

## 2. MANEJO DE ARCHIVOS Y ACCESO AL FILE SYSTEM



### 2.1 Introducción

El manejo de archivos de texto y datasets es crucial en la programación, especialmente en Python, debido a su amplia utilización en la ciencia de datos, análisis y desarrollo de aplicaciones. Python ofrece herramientas poderosas y eficientes para la lectura, escritura y manipulación de archivos, facilitando el procesamiento y análisis de grandes volúmenes de datos. Esto permite a los desarrolladores y científicos de datos extraer información valiosa y tomar decisiones informadas basadas en datos. Además es importante conocer sobre la manipulación del file system para el acceso a los archivos, tanto locales como remotos

### 2.2 Conceptos Básicos

#### 2.2.1 Archivo

Un archivo es una colección de datos almacenados en un dispositivo de almacenamiento, como un disco duro, SSD, o una unidad flash. Los archivos pueden contener diversos tipos de información, incluyendo texto, imágenes, audio, video, y datos binarios. Cada archivo tiene un nombre y una extensión que indica su tipo (por ejemplo, .txt para archivos de texto)

texto, .jpg para imágenes). Los archivos son fundamentales para el almacenamiento y la organización de datos en una computadora, permitiendo a los usuarios y aplicaciones guardar, recuperar y manipular información de manera estructurada.

### 2.2.2 Sistema de Archivos (File System)

El sistema de archivos (file system) es un componente del sistema operativo que gestiona la organización, almacenamiento, acceso y recuperación de datos en los dispositivos de almacenamiento. El sistema de archivos proporciona una estructura lógica para almacenar y organizar archivos y directorios (carpetas) de manera eficiente. Algunas de las funciones clave de un sistema de archivos incluyen:

**Organización:** Mantiene una estructura jerárquica de archivos y directorios.

**Acceso:** Permite a los usuarios y aplicaciones leer y escribir datos en archivos.

**Gestión de Espacio:** Asigna y libera espacio en el dispositivo de almacenamiento según sea necesario.

**Seguridad:** Controla los permisos de acceso a archivos y directorios para asegurar que solo los usuarios autorizados puedan acceder o modificar los datos.

**Integridad:** Asegura que los datos almacenados no se corrompan y se puedan recuperar en caso de errores o fallos del sistema.

**Ejemplos de sistemas de archivos:** NTFS (Windows), ext4 (Linux), HFS+ y APFS (MacOS), y FAT32 (usado en muchos dispositivos de almacenamiento portátil). Cada sistema de archivos tiene sus propias características y ventajas, adecuadas para diferentes tipos de aplicaciones y entornos.

### 2.2.3 Tipos de archivos

#### 2.2.3.1 Archivos Binarios

- Los archivos binarios contienen datos en un formato no legible directamente por humanos.
- Pueden incluir cualquier tipo de datos codificados en forma binaria, como imágenes, audio, video, archivos ejecutables, etc.

- Los datos en archivos binarios se representan como una secuencia de bytes.

## Características

No son fácilmente legibles ni editables con un editor de texto estándar.

Son más compactos y pueden almacenar datos más complejos y grandes.

Utilizan estructuras de datos específicas y requieren conocimiento del formato para interpretarlos correctamente.

Ejemplo:

```
# Escribir en un archivo binario
datos_binarios = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x00\x01...'

with open('archivo_binario.bin', 'wb') as archivo:
    archivo.write(datos_binarios)

# Leer de un archivo binario
with open('archivo_binario.bin', 'rb') as archivo:
    contenido = archivo.read()
    print(contenido)
```

### 2.2.3.2 Archivos de Texto

Un archivo plano es un tipo de archivo de almacenamiento de datos en el que los datos se almacenan como texto sin formato, a menudo en una estructura similar a una tabla con filas y columnas. Cada fila representa un solo registro, mientras que las columnas representan campos o atributos de los datos. Los formatos más comunes para archivos sin formato son valores separados por comas (CSV), valores separados por tabuladores (TSV) y archivos de texto sin formato. Los archivos planos se utilizan ampliamente por su simplicidad, facilidad de lectura humana y compatibilidad con varias plataformas y aplicaciones.

