún punto de mi código quiero llamar al nombre del usuario, y lo hago bajo la siguiente linea:

```
print(f"Nombre del usuario: {usuario['name']}")
```

Lo que pasaría es que python lanzaría un excepción con un mensaje como

```
KeyError: 'nombre'
```

Con esa advertencia simplemente nos dirigiríamos al diccionario para rectificar cómo nombramos a ese atributo y lo corregiríamos.

Sin embargo, cuando el código es más robusto, encontrar la fuente de estos errores es un proceso mucho más tedioso y demorado. En este punto entra la POO (Programación orientada a objetos).

Para el ejemplo, ese tipo de errores ocurren porque los diccionarios no están tipados ni validados automáticamente. Mientras que, con POO esto se puede evitar usando clases, donde "nombre" es un atributo claro, validado y fácil de detectar si se escribe mal.

Así que para efectos prácticos vamos a comenzar a trabajar todo código mudandolo a POO. Comencemos con el usuario de nuestro código anterior:

```
def registrar_usuario():
    id_usuario = input("Ingrese ID del usuario: ") #Pide al usuario que ingrese un ID
    if id_usuario in usuarios:
        print("El usuario ya existe.")
        return #Si el ID ya existe, muestra un mensaje de error y para el proceso de registro
    nombre = input("Ingrese nombre del usuario: ")  #Si no, contpinua con el ingreso de datos del usuario
    telefono = input("Ingrese teléfono del usuario: ")
```

Para mudar el código la primera parte es identificar las "entidades" principales del sistema.

En programación, las entidades son objetos reales del mundo que vas a modelar. Básicamente lo que definimos al inicio del ejercicio, siendo Usuario, Cuenta de Ahorros y Cuenta Corriente.

Luego de esa identificación, crearemos una Clase para cada entidad. Así mismo, lo que antes llamabamos funciones de ahora en adelante se identificarán como métodos.

Debes tener en cuenta que una clase es una plantilla que define las características y el comportamiento de los objetos. Son los bloques de construcción fundamentales de los programas orientados a objetos (POO).

Entonces, la clase Usuario se definiría de la siguiente forma:

```
class Usuario:
    def __init__(self, id_usuario, nombre, telefono):
        self.id_usuario = id_usuario
        self.nombre = nombre
        self.telefono = telefono
        self.cuenta_ahorros = None
        self.cuenta_corriente = None

    def to_dict(self):
        return {
            "id_usuario": self.id_usuario,
            "nombre": self.nombre,
            "telefono": self.telefono,
            "cuenta_ahorros": self.cuenta_ahorros.to_dict() if self.cuenta_ahorros else None,
            "cuenta_corriente": self.cuenta_corriente.to_dict() if self.cuenta_corriente else None
        }
```

Apenas comienza el código nos topamos con la palabra reservada `class` que nos permite anunciar que "Usuario" es una clase.

Posteriormente `def __init__(self, id_usuario, nombre, telefono):` define la estructura de un usuario con atributos como `id_usuario`, `nombre`, `telefono`, y referencias a sus cuentas de ahorros `(cuenta_ahorros)` y corrientes `(cuenta_corriente)`. Inicialmente, estas cuentas son `None`.

A ese proceso se lo conoce como el constructor de la clase. Se ejecuta automáticamente cada vez que se crea un nuevo objeto Usuario.

- `__init__` es un método especial que sirve para inicializar el objeto.
- `self` es una palabra clave que se refiere al propio objeto que se está creando. Siempre debe ir como primer parámetro.
- Los otros parámetros (`id_usuario`, `nombre`, `telefono`) son los datos que queremos usar para construir un usuario.

Luego tenemos estructuras de este estilo:

```
self.id_usuario = id_usuario
```

Esto significa que al atributo `id_usuario` del objeto se le va a asígnar el valor que se recibió como parámetro. Luego, se guarda el valor dentro del objeto para poder acceder a él más adelante como `usuario.id_usuario`

Lo mismo ocurre con

```
self.nombre = nombre
self.telefono = telefono
```

Por otro lado, `to_dict(self)` Es un método para convertir el objeto Usuario en un diccionario Python. Esto facilita la serialización a JSON. Notablemente, si las cuentas existen, también llama a su método `to_dict()` para obtener su representación en diccionario.

Con esta primera ejemplificación, podemos deducir que:

1. En POO la entidades de mi programa serán clases
2. Las clases siempre deben inicializarse con sus atributos bajo el método reservado `__init__` y la palabra reservada `self`
3. Todo lo que antes hacíamos con funciones, lo moverémos dentro de la clase correspondiente como métodos. (a este proceso se le conoce como encapsulamiento)
4. La creación de clases permite establecer listas de objetos en lugar de diccionarios sueltos:

- Sin POO:

```
usuarios = [
  {"id": "001", "nombre": "Ana", ...},
  {"id": "002", "nombre": "Luis", ...}
]
```

- Con POO:

```
usuarios = [
  Usuario("001", "Ana", "312...", CuentaAhorros("AH001", 10000)),
  Usuario("002", "Luis", "310...", CuentaAhorros("AH002", 50000))
]
```

Teniendo en cuenta el proceso que realizamos para la clase usuario, continuaremos mudando el código a POO

```
class CuentaAhorros:
    def __init__(self, numero_cuenta, interes, aforo_maximo): #constructor de clase
        self.numero_cuenta = numero_cuenta
        self.interes = interes
        self.aforo_maximo = aforo_maximo

    def to_dict(self):
        return {
            "numero_cuenta": self.numero_cuenta,
            "interes": self.interes,
            "aforo_maximo": self.aforo_maximo
        }
```

Define la estructura de una cuenta de ahorros con atributos como `numero_cuenta`, `interes`, y `aforo_maximo`.

