1. MANEJO DE ARCHIVOS

En el desarrollo de software solemos utilizar archivos por diferentes causas:

- Almacenamiento persistente de datos: Los archivos permiten almacenar datos de forma persistente en el sistema de archivos. Esto significa que los datos no se pierden cuando se cierra el programa y se pueden acceder nuevamente en futuras ejecuciones.
- Lectura y escritura de datos: Los archivos ofrecen la capacidad de leer y escribir datos de manera eficiente. Puedes leer información desde un archivo existente para utilizarla en tu programa o escribir datos generados o procesados por tu programa en un archivo para su posterior uso.
- Manejo de grandes volúmenes de datos: Si tienes grandes cantidades de datos que no pueden almacenarse en memoria, los archivos proporcionan una forma de trabajar con esos datos en partes, leyendo o escribiendo bloques más pequeños a medida que sea necesario.
- Comunicación con otros programas: Los archivos pueden utilizarse como medio de comunicación entre diferentes programas. Por ejemplo, un programa puede generar un archivo de salida que luego es utilizado por otro programa para procesar esa información.
- Almacenamiento y carga de configuraciones: Los archivos son útiles para almacenar configuraciones de programas. Puedes utilizar archivos de texto estructurados como archivos de configuración para personalizar el comportamiento de tu programa sin necesidad de modificar el código fuente.
- Procesamiento de archivos de texto estructurados: Si trabajas con archivos de texto estructurados, como archivos CSV, archivos de registro (logs) o archivos de texto plano con un formato específico, puedes utilizar Python para leer y analizar fácilmente estos archivos, extraer la información relevante y realizar operaciones o cálculos sobre los datos.

2. Tipos de archivos

Python te permite abrir diferentes tipos de archivos, como:

- Archivos de texto (.txt, .csv, .log, etc.): Son archivos que contienen texto legible.
- Archivos binarios (.jpg, .mp3, .pdf, etc.): Son archivos que contienen datos en un formato binario específico.
- Archivos JSON (.json): Son archivos que almacenan datos en formato JSON.
- Archivos XML (.xml): Son archivos que almacenan datos en formato XML.

- Archivos de hojas de cálculo (.xlsx, .xls): Son archivos utilizados para almacenar datos en forma tabular.
- Archivos de bases de datos (.db, .sqlite): Son archivos utilizados para almacenar datos estructurados en una base de datos.

3. Apertura de archivos

Para abrir un archivo en Python, puedes utilizar la función open(). La sintaxis básica para abrir un archivo es la siguiente:

```
archivo = open('nombre_archivo', 'modo')
```

Donde:

nombre_archivo es el nombre o la ruta del archivo que deseas abrir.

modo especifica el modo de apertura del archivo, que puede ser 'r' para lectura (por defecto), 'w' para escritura, 'a' para agregar contenido al final del archivo, o 'x' para crear un archivo nuevo y escribir en él. Además, puedes agregar 'b' al modo para abrir el archivo en modo binario, por ejemplo, 'rb' para lectura binaria.

Una vez abierto el archivo, puedes realizar diferentes operaciones según el modo de apertura:

- Modo de lectura ('r'): Puedes leer el contenido del archivo utilizando métodos como read(), readline() o readlines().
- Modo de escritura ('w'): Puedes escribir en el archivo utilizando el método write().
- Modo de agregar ('a'): Puedes agregar contenido al final del archivo utilizando el método write().

Es importante cerrar el archivo después de haber terminado de trabajar con él para liberar los recursos del sistema utilizando el método close(), por ejemplo: archivo.close().

Ejemplo de apertura de un archivo en modo de lectura:

```
archivo = open('archivo.txt', 'r')

contenido = archivo.read()

print(contenido) archivo.close()
```

Es importante tener en cuenta que a partir de Python 3.6, se recomienda utilizar el bloque with para abrir archivos, ya que se encarga automáticamente de cerrar el archivo una vez que se sale del bloque, incluso si ocurren excepciones. Por ejemplo:

with open('archivo.txt', 'r') as archivo:

contenido = archivo.read() print(contenido)

El uso del bloque with hace que el cierre del archivo sea más seguro y simplifica el código.