- Los archivos de texto contienen datos en un formato legible por humanos.
- Están compuestos por caracteres que representan texto, como letras, números y símbolos.

- Cada carácter en un archivo de texto está codificado usando una codificación de caracteres, como ASCII o UTF-8.

## Características

Fácilmente legibles y editables con cualquier editor de texto.

Los datos están separados por delimitadores como espacios, tabulaciones o saltos de línea.

Formatos comunes son: archivos planos (txt, csv, tsv), Json, XML, HTML

Adecuados para almacenar datos simples como configuraciones, logs, datos IoT, scripts y documentos.

Pueden almacenar conjuntos de datos complejos arrojados por procesos especiales para uso en ciencia de datos e IA llamados DATASET.

### Archivo de texto Plano

```
codigo,nombre,producto,ciudad,valor,fecha
1,Juan Perez,Laptop,Madrid,1200.5,2024-05-27
2,Maria Gomez,Smartphone,Barcelona,650.75,2024-05-26
3,Luis Martinez,Tablet,Valencia,300.0,2024-05-25
```

```
# Escribir en un archivo de texto
with open('archivo_texto.txt', 'w') as archivo:
    archivo.write("Hola, este es un archivo de texto.")

# Leer de un archivo de texto
with open('archivo_texto.txt', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
```

## Convertir un archivo de texto plano a Json

```
import json

# Datos en formato CSV
datos_csv = """
codigo,nombre,producto,ciudad,valor,fecha
1,Juan Perez,Laptop,Madrid,1200.5,2024-05-27
2,Maria Gomez,Smartphone,Barcelona,650.75,2024-05-26
3,Luis Martinez,Tablet,Valencia,300.0,2024-05-25
"""

# Dividir los datos en líneas y obtener los encabezados y las filas de datos
lineas = datos_csv.strip().split("\n")
encabezados = lineas[0].split(',')
filas = [linea.split(',') for linea in lineas[1:]]

# Convertir a formato JSON
datos_json = []
for fila in filas:
    datos_json.append({encabezados[i]: fila[i] for i in
range(len(encabezados))})

# Guardar en un archivo JSON
nombre_archivo_json = 'productos.json'
with open(nombre_archivo_json, 'w') as archivo_json:
    json.dump(datos_json, archivo_json, indent=4)

print(f"Archivo {nombre_archivo_json} creado con éxito.")
```

### 2.2.4 Diferencias texto - binario

#### **Formato y Legibilidad:**

Texto: Legible por humanos, fácil de editar.

Binario: No legible por humanos, requiere programas específicos para su edición y lectura.

#### **Uso y Aplicaciones:**



## CIENCIA DE DATOS - IA

**Texto:** Ideal para datos estructurados, datasets, ciencia de datos,, configuraciones, y scripts.

**Binario:** Ideal para datos complejos, multimedia, y archivos ejecutables.

### **Almacenamiento y Eficiencia:**

**Texto:** Menos eficiente en términos de espacio y procesamiento, ya que los datos están en un formato legible por humanos.

**Binario:** Más eficiente en términos de espacio y procesamiento, ya que los datos están en un formato que la computadora puede procesar directamente.

## 2.2.5 Operaciones Básicas con Archivos

Para la creación y Apertura de Archivos, existen varios modificadores para la apertura y creación de archivos, en python:

```
Abrir archivo para lectura : f = open("fichero.txt", "r")  
Abrir archivo para lectura en binario : f = open("fichero.txt", "rb")  
Abrir archivo para escribir en binario : f = open("fichero.txt", "wb")
```

Abrir archivo para escribir desde cero (sobreescribe)

```
                : f = open("fichero.txt", "w")  
Abrir fichero para añadir al final : f = open("fichero.txt", "a")  
Abrir fichero para lectura y escritura : f = open("fichero.txt", "r+")
```