`to_dict(self)` Similar a la clase Usuario, convierte el objeto `CuentaAhorros` a un diccionario.

Remplaza a la función Cuenta_de_ahorros del código sin POO.

```
class CuentaCorriente:
    def __init__(self, numero_cuenta, sobregiro_maximo):
        self.numero_cuenta = numero_cuenta
        self.sobregiro_maximo = sobregiro_maximo
```

```
5
6     def to_dict(self):
7         return {
8             "numero_cuenta": self.numero_cuenta,
9             "sobregiro_maximo": self.sobregiro_maximo
10        }
```

Define la estructura de una cuenta corriente con atributos como `numero_cuenta` y `sobregiro_maximo`.

`to_dict(self)` Convierte el objeto CuentaCorriente a un diccionario.

Remplaza la función Cuenta_corriente

```
1 usuarios = {}
```

`usuarios = {}` Es un diccionario global que se utiliza para almacenar los objetos Usuario. La clave del diccionario es el `id_usuario`, lo que permite un acceso rápido a los usuarios.

```
1 def log_evento(mensaje):
2     fecha_hora = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
3     linea = f"[{fecha_hora}] {mensaje}\n"
4     with open("registro.txt", "a", encoding="utf-8") as archivo:
5         archivo.write(linea)
```

Esta función toma un mensaje como argumento. Luego obtiene la fecha y hora actual con un formato específico (YYYY-MM-DD HH:MM:SS).

Con ello crea una línea de log con la fecha, hora y el mensaje, sbre el archivo `"registro.txt"` en modo de añadir (`"a"`), lo que significa que cada nuevo evento se escribe al final del archivo sin sobrescribir el contenido anterior.

Se especifica la codificación `utf-8` para manejar correctamente los caracteres. Finalmente escribe la línea de log en el archivo.

```
1 def registrar_usuario():
2     id_usuario = input("Ingrese ID del usuario: ")
3     if id_usuario in usuarios:
4         print("El usuario ya existe.")
5         return
6     nombre = input("Ingrese nombre del usuario: ")
7     telefono = input("Ingrese teléfono del usuario: ")
8     u = Usuario(id_usuario, nombre, telefono)
9     usuarios[id_usuario] = u
10    log_evento(f"Usuario registrado: {nombre}, Tel: {telefono}, ID: {id_usuario}")
11    print("Usuario registrado.")
```

Pide al usuario que ingrese el `ID`, `nombre` y `teléfono` del nuevo usuario. Verifica si el `ID` de usuario ya existe en el diccionario usuarios. Si existe, muestra un mensaje y termina la función.

Crea una nueva instancia de la clase Usuario, donde agrega el nuevo usuario al diccionario usuarios utilizando el `ID` como clave.

Finalmente llama a `log_evento()` para registrar la creación del usuario y muestra un mensaje de confirmación.

```
1 def crear_cuenta_ahorros():
2     id_usuario = input("Ingrese ID del usuario: ")
3     usuario = usuarios.get(id_usuario)
4     if not usuario:
5         print("Usuario no encontrado.")
6         return
```

```
    if usuario.cuenta_ahorros:
        print("El usuario ya tiene cuenta de ahorros.")
        return
    numero = input("Ingrese número de cuenta: ")
    interes = float(input("Ingrese interés (%): "))
    aforo = float(input("Ingrese aforo máximo: "))
    usuario.cuenta_ahorros = CuentaAhorros(numero, interes, aforo)
    log_evento(f"Cuenta Ahorros creada para {id_usuario}")
    print("Cuenta de ahorros creada.")
```

Pide el `ID` del usuario. Luego busca al usuario en el diccionario usuarios utilizando `usuarios.get(id_usuario)`.

Verifica si el usuario existe. Si no, muestra un mensaje y termina.

Verifica si el usuario ya tiene una cuenta de ahorros. Si la tiene, muestra un mensaje y termina.

Pide el `numero` de cuenta, el `interés` y el `aforo` máximo, para crear una nueva instancia de la clase `CuentaAhorros` y la asigna al atributo `cuenta_ahorros` del objeto Usuario.

Llama a `log_evento()` para registrar la creación de la cuenta de ahorros y muestra un mensaje de confirmación.

```
def crear_cuenta_corriente():
    id_usuario = input("Ingrese ID del usuario: ")
    usuario = usuarios.get(id_usuario)
    if not usuario:
        print("Usuario no encontrado.")
        return
    if usuario.cuenta_corriente:
        print("El usuario ya tiene cuenta corriente.")
        return
    numero = input("Ingrese número de cuenta: ")
    sobregiro = float(input("Ingrese sobregiro máximo: "))
    usuario.cuenta_corriente = CuentaCorriente(numero, sobregiro)
    log_evento(f"Cuenta Corriente creada para {id_usuario}")
    print("Cuenta corriente creada.")
```

Similar a `crear_cuenta_ahorros()`, pero para crear una cuenta corriente, pidiendo el `numero` de cuenta y el `sobregiro` máximo.

```
def guardar_datos_json(archivo="datos_banco.json"):
    data = [u.to_dict() for u in usuarios.values()]
    with open(archivo, "w", encoding="utf-8") as f:
        json.dump(data, f, indent=4)
    print(f"Datos guardados en {archivo}")
```

Crea una lista de diccionarios a partir de todos los objetos Usuario en el diccionario usuarios, utilizando el método `to_dict()` de cada usuario.

