Universidad de Guadalajara

Centro de Ciencias Exactas e ingenierías

Seminario de Solución de Estructuras de Datos II | D03
Ciclo: 2023-B

Profesora: Ávila Cárdenas Karla

Alumno: Azano Sepúlveda Cristian Michel - 215017473

Actividad 1: "Manejo de archivos"

Entrega: Domingo 27 de Agosto de 2023 | Guadalajara, Jalisco

Introducción

El "fichero", es un elemento clave en la organización, almacenamiento y manipulación de datos en diversos dispositivos y plataformas. Los ficheros, en su esencia, constituyen conjuntos de datos interrelacionados que se preservan de manera permanente en medios de almacenamiento como discos duros y memorias flash. Esta noción de permanencia contrasta con los datos temporales almacenados en la memoria RAM. El presente trabajo busca desentrañar la importancia y los fundamentos del concepto de fichero, analizando su utilidad en la programación, su rol en la estructura del Sistema Operativo y los diferentes tipos de ficheros existentes.

Concepto y Permanencia de los Ficheros

En el universo digital, los ficheros representan entidades fundamentales. Un fichero, en su definición más básica, engloba datos relacionados que se almacenan de manera perdurable en diversos dispositivos. Estos datos persisten incluso después de que los programas que los crearon han dejado de ejecutarse y, en muchos casos, sobreviven a la desconexión eléctrica. La distinción entre los ficheros y los datos volátiles en la memoria RAM, que desaparecen al finalizar la sesión, subraya su relevancia en la continuidad y recuperación de información.

Utilidad y Rol de los Ficheros

El concepto de fichero no solo es una abstracción del Sistema Operativo, sino que también es un pilar esencial en la programación moderna. Los ficheros proporcionan el soporte necesario para la existencia de aplicaciones en el disco duro y permiten a los desarrolladores crear interfaces de usuario eficaces. Los sistemas de ficheros, concebidos y gestionados por el Sistema Operativo, comprenden la estructura lógica que organiza y localiza los datos en el medio de almacenamiento. De este modo, los ficheros simplifican la interacción del usuario con las aplicaciones al permitir el almacenamiento y recuperación de datos entre sesiones de trabajo.

Tipos de Ficheros y su Estructura

La diversidad en el diseño y estructura de los ficheros refleja la flexibilidad que los desarrolladores poseen al trabajar con ellos. En función de cómo almacenan los datos, los ficheros se dividen en dos categorías principales: ficheros binarios y ficheros de texto. Los primeros contienen una representación exacta de los datos en forma binaria y, por lo general, no son editables. En contraste, los ficheros de texto almacenan datos en formato alfanumérico y pueden ser leídos y modificados utilizando editores de texto convencionales.

Ficheros de Texto

Este trabajo se enfocará en el análisis y comprensión de los ficheros de texto, que desempeñan un papel fundamental en la programación y manipulación de datos. El manejo de ficheros de texto se asemeja a la interacción tradicional de entrada/salida a través de la consola, facilitando la creación, lectura y modificación de datos. A lo largo de las próximas páginas, exploraremos en detalle cómo los ficheros de texto permiten el intercambio de información entre aplicaciones y cómo su estructura alfanumérica brinda versatilidad y facilidad de uso en diversos contextos.

En síntesis, este trabajo se adentrará en el universo de los ficheros, explorando su definición, utilidad y tipología, con un enfoque particular en los ficheros de texto y su relevancia en la ingeniería informática moderna. A medida que avanzamos en el análisis, desvelaremos la importancia de estos elementos en la programación y en la preservación eficiente de datos en sistemas tecnológicos contemporáneos.

Gestión de archivos en Java

Archivos, ¿qué son?

Antes de hablar sobre cómo gestionamos los archivos debemos de conocer qué son, para ello tenemos esta sencilla explicación: un archivo es un conjunto de datos estructurados guardados en algún medio de almacenamiento que pueden ser utilizados por aplicaciones.

Está compuesto por:

- **Nombre**: Identificación del archivo.
- **Extensión**: Indica el tipo de archivo.