Para cerrarlo, basta llamar a **f.close()**

### Leer de archivos¶

Para leer del archivo, podemos usar las funciones `f.read()` y `f.readline()`

```
Lectura de todo el archivo: archivo = f.read()  
Lectura de 100 caracteres : dato = f.read(100)  
Lectura de una línea completa : linea = f.readline()
```

### Moverse por el archivo¶

Con `f.tell()` podemos saber en qué posición estamos del archivo  
Con `f.seek()` podemos desplazarnos por él, para leer o escribir en una determinada posición.

```
f.seek(n) : Ir al byte n del archivo  
f.seek(n,0) : Equivalente al anterior  
f.seek(n,1) : Desplazarnos n bytes a partir de la posición actual del archivo  
f.seek(n,2) : Situarnos n bytes antes del final de archivo.
```

El segundo parámetro en estos ejemplos es  
Ninguno o 0 : la posición es relativa al principio del archivo  
1 : la posición es relativa a la posición actual  
2 : la posición es relativa al final del archivo y hacia atrás.

Referencia. [https://aulasoftwarelibre.github.io/taller-de-python/Python\\_Avanzado/Ficheros/](https://aulasoftwarelibre.github.io/taller-de-python/Python_Avanzado/Ficheros/)

### 2.2.5.1 Ciclo para crear una archivo a partir de otro:

En el ciclo se lee line por línea

```
f = open("origen.txt")
g = open("destino.txt", "w")
for linea in f:
    g.write(linea)
g.close()
f.close()
```

El archivo se puede leer con `f.readline()` que nos da una línea cada vez, incluyendo el salto de línea `\n` al final. Cuando lleguemos a final de archivo nos devolverá una línea vacía. Una línea en blanco en medio del archivo nos sería devuelta como un `\n`, no como una línea vacía `""`. El siguiente ejemplo hace la copia del archivo leyendo con `readline()` y en un bucle hasta fin de archivo.

```
f = open("origen.txt")
g = open("destino.txt", "w")
linea = f.readline()
while linea != "":
    g.write(linea)
    linea = f.readline()
g.close()
f.close()
```

Una alternativa a la manera clásica de manejar archivos como en C, es usar lo que se denomina **context managers** o "gestores de contexto":

Si el código está en el contexto **with**, no necesita cerrar el archivo. Ejemplo: leer el archivo por líneas:

```
with open("fichero.txt", "r") as f:
    for linea in f:
        print(f)

print("El fichero ya está cerrado")
```



### 2.2.5.2 Cierre de Archivos

En Python, es importante asegurarse de que los archivos se cierren correctamente después de que se hayan utilizado para evitar fugas de recursos, restringir accesos o bloquear para otros usos. Esto se puede lograr usando bloques **try...finally** y **context managers (with)**. A continuación, te muestro cómo se usan ambas técnicas.

#### Uso de **try...finally**

El bloque **try...finally** garantiza que el archivo se cierre incluso si ocurre una excepción durante la operación con el archivo.

```
try:
    archivo = open('mi_archivo.txt', 'r')
    # Realizar operaciones de lectura/escritura
    contenido = archivo.read()
    print(contenido)
finally:
    archivo.close()
```

En este ejemplo:

El archivo se abre en modo lectura ('r').

Se realizan operaciones de lectura (en este caso, leer todo el contenido del archivo).

El bloque **finally** asegura que **archivo.close()** se ejecute, cerrando el archivo independientemente de si ocurrió una excepción durante la lectura.