Abre un archivo JSON (por defecto `"datos_banco.json"`) en modo escritura (`"w"`) con codificación `utf-8`.

Utiliza `json.dump()` para escribir la lista de diccionarios en el archivo JSON. El argumento `indent=4` formatea el JSON para que sea más legible y finalmente muestra un mensaje de confirmación.

```
def cargar_datos_json(archivo="datos_banco.json"):
    try:
        with open(archivo, "r", encoding="utf-8") as f:
            data = json.load(f)
        usuarios.clear()
        for u in data:
            usuario = Usuario(u["id_usuario"], u["nombre"], u["telefono"])
            if u["cuenta_ahorros"]:
                ca = u["cuenta_ahorros"]
```

```
                usuario.cuenta_ahorros = CuentaAhorros(
                    ca["numero_cuenta"], ca["interes"], ca["aforo_maximo"]
                )
            if u["cuenta_corriente"]:
                cc = u["cuenta_corriente"]
                usuario.cuenta_corriente = CuentaCorriente(
                    cc["numero_cuenta"], cc["sobregiro_maximo"]
                )
            usuarios[usuario.id_usuario] = usuario
        print(f"Datos cargados desde {archivo}")
    except FileNotFoundError:
        print("Archivo no encontrado.")
    except Exception as e:
        print(f"Error al cargar datos: {e}")
```

Intenta abrir el archivo JSON en modo lectura (`"r"`) con codificación `utf-8` .

Si el archivo se encuentra, utiliza `json.load()` para cargar los datos JSON en la variable data (que será una lista de diccionarios).

Limpia el diccionario global usuarios para evitar duplicados. E itera sobre cada diccionario en data; para cada diccionario, crea una nueva instancia de Usuario y, si existen las claves correspondientes, también crea instancias de `CuentaAhorros` y `CuentaCorriente` y las asigna al usuario.

Finalmente, vuelve a poblar el diccionario usuarios con los objetos Usuario reconstruidos, muestra un mensaje de confirmación.

Incluye bloques `try...except` para manejar `FileNotFoundError` (si el archivo no existe) y otras posibles excepciones durante la carga.

```
def guardar_csv():
    with open("usuarios.csv", "w", newline='', encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["id_usuario", "nombre", "telefono"])
        for u in usuarios.values():
            writer.writerow([u.id_usuario, u.nombre, u.telefono])
    with open("cuentas_ahorros.csv", "w", newline='', encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["id_usuario", "numero_cuenta", "interes", "aforo_maximo"])
        for u in usuarios.values():
            if u.cuenta_ahorros:
                ca = u.cuenta_ahorros
                writer.writerow([u.id_usuario, ca.numero_cuenta, ca.interes, ca.aforo_maximo])
    with open("cuentas_corriente.csv", "w", newline='', encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["id_usuario", "numero_cuenta", "sobregiro_maximo"])
        for u in usuarios.values():
            if u.cuenta_corriente:
                cc = u.cuenta_corriente
                writer.writerow([u.id_usuario, cc.numero_cuenta, cc.sobregiro_maximo])
    print("Datos guardados en archivos CSV.")
```

Crea tres archivos CSV separados: `"usuarios.csv"` , `"cuentas_ahorros.csv"` , y `"cuentas_corriente.csv"` .

Para `"usuarios.csv"` , escribe una fila de encabezado con los nombres de los atributos del usuario y luego escribe una fila por cada usuario en el diccionario usuarios.

Para `"cuentas_ahorros.csv"` , escribe una fila de encabezado y luego itera sobre los usuarios. Si un usuario tiene una cuenta de ahorros, escribe una fila con el `ID` del usuario y los detalles de la cuenta de ahorros.

Para `"cuentas_corriente.csv"` , realiza una operación similar para las cuentas corrientes.

Luego muestra un mensaje de confirmación.

```
def menu():
    while True:
        print("\n--- Menú ---")
        print("1. Registrar usuario")
        print("2. Crear cuenta de ahorros")
        print("3. Crear cuenta corriente")
        print("4. Guardar en JSON")
        print("5. Guardar en CSV")
        print("6. Cargar datos desde JSON")
        print("7. Salir")
        opcion = input("Seleccione una opción: ")
        if opcion == "1":
            registrar_usuario()
        elif opcion == "2":
            crear_cuenta_ahorros()
        elif opcion == "3":
            crear_cuenta_corriente()
        elif opcion == "4":
            guardar_datos_json()
        elif opcion == "5":
            guardar_csv()
        elif opcion == "6":
            cargar_datos_json()
        elif opcion == "7":
            print("Saliendo del programa...")
            break
        else:
            print("Opción inválida.")
```

Esta función presenta un menú interactivo al usuario; utiliza un bucle while True para mantener el menú activo hasta que el usuario elija salir, luego imprime las opciones del menú. Pide al usuario que seleccione una opción y utiliza una serie de `if/elif/else` para ejecutar la función correspondiente a la opción seleccionada.

Si la opción es `"7"`, muestra un mensaje de salida y rompe el bucle.