Clase File

La clase **File** además de proporcionarnos información sobre los archivos y directorios nos permite crearlos y eliminarlos.

Para ello esta clase nos permite crearlos utilizando:

```
File nombreFile = new File("/carpeta/archivo");
```

y borrarlos con:

```
nombreFile.delete();
```

Además de los anteriores disponemos de estos métodos para gestionarlos:

Método	**Descripción**
createNewFile()	Crea (si se puede) el fichero indicado
delete()	Borra el fichero indicado
mkdirs()	Crea el directorio indicado
getName()	Devuelve un *String* con el nombre del fichero
getPath()	Devuelve un *String* con la ruta relativa
getAbsolutePath()	Devuelve un *String* con la ruta absoluta
getParent()	Devuelve un *String* con el directorio que tiene por encima
renameTo()	Renombra un fichero al nombre del fichero pasado como parámetro (se puede mover el fichero resultante a otra ruta, en caso de ser la misma se sobrescribirá)

exists()	*Boolean* que nos indicará si el fichero existe
canWrite()	*Boolean* que nos indicará si el fichero puede ser escrito
canRead()	*Boolean* que nos indicará si el fichero puede ser leído
isFile()	*Boolean* que indica si el fichero es un archivo
listFiles()	Método que devuelve un *array* con los ficheros contenidos en el directorio indicado
isDirectory()	*Boolean* que indica si el fichero es un directorio
lastModified()	Devuelve la última hora de modificación del archivo
length()	Devuelve la longitud del archivo

Ya que somos capaces de trabajar tanto archivos como directorios tenemos métodos que nos permiten diferenciarlos. Debajo de estas líneas tenemos un ejemplo para ello.

```
/* En este ejemplo asignamos la ruta del directorio utilizando
   la primera posición del array args[] del método main. En este segmento
   de código comprobamos si el array no está vacío para luego
   comprobar si el fichero es un directorio, en este caso
   guarda los nombres del contenido del directorio en un array
   para mostrarlos luego gracias al foreach */
if(args.length > 0) {
  File f = new File(args[0]);

   if(f.isDirectory()) {
     File[] ficheros = f.listFiles();

     System.out.println("Lista de los nombres de ficheros dentro del directorio");

     for(File file : ficheros)
       System.out.println("\t" + file.getName());
   }
}
```

A la hora de pasar el nombre de un archivo a través del array *args[]* podemos comprobar si éste existe o no, para que lo cree en caso contrario, y evitar posibles errores utilizando el código a continuación.

```
if(args.length > 0) {
  File fichero = new File(args[0]);

  if(!fichero.exists()) {
    try {
      fichero.createNewFile();
```

```
   } catch (IOException e) {
     e.printStackTrace();
   }
  }
}
```

Las demás clases de lectura y escritura de archivos que veremos a continuación son susceptibles de generar excepciones, por lo que debemos tratarlas siempre con *try/catch* y además tener la buena costumbre de utilizar el método *close()* para cerrarlos.

Un ejemplo por adelantado:

```
/* Este ejemplo de uso del try/catch escribe en un archivo txt,
   números aleatorios entre 1 y 10 */
try {
  FileWriter fl = new FileWriter("test.txt");
  PrintWriter pw = new PrintWriter(fl);

  for(int i = 0; i < 10; i++) {
    pw.println(i + 1);
    System.out.println(i + 1);
  }

  pw.close();
} catch(IOException e) {
  e.printStackTrace();
}
```