### 2.2.5.3 Uso de Context Managers (*with*)

El uso de context managers con la declaración **with** es una forma más elegante y preferida en Python para manejar archivos. El archivo se cierra automáticamente al salir del bloque **with**, incluso si se produce una excepción.

```
with open('mi_archivo.txt', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
```

En este ejemplo:

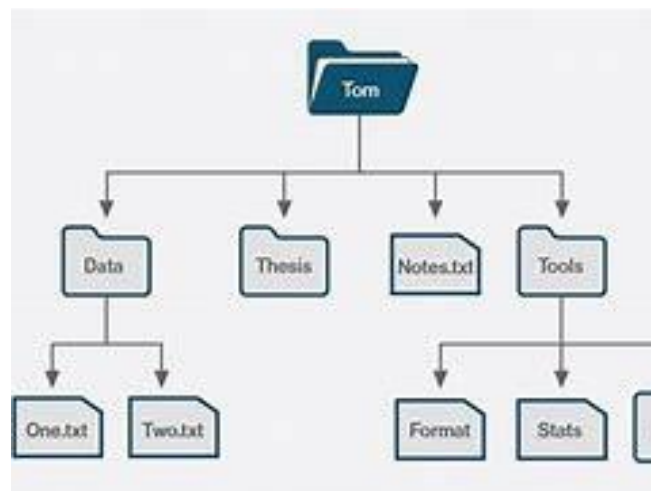
El archivo se abre en modo lectura ('r').

Se realiza la operación de lectura.

Cuando se sale del bloque **with**, el archivo se cierra automáticamente.

## 2.2.6 Organización de los archivos en el sistema de archivos

La organización de los archivos es dependiente del sistema operativo y de su estructura interna, de esta dependen los niveles de acceso, la seguridad y la manipulación. Sin embargo, existe la posibilidad de intercambiar entre sistemas, archivos de datos y comúnmente archivos de texto y dataset para el análisis de datos.



### 2.2.6.1 Tipos de Sistemas de Archivos (File System)

#### **FAT (File Allocation Table):**

Es uno de los sistemas de archivos más antiguos y simples.

Utilizado principalmente en sistemas operativos como DOS y versiones antiguas de Windows.

Utiliza una tabla de asignación de archivos (FAT) para almacenar la información sobre la ubicación de los archivos en el disco.

Limitaciones en el tamaño máximo de los archivos y el tamaño del volumen.

#### **NTFS (New Technology File System):**

Desarrollado por Microsoft como el sistema de archivos predeterminado para los sistemas operativos Windows a partir de Windows NT.

Ofrece características avanzadas como la seguridad de archivos y carpetas, la compresión de archivos, la encriptación y el control de acceso.

Es más robusto y seguro que FAT, con mejor gestión de espacio en disco y manejo de archivos grandes.

**ext3/ext4:**

ext3 es una versión anterior del sistema de archivos ext4, utilizada en sistemas Linux. ext4 es la evolución de ext3 y ofrece mejoras significativas en rendimiento y capacidad. Soporta características como tamaños de archivo y sistema de archivos más grandes, journaling para recuperación de datos después de fallos, y asignación diferida para mejorar el rendimiento.

**HFS+ (Hierarchical File System Plus):**

Desarrollado por Apple y utilizado en sistemas operativos Mac OS X hasta macOS 10.13 High Sierra.

Proporciona características como la journaling, soporte para archivos grandes y nombres de archivos Unicode.

Sin embargo, tiene limitaciones en la interoperabilidad con otros sistemas operativos y puede presentar problemas de fragmentación en discos duros de gran capacidad.

## 2.3. Operaciones Avanzadas con Archivos

Para la manipulación del file sistema en Python utilizamos módulos como os, shutil.

Se deben adaptar las rutas y nombres de archivos según la estructura de directorios y necesidades específicas. Además, ten en cuenta que las operaciones de mover, copiar y eliminar archivos pueden ser irreversibles, ¡así que ten cuidado al utilizarlas!