Si la opción no es válida, muestra un mensaje de `error`.

```
if __name__ == "__main__":
    menu()
```

Este bloque asegura que la función `menu()` solo se ejecute cuando el script se ejecuta directamente (no cuando se importa como un módulo en otro script).

Parte 5: Ejemplo con sensor ultrasónico

Ya hicimos una prueba con un programa sencillo referente a un banco, ahora vamos a hacerlo con la transmisión de datos a un sensor, en este caso un ultrasónico

1. Simular los datos

Para ello vamos a recurrir a números aleatorios y un parámetro de control.

```
import json
import csv
from datetime import datetime
```

```
import random

#Se importa lo que necesitemos para el proyecto
```

```
# Clase que representa una lectura del sensor ultrasónico de distancia

class LecturaDistancia: # Clase sensor
    def __init__(self, id_sensor, distancia_cm, timestamp=None):
        self.id_sensor = id_sensor            #Se asigna un ID
        self.distancia_cm = distancia_cm      #Se brinda un atributo distancia
        self.timestamp = timestamp or datetime.now().strftime('%Y-%m-%d %H:%M:%S')   #Se guarda una fecha y hora

    def to_dict(self):
        return {
            "id_sensor": self.id_sensor,
            "distancia_cm": self.distancia_cm,
            "timestamp": self.timestamp
        }
```

Define cómo se representa una lectura del sensor ultrasónico. `__init__(self, id_sensor, distancia_cm, timestamp=None)` El constructor de la clase.

`id_sensor` Identificador único del sensor.

`distancia_cm` La distancia medida por el sensor en centímetros.

`timestamp` La fecha y hora en que se tomó la lectura. Si no se proporciona, se genera automáticamente la hora actual.

`to_dict(self)` Un método para convertir el objeto LecturaDistancia en un diccionario de Python. Esto facilita la conversión a formato JSON.

```
# Lista global para almacenar lecturas simuladas
lecturas_distancia = []
```

`lecturas_distancia = []` Es una lista vacía que se utilizará para almacenar múltiples objetos LecturaDistancia a medida que se toman las lecturas.

```
# Simulación de lectura desde un sensor ultrasónico
def tomar_lectura_distancia():

    id_sensor = input("Ingrese ID del sensor ultrasónico: ")

    # Simula distancia en centímetros, se ajusta el rango según el sensor
    distancia = round(random.uniform(5.0, 100.0), 2)
    lectura = LecturaDistancia(id_sensor, distancia)
    lecturas_distancia.append(lectura)
    print(f"Lectura registrada: {lectura.distancia_cm} cm (Sensor {id_sensor}) a las {lectura.timestamp}")
    registrar_txt_distancia(lectura)
```

Simula la acción de tomar una lectura del sensor ultrasónico:

`id_sensor = input("Ingrese ID del sensor ultrasónico: ")` Pide al usuario que ingrese un ID para el sensor.

`distancia = round(random.uniform(5.0, 100.0), 2)` Genera un número aleatorio con decimales entre 5.0 y 100.0 para simular la distancia medida en centímetros. Puedes ajustar este rango según las especificaciones de tu sensor.

`lectura = LecturaDistancia(id_sensor, distancia)` Crea una nueva instancia de la clase `LecturaDistancia` con el `ID` del sensor y la distancia simulada.

`lecturas_distancia.append(lectura)` Agrega la nueva lectura a la lista global `lecturas_distancia`.

`print(...)` Muestra en la consola la lectura registrada, incluyendo la distancia, el ID del sensor y la hora.

`registrar_txt_distancia(lectura)` Llama a otra función para guardar esta lectura en un archivo de texto.

```
# Guardar en archivo TXT para distancia
def registrar_txt_distancia(lectura):
    linea = f"[{lectura.timestamp}] Sensor {lectura.id_sensor}: {lectura.distancia_cm} cm de distancia\n"
    with open("registro_distancia.txt", "a", encoding="utf-8") as archivo:
        archivo.write(linea)
```

Guarda una única lectura en un archivo de texto llamado `registro_distancia.txt` .

`linea = f"[{lectura.timestamp}] Sensor {lectura.id_sensor}: {lectura.distancia_cm} cm de distancia\n"`

Formatea una línea de texto que incluye la hora, el `ID` del sensor y la distancia.

`with open("registro_distancia.txt", "a", encoding="utf-8") as archivo:` Abre el archivo en modo "append" (`"a"`), lo que significa que las nuevas líneas se agregarán al final del archivo sin borrar el contenido existente. Se especifica la codificación `utf-8` para manejar correctamente los caracteres.

`archivo.write(linea)` Escribe la línea formateada en el archivo de texto.

```
# Guardar datos de distancia en archivo JSON
def guardar_json_distancia(nombre_archivo="lecturas_distancia.json"):
    data = [l.to_dict() for l in lecturas_distancia]
    with open(nombre_archivo, "w", encoding="utf-8") as f:
        json.dump(data, f, indent=4)
    print(f"Datos de distancia guardados en {nombre_archivo}")
```

Guarda todas las lecturas almacenadas en la lista `lecturas_distancia` en un archivo JSON.

`data = [l.to_dict() for l in lecturas_distancia]` Crea una lista de diccionarios, donde cada diccionario representa una lectura (utilizando el método `to_dict()` de la clase `LecturaDistancia`).