Ejemplo:

```
# Abrir el archivo de entrada en modo de lectura with
open('entrada.txt', 'r') as archivo_entrada:     # Abrir
el archivo de salida en modo de escritura     with
open('salida.txt', 'w') as archivo_salida:         # Leer
el contenido línea por línea         for linea in
archivo_entrada:
            # Realizar operaciones con cada línea
            palabras = linea.split()  # Dividir la línea en
palabras
            num_palabras = len(palabras)  # Contar el número de
palabras
            resultado = f"Línea: {linea.strip()} - Número de
palabras: {num_palabras}"

            # Escribir el resultado en el archivo de salida
archivo_salida.write(resultado + '\n')

# Imprimir un mensaje de éxito
print("Operación completada con éxito. Se ha generado el archivo
de salida.")
```

4. Lectura de archivos

Lectura línea por línea:

Puedes utilizar un bucle for para leer el archivo línea por línea. Esto es útil cuando deseas procesar el archivo línea por línea sin cargar todo el contenido en memoria de una vez. `with open('archivo.txt', 'r') as archivo:  for linea in archivo:  # Procesar cada línea  print(linea)`

Lectura de todo el contenido:

Podés utilizar el método read() para leer todo el contenido del archivo en una sola cadena. Esto es útil cuando deseas tener el contenido completo del archivo en memoria para manipularlo.

```
with open('archivo.txt', 'r') as archivo:
```

```
    contenido = archivo.read()      #
Procesar el contenido completo
print(contenido)
```

Lectura de líneas específicas:

Podés utilizar el método readlines() para leer todas las líneas del archivo y almacenarlas en una lista. Luego, puedes acceder a líneas específicas utilizando los índices de la lista.

```
with open('archivo.txt', 'r') as archivo:
    lineas = archivo.readlines()      #
Acceder a líneas específicas
print(lineas[0])  # Primera línea
print(lineas[2])  # Tercera línea
```

5. Escritura de archivos

En Python, puedes escribir en un archivo utilizando la función write() o el método writelines(). A continuación, te muestro ejemplos de ambos métodos:

Escribir un archivo línea por línea utilizando write():

Puedes utilizar el método write() para escribir una línea a la vez en un archivo. Asegúrate de incluir los caracteres de nueva línea (\n) al final de cada línea si deseas separarlas.

```
with open('archivo.txt', 'w') as archivo:
    archivo.write('Línea 1\n')      archivo.write('Línea
2\n')      archivo.write('Línea 3\n')
```

Escribir múltiples líneas utilizando writelines():

Puedes utilizar el método writelines() para escribir varias líneas a la vez en un archivo. Este método acepta una lista de cadenas, donde cada cadena representa una línea.

```
lineas = ['Línea 1\n', 'Línea 2\n', 'Línea 3\n']
 with open('archivo.txt', 'w') as archivo:
    archivo.writelines(lineas)
```

Recordá que al abrir el archivo en modo de escritura ('w'), si el archivo ya existe, se sobrescribirá. Si deseas agregar contenido al final del archivo sin sobrescribirlo, puedes utilizar el modo de apertura 'a' en lugar de 'w'.

Es importante cerrar el archivo después de escribir para asegurarte de que los cambios se guarden correctamente. Al utilizar la declaración with open() as archivo, el archivo se cerrará automáticamente al salir del bloque with.

Estos son solo ejemplos básicos de cómo escribir en un archivo en Python. Puedes adaptar estos métodos según tus necesidades, concatenar variables, generar contenido dinámico, entre otros.

6. Archivos binarios

Un archivo binario es un tipo de archivo que almacena datos en una representación binaria, es decir, en forma de secuencias de unos y ceros. A diferencia de los archivos de texto, que almacenan caracteres legibles por humanos, los archivos binarios almacenan información en su forma más básica, sin ninguna interpretación directa.

En un archivo binario, los datos se organizan en estructuras de bits, bytes y patrones específicos que representan diferentes tipos de información, como números enteros, números de punto flotante, caracteres, imágenes, sonidos, videos, entre otros. Estos archivos se utilizan comúnmente para almacenar información compleja y detallada que no se puede representar fácilmente en formato de texto.

Los archivos binarios pueden contener cualquier tipo de información y no están limitados a ningún formato específico. Algunos ejemplos de archivos binarios comunes incluyen archivos de imágenes (como JPEG, PNG), archivos de audio (como MP3, WAV), archivos ejecutables (como EXE, DLL), archivos de bases de datos y archivos comprimidos (como ZIP, RAR).

Para trabajar con archivos binarios en Python, puedes utilizar los modos de apertura 'rb' para lectura y 'wb' para escritura. Ejemplo: leer y escribir archivos binarios en Python:

Lectura de archivos binarios:

Abre el archivo en modo de lectura binaria utilizando el modo 'rb'.

Utiliza el método read() para leer todo el contenido binario del archivo.

Procesa los datos leídos según tus necesidades.