Clases FileWriter y PrintWriter

Ambas clases sirven para escribir caracteres en ficheros, simplemente **PrintWriter** es una mejora de la clase **FileWriter** (comparten el mismo padre **java.io.Writer**) que nos permite utilizar métodos adicionales.

```
/* Este ejemplo pregunta a un usuario unos datos para luego
   almacenarlos en el archivo ejercicio1.txt */
int numPersonas = 5;

int[] edad = new int[numPersonas];
String[] nombre = new String[numPersonas];
String[] apellido = new String[numPersonas];

try {
  /* Aunque no se utilice normalmente de esta forma aquí
     os pongo un ejemplo de uso para ambas clases */
  FileWriter fichero = new FileWriter("ejercicio1.txt");
  PrintWriter pw = new PrintWriter(fichero);
```

```
 for(int i = 0; i < numPersonas; i++) {
  System.out.print("\nDime el nombre del usuario " + (i + 1) + ": ");
  nombre[i] = Entrada.cadena();

  System.out.print("Dime el apellido del usuario " + (i + 1) + ": ");
  apellido[i] = Entrada.cadena();

  System.out.print("Dime la edad del usuario " + (i + 1) + ": ");
  edad[i] = Entrada.entero();

  pw.println("\nUsuario " + (i + 1) + "\tNombre: " + nombre[i] + "\tApellido: "
          + apellido[i] + "\tEdad: " + edad[i]);
 }

 pw.close();
} catch(IOException e) {
 e.printStackTrace();
}
```

De los métodos más útiles podemos destacar:

FileWritter	
write()	Escribe uno o varios caracteres
flush()	Limpia el flujo de datos
close()	Cierra **FileWriter** para terminar la gestión con el archivo (internamente llama al método *flush()*)
PrintWriter	
println()	Escribe en el archivo el parámetro que le introduzcamos
append()	Añade el carácter (*char*) o caracteres (CharSequence) específicos
checkError()	*Boolean* que llama al método *flush()* y comprueba si hay algún error
close()	Cierra **PrintWriter** para terminar la gestión con el archivo

Clases FileReader y BufferedReader

Es una buena práctica utilizar estas clases conjuntamente pues **FileReader** simplemente lee caracteres de un fichero y **BufferedReader** nos ayuda a guardarlos en un *buffer* para tratarlos de una forma más segura.

```
/* Este fragmento de código lee un archivo
   y lo muestra por la terminal */
File f = new File("ejercicio1.txt");

try {
 FileReader fr = new FileReader(f);
 BufferedReader br = new BufferedReader(fr);
```

```
 String linea = br.readLine();

 System.out.println();

 while(linea != null) {
  System.out.println(linea);
  linea = br.readLine();
 }

 br.close();
} catch(IOException e) {
 e.printStackTrace();
}
```

Los métodos con los que comúnmente trabajaremos son:

FileReader	
mark()	Marca la posición actual
read()	Lee uno o varios caracteres
reset()	Reinicia la lectura
toString()	Devuelve en forma de *String* el contenido del objeto
BufferedReader	
readLine()	Lee una línea de texto
skip()	Se salta “n” caracteres
close()	Cierra la lectura del archivo

Comparar líneas

Si queremos comparar el texto que contiene un archivo con un texto que introduzcamos nosotros, a través de un *String* por ejemplo, hemos de ser conscientes que existe la posibilidad de que las mayúsculas no nos coincidan o incluso que se nos colara algún espacio al final o al principio del *String*, para ello podemos utilizar los métodos *toLowerCase()* para pasar todo el texto a minúscula o *toUpperCase()* para mayúscula y también utilizar el método *trim()* para quitar los espacios.

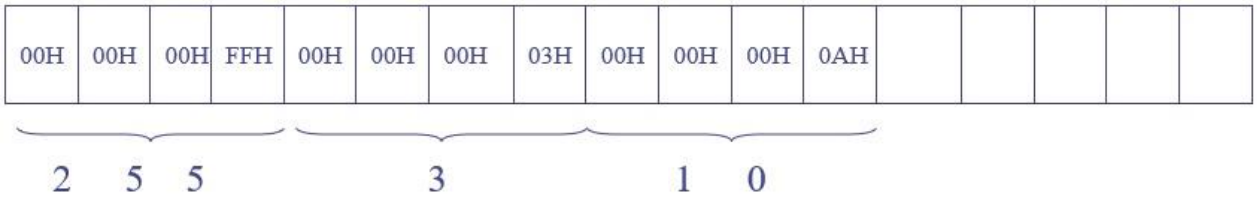
```
String lineaAComparar = "hola mundo";

if(lineaArchivo.toLowerCase().trim().equals(lineaAComparar.toLowerCase().trim()))
  // Las líneas son iguales
```

Tipos de ficheros

El **contenido** y la **estructura** de un fichero responde a un criterio de **diseño libre**, elegido por el desarrollador de una aplicación. En cualquier caso, con relación a la forma en que los datos son almacenados, los ficheros podríamos clasificarlos como:

- **Ficheros binarios**: Contienen una representación exacta del contenido (binario, ceros y unos) de los datos. No son *editables*.