`with open(nombre_archivo, "w", encoding="utf-8") as f:` Abre el archivo especificado (por defecto `lecturas_distancia.json`) en modo escritura (`"w"`), lo que sobrescribirá cualquier contenido existente.

`json.dump(data, f, indent=4)` Utiliza la función dump del módulo json para escribir la lista de diccionarios en el archivo JSON. El argumento `indent=4` formatea el JSON para que sea más legible.

`print(...)` Muestra un mensaje indicando que los datos se han guardado en el archivo JSON.

```
# Cargar datos de distancia desde JSON
def cargar_json_distancia(nombre_archivo="lecturas_distancia.json"):
    try:
        with open(nombre_archivo, "r", encoding="utf-8") as f:
            data = json.load(f)
        lecturas_distancia.clear()
        for l in data:
            lecturas_distancia.append(LecturaDistancia(l["id_sensor"], l["distancia_cm"], l["timestamp"]))
        print(f"Datos de distancia cargados desde {nombre_archivo}")
    except FileNotFoundError:
        print("Archivo no encontrado.")
    except Exception as e:
        print(f"Error al cargar datos: {e}")
```

Carga las lecturas de distancia desde un archivo JSON.

`try...except FileNotFoundError:` Intenta abrir y leer el archivo JSON. Si el archivo no existe, captura la excepción `FileNotFoundError` e imprime un mensaje.

`with open(nombre_archivo, "r", encoding="utf-8") as f:` Abre el archivo especificado en modo lectura (`"r"`).

`data = json.load(f)` Utiliza la función load del módulo json para leer el contenido del archivo JSON y convertirlo en una lista de diccionarios.

`lecturas_distancia.clear()` Borra cualquier lectura que ya esté en la lista lecturas_distancia.

`for l in data:` Itera sobre cada diccionario leído del archivo JSON.

`lecturas_distancia.append(LecturaDistancia(l["id_sensor"], l["distancia_cm"], l["timestamp"]))` Crea una nueva instancia de LecturaDistancia a partir de los datos del diccionario y la agrega a la lista lecturas_distancia.

`print(...)` Muestra un mensaje indicando que los datos se han cargado desde el archivo JSON.

`except Exception as e:` Captura cualquier otra excepción que pueda ocurrir durante la carga e imprime un mensaje de error.

```
# Guardar datos de distancia en archivo CSV
def guardar_csv_distancia(nombre_archivo="lecturas_distancia.csv"):
    with open(nombre_archivo, "w", newline='', encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["id_sensor", "distancia_cm", "timestamp"])
        for l in lecturas_distancia:
            writer.writerow([l.id_sensor, l.distancia_cm, l.timestamp])
        print(f"Datos de distancia guardados en {nombre_archivo}")
```

Guarda las lecturas de distancia en un archivo CSV.

`with open(nombre_archivo, "w", newline='', encoding="utf-8") as f:` Abre el archivo especificado (por defecto `lecturas_distancia.csv`) en modo escritura, `con newline=''` para evitar líneas en blanco adicionales en el CSV.

`writer = csv.writer(f)` Crea un objeto writer del módulo csv para escribir en el archivo.

`writer.writerow(["id_sensor", "distancia_cm", "timestamp"])` Escribe la primera fila del CSV, que contiene los encabezados de las columnas.

`for l in lecturas_distancia:` Itera sobre cada objeto LecturaDistancia en la lista.

`writer.writerow([l.id_sensor, l.distancia_cm, l.timestamp])` Escribe una nueva fila en el CSV con los atributos de cada lectura.

`print(...)` Muestra un mensaje indicando que los datos se han guardado en el archivo CSV.

```
# Menú principal modificado para distancia
def menu_distancia():
    while True:
        print("\n--- Sistema de Monitoreo de Distancia Ultrasónica ---")
        print("1. Tomar lectura de distancia")
        print("2. Guardar datos de distancia en JSON")
        print("3. Guardar datos de distancia en CSV")
        print("4. Cargar datos de distancia desde JSON")
        print("5. Salir")
        opcion = input("Seleccione una opción: ")
        if opcion == "1":
            tomar_lectura_distancia()
        elif opcion == "2":
            guardar_json_distancia()
        elif opcion == "3":
            guardar_csv_distancia()
        elif opcion == "4":
            cargar_json_distancia()
        elif opcion == "5":
            print("Finalizando monitoreo de distancia...")
            break
        else:
            print("Opción no válida.")
```

Presenta una interfaz de menú al usuario para interactuar con el sistema de monitoreo de distancia. Luego utiliza un bucle while True para mantener el menú activo hasta que el usuario decida salir. Finalmente, imprime las opciones disponibles y pide al usuario que ingrese su selección.

Según esa selección:

1: Llama a `tomar_lectura_distancia()` para simular una nueva lectura.

2: Llama a `guardar_json_distancia()` para guardar los datos en JSON.

3: Llama a `guardar_csv_distancia()` para guardar los datos en CSV.

4: Llama a `cargar_json_distancia()` para cargar datos desde JSON.

5: Imprime un mensaje de finalización y rompe el bucle para salir del programa.

Cualquier otra entrada: Muestra un mensaje de opción no válida.

```
1 if __name__ == "__main__":
2     menu_distancia()
```

`if __name__ == "__main__":` Asegura que el código dentro de este bloque solo se ejecute cuando el script se ejecuta directamente (no cuando se importa como un módulo en otro script). `menu_distancia()` Llama a la función `menu_distancia()` para iniciar la interfaz del programa.