Ejemplo de lectura de un archivo binario:

```
with open('archivo.bin', 'rb') as archivo:     contenido_binario
= archivo.read()
    # Procesa los datos binarios según tus necesidades
```

Escritura en archivos binarios:

Abre el archivo en modo de escritura binaria utilizando el modo 'wb'.

Utiliza el método write() para escribir datos binarios en el archivo.

Aquí tienes un ejemplo de escritura en un archivo binario:

```
datos_binarios = b'\x00\x01\x02\x03'  # Ejemplo de datos binarios
with open('archivo.bin', 'wb') as archivo:
    archivo.write(datos_binarios)
```

Es importante recordar que al trabajar con archivos binarios, los datos se manejan en su representación binaria directa, lo que significa que debes asegurarte de utilizar los métodos y formatos de datos adecuados para leer y escribir correctamente la información en el archivo.

Además, ten en cuenta que al utilizar archivos binarios, los datos no se pueden leer o interpretar directamente como texto. Por lo tanto, es posible que necesites realizar conversiones o manipulaciones adicionales según el formato o estructura de los datos binarios que estés manejando.

7. Archivos CSV

Un archivo CSV (Comma-Separated Values) es un tipo de archivo que se utiliza para almacenar datos tabulares en forma de texto plano. Como su nombre lo indica, los datos en un archivo CSV se separan por comas (u otro delimitador) para organizar la información en columnas y filas.

Cada línea del archivo CSV representa una fila de datos, y los valores de cada columna se separan por el delimitador, que suele ser una coma. Por ejemplo, un archivo CSV que almacena información de empleados podría tener la siguiente estructura:

```
Nombre,Apellido,Edad,Departamento
Juan,Pérez,35,Ventas
María,Gómez,42,Recursos Humanos
```

Python proporciona varias formas de trabajar con archivos CSV. Una de las formas más comunes es utilizando el módulo csv incorporado en la biblioteca estándar de Python. Con este módulo, puedes leer y escribir archivos CSV de manera sencilla.

Aquí tienes un ejemplo básico de cómo leer un archivo CSV en Python utilizando el módulo csv:

```
import csv
```

```
# Abrir el archivo CSV en modo lectura with
open('datos.csv', 'r') as archivo_csv:
    # Crear un lector de CSV     lector_csv
= csv.reader(archivo_csv)

    # Leer los datos línea por línea
for fila in lector_csv:
        # Acceder a los valores de cada columna
nombre = fila[0]          apellido = fila[1]
edad = fila[2]          departamento = fila[3]

        # Hacer algo con los datos...
```

Del mismo modo, puedes utilizar el módulo csv para escribir datos en un archivo CSV. Aquí tienes un ejemplo:

```
import csv

# Datos a escribir en el archivo CSV datos =
[
    ['Juan', 'Pérez', 35, 'Ventas'],
    ['María', 'Gómez', 42, 'Recursos Humanos']
]

# Abrir el archivo CSV en modo escritura with
open('datos.csv', 'w', newline='') as archivo_csv:
    # Crear un escritor de CSV
escritor_csv = csv.writer(archivo_csv)

    # Escribir los datos fila por fila
    for fila in datos:
        escritor_csv.writerow(fila)
```

8. Directorios

Para acceder a los directorios donde se encuentran los archivos de manera independiente del sistema operativo utilizado, puedes utilizar el módulo os de Python. El módulo os proporciona funciones y métodos para interactuar con el sistema operativo, incluyendo la manipulación de rutas de archivos y directorios.

Aquí tienes algunas funciones y métodos útiles del módulo os para trabajar con directorios:

- os.getcwd(): Devuelve la ruta del directorio de trabajo actual.
- os.chdir(ruta): Cambia el directorio de trabajo actual a la ruta especificada.
- os.listdir(ruta): Devuelve una lista con los nombres de los archivos y directorios en la ruta especificada.
- os.path.join(ruta1, ruta2): Combina dos rutas para formar una nueva ruta.
- os.path.exists(ruta): Verifica si la ruta especificada existe.
- os.path.isdir(ruta): Verifica si la ruta especificada es un directorio.

Acá tenés un ejemplo que muestra cómo obtener los nombres de los archivos en un directorio específico:

```
import os


# Obtener la ruta del directorio actual directorio_actual =

os.getcwd()


# Obtener la lista de nombres de archivos en el directorio actual
archivos = os.listdir(directorio_actual)


# Imprimir los nombres de los archivos for

archivo in archivos:

    print(archivo)
```

Tené en cuenta que el comportamiento exacto puede variar según el sistema operativo utilizado. Sin embargo, utilizando las funciones y métodos proporcionados por el módulo, podés escribir código que sea independiente del sistema operativo y funcione correctamente en diferentes plataformas.