Ejemplo: Representación de 3 números enteros 255 3 10 en complemento a 2. Cada entero ocupa el mismo número de bytes.



- **Ficheros de texto**: Los datos están representados con los caracteres alfanuméricos que los representan. Pueden ser leídos y modificados a través de un editor de texto.

Ejemplo: Representación de 3 números enteros 255 3 10 codificados en ASCII, separados por espacios en blanco (20H en ASCII). Cada entero ocupa un número de bytes distinto.

En este cuaderno trabajaremos con ficheros de texto. Vamos a ver que el trabajo con los ficheros de texto es esencialmente análogo al trabajo con entrada/salida convencional a través de la consola (habitualmente el teclado y la pantalla).

Elementos básicos de programación con ficheros

La **equivalencia** entre entrada/salida a través de teclado y pantalla y la utilización de ficheros es muy profunda. Los S.O. actuales hacen un tratamiento unificado de estos recursos y tratan, por ejemplo, a la pantalla y al teclado como ficheros de salida y de entrada respectivamente, *ficheros* que están siempre *listos* para ser utilizados. Es una muestra más del mecanismo de abstracción mencionado más arriba.

Recalquemos que cuando en Python usamos print(), estamos escribiendo datos en el fichero *por defecto* o *estándar*, la pantalla, y que cuando empleamos input(), estamos leyendo datos del fichero *por defecto* o *estándar*, el teclado.

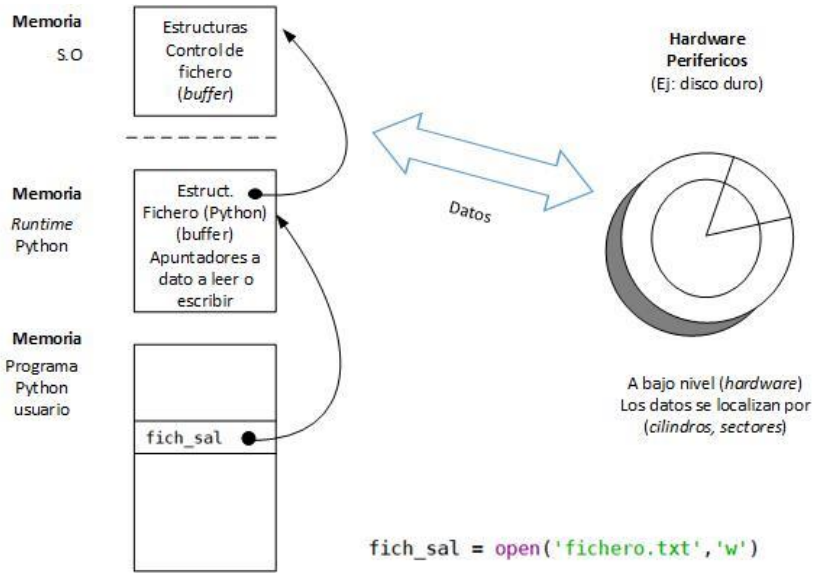
Cuando no usamos los *ficheros estándar*, tanto en Python como en cualquier otro lenguaje de programación, debemos realizar algunas tareas *adicionales*:

- **Abrir el fichero**: hay que asociar el fichero (definido a nivel del S.O.) con un *objeto* que provea la fuente de datos y definir si se va utilizar para entrada o para salida de datos, es decir, para leer o para escribir.
- **Cerrar el fichero**: Una vez finalizada la interacción con el *objeto* que representa el fichero, este hecho debe ser informado al S.O. mediante los métodos apropiados. Así, el S.O. podrá realizar las acciones requeridas para garantizar que el fichero queda en un estado consistente y seguro.