2. Probar con el sensor

¿Qué hace?

En primera instancia, Python lee los datos seriales enviados por el Arduino.

Cada lectura se almacena temporalmente en una lista junto con una marca de tiempo.

El usuario puede elegir guardar estos datos en:

- `.txt` : como un log de eventos cronológicos.
- `.json` : en formato estructurado, ideal para intercambios con otras apps o APIs.
- `.csv` : perfecto para análisis en Excel, LibreOffice o pandas (Python).

Diagrama de Conexiones

Para conectar los componentes del HC-SR04 se puede hacer así:

--------*Sensor*---------**Arduino**------

- *Vcc* ----------> **5V**
- *GND* ----------> **GND**
- *Trig* ----------> **Un pin digital de Arduino (ej: 9)**
- *Echo* ----------> **Otro pin digital de Arduino (ej: 10)**

Código para Arduino:

El siguiente código permite enviar por el puerto serial la distancia medida en centímetros cada 500 milisegundos (ajustable).

```
const int trigPin = 9; // Pin digital para el Trigger del sensor
const int echoPin = 10; // Pin digital para el Echo del sensor
```

```
// Variables para calcular la distancia
long duracion;
int distanciaCm;

void setup() {
  Serial.begin(9600); // Inicializa la comunicación serial a 9600 baudios
  pinMode(trigPin, OUTPUT); // Configura el pin Trig como salida
  pinMode(echoPin, INPUT);  // Configura el pin Echo como entrada
}

void loop() {
  // Genera un pulso corto en el pin Trig para iniciar la medición
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // Mide la duración del pulso en el pin Echo
  duracion = pulseIn(echoPin, HIGH);

  // Calcula la distancia en centímetros (la velocidad del sonido es aprox. 0.0343 cm/microsegundo)
  distanciaCm = duracion * 0.0343 / 2;

  // Envía la distancia por el puerto serial
  Serial.print("Distancia: ");
  Serial.print(distanciaCm);
  Serial.println(" cm");

  delay(500); // Espera 500 milisegundos antes de la siguiente medición
}
```

¿Cómo Funciona?

El sensor ultrasónico `HC-SR04` funciona emitiendo un pulso de sonido ultrasónico a través del pin Trig. Cuando este pulso golpea un objeto, rebota y regresa al sensor, donde es detectado por el pin Echo.

El Arduino genera un pulso corto de alto nivel en el pin Trig para activar la emisión del sonido. Luego, mide el tiempo que tarda el pulso en regresar, utilizando la función pulseIn(echoPin, HIGH).

Esta función espera a que el pin Echo se ponga en alto y luego mide la duración del pulso en alto. La duración del pulso está directamente relacionada con la distancia al objeto.

Se utiliza una fórmula basada en la velocidad del sonido para convertir la duración en una distancia en centímetros. La división por 2 es necesaria porque el sonido viaja hacia el objeto y luego regresa.

Finalmente, el Arduino envía la distancia calculada por el puerto serial a la PC.

Código python

```
# Debe instalarse la librería pyserial con el comando "pip install pyserial"; si se está usando Jupyter "!pip inst
import serial
import time
import json
import csv
from datetime import datetime
```

`import time` Importa el módulo time para introducir pausas en la ejecución del programa (por ejemplo, para esperar la conexión serial o para dar tiempo entre lecturas).

Para este programa hemos encontrado 2 entidades principales, el sensor y el menú, por tanto crearemos dos clases, el Sensor de distancia y la aplicación del menú.

El Sensor de distancia debe contener los siguiente métodos:

1. Constructor de clase
2. Leer datos del sensor
3. Guardar datos en JSON
4. Guardar dados en CSV
5. Guardar datos en TXT
6. Cerrar la conexión

Y la Aplicación debe tener los siguiente métodos:

1. Constructor de clase
2. Menú

Así que comenzaremos con la clase sel sensor

```
class SensorDistancia:
    def __init__(self, puerto='COM3', baudios=9600):
        self.arduino = serial.Serial(puerto, baudios)
        time.sleep(2)  # Espera a que se establezca la conexión
        self.datos_distancia = []

    def leer_dato_sensor(self):
        if self.arduino.in_waiting:
            lectura = self.arduino.readline().decode().strip()
            try:
                distancia = float(lectura.split(':')[1].strip().split(' ')[0])
                fecha_hora = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
                print(f"[{fecha_hora}] Distancia: {distancia} cm")
                self.datos_distancia.append({
                    "fecha": fecha_hora,
                    "distancia": distancia
                })
            except ValueError:
                print(f"Dato no válido recibido: {lectura}")
            except IndexError:
                print(f"Formato de dato incorrecto: {lectura}")

    def guardar_txt(self, nombre_archivo="distancia_log.txt"):
        with open(nombre_archivo, "a", encoding="utf-8") as f:
            for dato in self.datos_distancia:
                f.write(f"{dato['fecha']}, Distancia: {dato['distancia']} cm\n")
        print("Datos de distancia guardados en .txt")

    def guardar_json(self, nombre_archivo="distancia.json"):
        with open(nombre_archivo, "w", encoding="utf-8") as f:
            json.dump(self.datos_distancia, f, indent=4)
        print("Datos de distancia guardados en .json")

```

```
    def guardar_csv(self, nombre_archivo="distancia.csv"):
        with open(nombre_archivo, "w", newline='', encoding="utf-8") as f:
            writer = csv.writer(f)
            writer.writerow(["fecha", "distancia_cm"])
            for dato in self.datos_distancia:
                writer.writerow([dato['fecha'], dato['distancia']])
        print("Datos de distancia guardados en .csv")

    def cerrar(self):
        self.arduino.close()
        print("Conexión cerrada.")
```

La clase `SensorDistancia` encapsula toda la lógica relacionada con la conexión al Arduino, la lectura de datos enviados desde el sensor ultrasónico y el almacenamiento de dichos datos en diferentes formatos.