### 2.3.1 - Renombrar archivos

```
import os

# Ruta del archivo original
ruta_original = 'ruta/del/archivo/original.txt'

# Nuevo nombre del archivo
nuevo_nombre = 'nuevo_nombre.txt'

# Renombrar el archivo
os.rename(ruta_original, nuevo_nombre)
```

### 2.3.2 Mover archivos

```
import shutil

# Ruta del archivo original
ruta_original = 'ruta/del/archivo/original.txt'

# Ruta de destino
ruta_destino = 'ruta/de/destino/nuevo_directorio/original.txt'

# Mover el archivo
shutil.move(ruta_original, ruta_destino)
```

### 2.3.3 Copiar archivos

```
import shutil

# Ruta del archivo original
ruta_original = 'ruta/del/archivo/original.txt'

# Ruta de destino para la copia
ruta_destino = 'ruta/de/destino/copia/original_copia.txt'

# Copiar el archivo
shutil.copy(ruta_original, ruta_destino)
```

### 2.3.4 Eliminar archivos

```
import os

# Ruta del archivo a eliminar
ruta_archivo = 'ruta/del/archivo/a_eliminar.txt'

# Eliminar el archivo
os.remove(ruta_archivo)
```

## 2.4 Manejo de Directorios [Carpetas]

### 2.4.1 - Creación de directorios

```
# Ruta del nuevo directorio a crear
nuevo_directorio = 'ruta/del/nuevo/directorio'
# Crear el directorio
os.makedirs(nuevo_directorio)
```

### 2.4.2 - Eliminación de directorios

```
import shutil
# Ruta del directorio a eliminar
directorio_a_eliminar = 'ruta/del/directorio/a_eliminar'
shutil.rmtree(directorio_a_eliminar) # Eliminar el directorio y su contenido
```

### 2.4.3 -moverse a un directorio superior (padre)

```
import os
# Moverse al directorio superior
os.chdir('..')
```

### 2.4.4 -moverse a un directorio inferior (hijo)

```
import os
# Moverse a un directorio inferior
os.chdir('nombre_del_directorio')
```

### 2.4.5 -moverse a un directorio absoluto

```
import os
# Moverse a un directorio absoluto
os.chdir('/ruta/absoluta/del/directorio')
```

### 2.4.6 Listado de archivos en un directorio

```
import os
# Ruta del directorio a listar
directorio = 'ruta/del/directorio/a_listar'

# Obtener lista de archivos en el directorio
archivos = os.listdir(directorio)

# Imprimir la lista de archivos
print("Archivos en el directorio:")
for archivo in archivos:
    print(archivo)
```

### 2.4.7 Permisos de Archivos

Permiso de lectura para todos los usuarios

```
import os

# Ruta del archivo
archivo = 'ruta/del/archivo.txt'

# Establecer permisos de lectura para todos los usuarios
os.chmod(archivo, 0o444) # 0o444 indica permisos de solo lectura p
```

Permiso lectura y escritura para el propietario

```
import os

# Ruta del archivo
archivo = 'ruta/del/archivo.txt'

# Establecer permisos de escritura y lectura para el propietario del archivo
os.chmod(archivo, 0o600) # 0o600 indica permisos de lectura y escritura para el propietario
```

#### 2.4.8 Permiso de ejecución para todos los usuarios

```
import os

# Ruta del archivo
archivo = 'ruta/del/archivo.sh'

# Establecer permisos de ejecución para todos los usuarios
os.chmod(archivo, 0o755) # 0o755 indica permisos de ejecución para todos los usuarios
```

### 2.5 Archivos y ciencia de datos en la nube

Existen varias plataformas que presentan sus servicios de máquinas virtuales en la nube y empresas que proveen sus plataformas cada uno con su particularidad y características posiblemente diferentes. Algunas de ellas:

- 🔹 Google COLAB
- 🔹 Amazon SageMaker
- 🔹 IBM Watson
- 🔹 Azure Machine Learning
- 🔹 Anaconda
- 🔹 Cocalc
- 🔹 Kaggle
- 🔹 Paperspace

### 2.5.1 COLAB

COLAB es una herramienta de Google conocido como "Colaboratory", que permite ejecutar sus máquinas virtuales para ejecutar código Python. Es una herramienta muy utilizada en la comunidad de aprendizaje automático. Algunas ventajas:

- No requiere configuración
- Puede ejecutar comandos de consola
- Acceso a GPUs sin coste adicional
- Permite compartir contenido fácilmente
- Pued einteractuar con Google IA studio,
- Pueden crearse API con Gemini
- Puede importar grandes dataframe con pandas puede ejecutar sentencias BigQuery y llevar resultado a dataframe