El concepto de flujo de datos (stream)

Las operaciones de lectura/escritura hacen uso del concepto de **flujo** o **corriente** de datos (**stream**). El símil se basa en el hecho de interpretar la entrada/salida como una corriente o río de datos, donde estos son representados por la aparición **en serie**, uno tras otro, de los **bytes** que representan cada uno de los valores transferidos.

El concepto de transferencia de datos en serie es clave. Si desde el teclado queremos introducir el número 543, es evidente que antes de escribir el 3, debemos escribir el 4 y antes el 5. En el caso de los ficheros se utiliza el mismo paradigma, se escribe un dato siempre a continuación del anterior.



La figura previa describe la relación que existe entre los datos almacenados en el medio físico y la *construcción lógica* que es el fichero. A nivel del dispositivo periférico, la forma de referirse al dato es a través de atributos de muy *bajo nivel*, tales como su *dirección*, que puede ser descrita, por ejemplo, por la intersección de *cilindros* y *sectores* definidos por el *hardware*.

El **fichero**, por su parte, es un conjunto **lógico** de datos donde el programa que lo crea (con la ayuda indispensable del Sistema Operativo) *decide* tratarlos como una unidad. Nótese que, en el nivel hardware, si se usa un disco duro como almacenamiento, ese conjunto lógico involucrará a múltiples *cilindros* y *sectores*.

Memoria intermedia (buffers)

La entrada/salida desde/hacia **ficheros** está mediada, de forma transparente al programador, por memoria auxiliar o búferes (**buffers**).

Los **búferes** cumplen distintos cometidos, pero el fundamental es el de servir como **pulmón** de la CPU. Dado que los procesos de lectura/escritura en los dispositivos son mucho más lentos que los realizados en la RAM, los datos son temporalmente leídos/escritos en búferes. De esta forma, los accesos a un dispositivo lento no ralentizan las aplicaciones.

Necesidad de cerrar el fichero

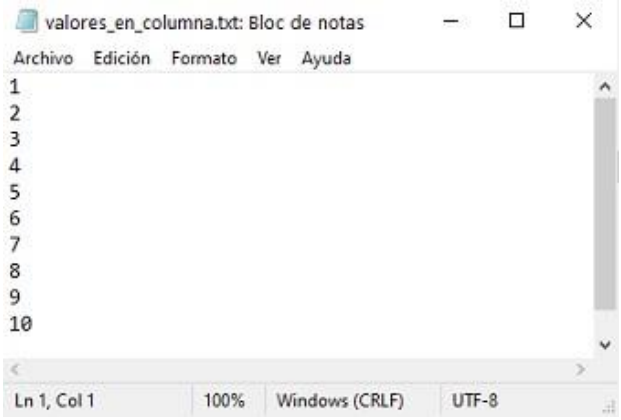
Cuando nuestro programa ha terminado de utilizar el **fichero** debe **cerrarlo**. ¿Por qué es necesario cerrar el fichero?

- La escritura/lectura se realiza sobre los *búferes* y estos son eventualmente transferidos al medio físico. Cuando se *cierra* el fichero, se **fuerza** a realizar esa transferencia. De otro modo, se podría incurrir en pérdida de información.
- La utilización de un fichero comporta un consumo importante de memoria del ordenador (*búferes* y otros elementos). Si no se cierra el fichero, esa memoria resulta *inutilizada*.
- El Sistema Operativo establece un número máximo de ficheros que pueden estar *abiertos* simultáneamente. Si no *cerramos* los ficheros que, de momento, no estamos utilizando, puede que nuestro programa no pueda *abrir* otros que sí son necesarios.

Lectura línea por línea de ficheros en Python

Para entender el manejo de ficheros en Python utilizaremos inicialmente un sencillo ejemplo. Para ello, utilizaremos un editor de textos con el que hemos creado un archivo llamado *valores_en_columna.txt*. El contenido es una secuencia

de valores enteros fácil de recordar, para que las comprobaciones del buen funcionamiento del código sean sencillas de verificar.