Constructor de clase

El método **init** es el constructor de la clase. En él se establece la conexión con el puerto serial del Arduino:

```
def __init__(self, puerto='COM3', baudios=9600):
    self.arduino = serial.Serial(puerto, baudios)
    time.sleep(2)  # Espera a que se establezca la conexión serial
    self.datos = []
```

Aquí se definen tres cosas:

1. Se abre la conexión con el Arduino en el puerto COM3 y a 9600 baudios.
2. Se hace una pausa de 2 segundos para asegurar que la conexión esté lista antes de usarla.
3. Se inicializa una lista vacía llamada datos, donde se guardarán los registros de las lecturas.

Cuando se abre la concexión con el Arduino, 'COM3' especifica el puerto serial al que está conectado el Arduino en tu computadora. Es crucial que reemplaces 'COM3' con el puerto correcto para tu sistema operativo.

En Linux, podría ser algo como /dev/ttyUSB0, y en macOS, algo como /dev/cu.usbmodemXXXX. Puedes encontrar el puerto correcto en el IDE de Arduino (Herramientas -> Puerto).

9600 Establece la velocidad de comunicación en baudios. Este valor debe coincidir con la velocidad configurada en el código de Arduino (Serial.begin(9600)).

Método leer_dato()

Este método lee una línea enviada por el Arduino, que contiene la distancia medida por el sensor:

```
def leer_dato_sensor(self):
  if self.arduino.in_waiting:
      lectura = self.arduino.readline().decode().strip()
      try:
          distancia = float(lectura.split(':')[1].strip().split(' ')[0])
          fecha_hora = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
          print(f"[{fecha_hora}] Distancia: {distancia} cm")
          self.datos_distancia.append({
              "fecha": fecha_hora,
              "distancia": distancia
          })
      except ValueError:
          print(f"Dato no válido recibido: {lectura}")
```

```
            except IndexError:
                print(f"Formato de dato incorrecto: {lectura}")
```

Esta función intenta leer los datos enviados por el Arduino a través del puerto serial.

`if self.arduino.in_waiting:` Verifica si hay algún dato disponible para ser leído en el buffer del puerto serial.

`self.arduino.readline()` Lee una línea completa de datos desde el puerto serial hasta que encuentra un carácter de nueva línea (\n).

`.decode()` Convierte los bytes recibidos (que están en un formato binario) a una cadena de texto (string) utilizando la codificación predeterminada (generalmente UTF-8).

`.strip()` Elimina cualquier espacio en blanco al principio y al final de la cadena de texto.

`try...except ValueError...except IndexError:` Se utiliza un bloque try-except para manejar posibles errores si el formato del dato recibido no es el esperado.

`distancia = float(lectura.split(':')[1].strip().split(' ')[0])` Intenta extraer el valor numérico de la distancia del string lectura. Se asume que el Arduino envía datos con el formato `"Distancia: XXX cm"`. Donde:

1. `lectura.split(':')` Divide la cadena en dos partes usando `":"` como delimitador. Se toma la segunda parte (índice 1).
2. `.strip()` Elimina espacios en blanco alrededor de la segunda parte.
3. `[1]` toma la segunda parte (índice 1), que debería contener `" XXX cm"`
4. `.split(' ')` Divide la parte restante usando el espacio como delimitador. Se toma la primera parte (índice 0), que se espera que sea el valor numérico de la distancia.
5. `[0]` toma el primer elemento, que se espera que sea el valor numérico (por ejemplo, "15.2").

`fecha_hora = datetime.now().strftime('%Y-%m-%d %H:%M:%S')` Obtiene la fecha y hora actual y la formatea como una cadena.

`self.datos_distancia.append({"fecha": fecha_hora, "distancia": distancia})` Añade un diccionario con la fecha y la distancia a la lista datos_distancia.

`except ValueError:` Captura el error que ocurre si la parte extraída no se puede convertir a un número (por ejemplo, si el formato del dato es incorrecto).

`except IndexError:` Captura el error que ocurre si la cadena lectura no tiene el formato esperado para ser dividido correctamente (por ejemplo, si no contiene ":").

Si no hay datos disponibles (`if self.arduino.in_waiting` es falso), la función no hace nada.

Métodos guardar_txt(), guardar_json(), guardar_csv()

Estos métodos permiten guardar los datos recogidos en distintos formatos::

A. guardar_txt():

```
    def guardar_txt(self, nombre_archivo="distancia_log.txt"):
        with open(nombre_archivo, "a", encoding="utf-8") as f:
            for dato in self.datos_distancia:
                f.write(f"{dato['fecha']}, Distancia: {dato['distancia']} cm\n")
        print("Datos de distancia guardados en .txt")
```

Se realiza la persistencia como en cualquier `.txt`

Abre un archivo llamado `distancia_log.txt` en modo de añadir (`"a"`), lo que permite escribir nuevos datos al final del archivo sin borrar el contenido existente. Se utiliza la codificación `utf-8`.