BigQuery es un almacén de datos de Google Cloud de bajo coste, multinube, escalable, serverless y totalmente administrado. I

COLAB ES Muy útil para estudiantes, científicos de datos y trabajo con ia y otros trabajos como:

- Aprendizaje automático y Deep learning
- Análisis de datos
- Educación e investigación
- Simulación científica
- Desarrollo colaborativo
- Prototipo de código

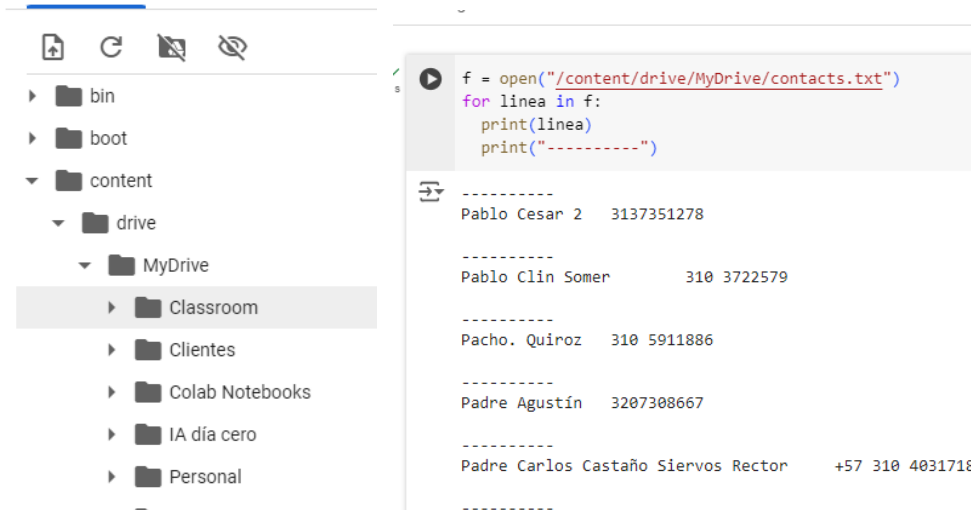
Para acceder a los archivos de DRIVE desde COLAB:

Debe activar el acceso a drive

Identificar el recurso (carpeta, archivo)

Tomar la ruta





```
f = open("/content/drive/MyDrive/contacts.txt")
for linea in f:
    print(linea)
    print("-----")
```

-----  
Pablo Cesar 2    3137351278  
  
-----  
Pablo Clin Somer            310 3722579  
  
-----  
Pacho. Quiroz    310 5911886  
  
-----  
Padre Agustín    3207308667  
  
-----  
Padre Carlos Castaño Siervos Rector    +57 310 4031718  
  
-----

En COLAB se puede acceder a otros archivos remotos como Github, u otro servidor.



## Referencias

1. [https://aulasoftwarelibre.github.io/taller-de-python/Python\\_Avanzado/Ficheros/](https://aulasoftwarelibre.github.io/taller-de-python/Python_Avanzado/Ficheros/)
2. <https://barcelonageeks.com/modulo-de-sistema-operativo-en-python-con-ejemplos/>
3. <https://pywombat.com/articles/shutil-python>
4. [https://www.w3schools.com/python/python\\_file\\_handling.asp](https://www.w3schools.com/python/python_file_handling.asp)
5. <https://www.delftstack.com/es/howto/python-pandas/paramiko-python/>
6. <https://community.cisco.com/t5/blogs-general/01-netmiko-la-herramienta-de-automatizaci%C3%B3n-para-dispositivos-de/ba-p/4954431>
7. <https://python.readthedocs.io/en/v2.7.2/library/shutil.html>
8. <https://www.youtube.com/watch?v=inN8seMm7UI>

Fecha Creación	Enero 25 2024
Responsable	Plinio Neira Vargas
Revisado por	Sonia Escobar
Fecha Revisión	Febrero 10 2024