Itera sobre cada diccionario en `self.datos_distancia`. Para cada lectura, escribe una línea en el archivo con la fecha y la distancia.

Finalmente imprime un mensaje indicando que los datos se han guardado en el archivo `.txt`.

B. guardar_json():

```
def guardar_json(self, nombre_archivo="distancia.json"):
    with open(nombre_archivo, "w", encoding="utf-8") as f:
        json.dump(self.datos_distancia, f, indent=4)
    print("Datos de distancia guardados en .json")
```

Abre un archivo llamado `distancia.json` en modo escritura (`"w"`), lo que sobrescribirá cualquier contenido existente. Se utiliza la codificación `utf-8`.

Utiliza la función `json.dump()` para escribir `self.datos_distancia` en el archivo JSON. El argumento `indent=4` formatea el JSON para que sea más legible.

Imprime un mensaje indicando que los datos se han guardado en el archivo `.json`.

C. guardar_csv():

```
def guardar_csv(self, nombre_archivo="distancia.csv"):
    with open(nombre_archivo, "w", newline='', encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow(["fecha", "distancia_cm"])
        for dato in self.datos_distancia:
            writer.writerow([dato['fecha'], dato['distancia']])
    print("Datos de distancia guardados en .csv")
```

Abre un archivo llamado `distancia.csv` en modo escritura (`"w"`) con `newline=''` para evitar líneas en blanco adicionales en el CSV. Se utiliza la codificación `utf-8`.

Crea un objeto writer del módulo csv; escribe la primera fila del CSV como encabezado: `["fecha", "distancia_cm"]`.

Itera sobre cada diccionario en `self.datos_distancia`; para cada lectura, escribe una nueva fila en el CSV con la fecha y la distancia (accediendo a los valores del diccionario por sus claves).

Imprime un mensaje indicando que los datos se han guardado en el archivo `.csv`.

Método cerrar()

```
def cerrar(self):
    self.arduino.close()
    print("Conexión cerrada.")
```

Este método cierra la conexión serial abierta al inicio. Es importante llamarlo antes de salir del programa para liberar correctamente el recurso.

Ahora continuarémos con la clase Aplicacion

```
class Aplicacion:
    def __init__(self):
        self.sensor = SensorDistancia()

    def mostrar_menu(self):
```

```
        while True:
            print("\n--- Menú ---")
            print("1. Leer datos del sensor de distancia")
            print("2. Guardar en TXT")
            print("3. Guardar en JSON")
            print("4. Guardar en CSV")
            print("5. Salir")

            opcion = input("Seleccione una opción: ")
            if opcion == "1":
                for _ in range(5):  # Lee 5 muestras
                    self.sensor.leer_dato_sensor()
                    time.sleep(0.5)
            elif opcion == "2":
                self.sensor.guardar_txt()
            elif opcion == "3":
                self.sensor.guardar_json()
            elif opcion == "4":
                self.sensor.guardar_csv()
            elif opcion == "5":
                print("Saliendo...")
                self.sensor.cerrar()
                break
            else:
                print("Opción inválida")
```

Esta clase se encarga de gestionar la interacción del usuario mediante un menú en consola.

Constructor de clase

```
def __init__(self):
        self.sensor = SensorDistancia()
```

El constructor recibe una instancia de la clase `SensorDistancia`, lo que permite a la aplicación acceder a todos los métodos y datos del sensor

Método mostrar_menu

```
def mostrar_menu(self):
        while True:
            print("\n--- Menú ---")
            print("1. Leer datos del sensor de distancia")
            print("2. Guardar en TXT")
            print("3. Guardar en JSON")
            print("4. Guardar en CSV")
            print("5. Salir")

            opcion = input("Seleccione una opción: ")
            if opcion == "1":
                for _ in range(5):  # Lee 5 muestras
                    self.sensor.leer_dato_sensor()
                    time.sleep(0.5)
            elif opcion == "2":
                self.sensor.guardar_txt()
            elif opcion == "3":
                self.sensor.guardar_json()
```

```
            elif opcion == "4":
                self.sensor.guardar_csv()
            elif opcion == "5":
                print("Saliendo...")
                self.sensor.cerrar()
                break
            else:
                print("Opción inválida")
```

Este es el núcleo de la interacción. El menú ofrece 5 opciones: leer datos, guardar en distintos formatos o salir. Según la opción elegida, llama al método correspondiente del sensor.

Finalmente necesitamos nuestro bloque principal para ejecutar todo lo que hemos programado

```
if __name__ == "__main__":
    app = Aplicacion()
    app.mostrar_menu()
```

En Python, el bloque `if __name__ == "__main__":` se utiliza para asegurarse de que cierto código solo se ejecute cuando el archivo es ejecutado directamente por el usuario, y no cuando es importado como un módulo desde otro script. Es una buena práctica que permite reutilizar clases y funciones en otros programas sin que el código principal se ejecute automáticamente.

Dentro de este bloque se crean las instancias principales de la aplicación. En este caso, se instancia un objeto de la clase `Aplicacion` y se llama a su método `mostrar_menu()`.

Esto significa que al ejecutar este archivo, se lanza directamente la interfaz de menú que permite al usuario interactuar con el sistema para leer los datos del sensor, guardarlos en diferentes formatos, o salir del programa. Todo el flujo del programa parte desde este punto.