De momento, para simplificar la **ruta de acceso** al fichero, este archivo está situado al mismo nivel que este cuaderno. La extensión .txt es la habitual para *recalcar* que el fichero es de tipo texto, pero podemos usar cualquier extensión, preferentemente no utilizando aquellas habituales en aplicaciones estándar, tales como .doc, .exe, .lib, .pdf, etc.

La apertura y el cierre

Los ficheros no estándar deben ser abiertos antes de ser utilizados, y cerrados cuando se concluya (al menos provisionalmente) el trabajo con ellos.

Para abrir un fichero debemos tener en cuenta:

- La localización del fichero: (Ej.: "*datos/temperaturas/Valladolid.dat*")
- La declaración del **modo de apertura**, que es un parámetro que indica si, por ejemplo, queremos leer del fichero o escribir en el fichero:
  - 'w' para escritura,
  - 'r' para lectura
- La asignación de un nombre en el programa que a partir de ese momento representará al fichero (Ej.: fich_sal)

Para abrir un fichero disponemos de la función open(), que nos devuelve el **objeto** fichero con el que vamos a poder trabajar a partir de ese momento. En el ejemplo, se abre un fichero 'Valladolid.dat' especificando la ruta de acceso desde el **directorio de trabajo**, con la intención de escribir en él datos, ('w') y al que se referenciará con el nombre fich_sal.

```
fich_sal = open('datos/temperaturas/Valladolid.dat','w')
```

En la siguiente tabla se muestran los diferentes **modos de apertura**. Con saber utilizar los modos 'w' y 'r' es más que suficiente para los objetivos del curso.

Modo de apertura	Descripción	Acción
'w'	Escritura	Si el fichero no existe lo crea. Si existe, borra su contenido
'r'	Lectura	Si existe fichero: lo abre. Si no existe: excepción FileNotFoundError
'a'	Añadir	Si fichero no existe, lo crea para escritura. Si existe, añade al final
'w+'	Actualizar	Escritura/ lectura. Si el fichero no existe lo crea. Si existe: borra
'r+'	Actualizar	Lectura/Escritura. Si no existe: excepción FileNotFoundError
'a+'	Añadir	Escritura/lectura. Si existe, añade al final.
'b'	Binario	Abre en binario. Combinadas con otras banderas: establece modo
'x'	Creación	Abre exclusivamente para crear fichero. Si ya existe, falla

Para cerrar el fichero se usa el método close(). Siguiendo con el ejemplo anterior:

```
fich_sal = open('datos/temperaturas/Valladolid.dat', 'w')
# Código de escritura en el fichero
# ...
fich_sal.close()  # Cerramos el fichero
```

En algunas celdas que siguen a continuación vamos a utilizar el **comando mágico** de **IPython** %reset -f para resetear las variables del espacio de nombres y que los resultados de una celda no influyan en las otras.

Lectura línea por línea

Tras abrir el fichero correspondiente, la forma básica de leer un fichero línea por línea es:

```
fich_ent = open('nombre_fichero.txt', 'r')
for linea in fich_ent:
    # Procesar la línea
fich_ent.close()
```

La variable linea es una **cadena de caracteres** que va tomando secuencialmente las cadenas de caracteres correspondientes a cada una de las líneas del fichero, desde la primera a la última.

```
# Leyendo del fichero "valores_en_columna.txt" línea a línea
fich_ent = open('valores_en_columna.txt', 'r')  # Apertura

for linea in fich_ent:
    print(linea)

fich_ent.close()  # Cierre
```

```
1

2

3

4

5

6

7

8

9

10
```

Al ejecutar el código observaréis que aparece una línea en blanco entre cada uno de los números. Esto es así porque cada línea en el fichero de texto tiene un carácter no imprimible nueva línea, el carácter \n. A eso se une el que por defecto introduce la función print().

Este hecho debe recordarnos que, al igual que con la función de lectura estándar input(), lo que estamos leyendo son cadenas de caracteres, es decir, linea es un dato tipo str. Si el usuario sabe que cada línea corresponde a un valor entero, podemos recurrir a la función int().

En el siguiente ejemplo, rehacemos el código para almacenar los valores enteros en una lista.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  línea a línea
fich_ent = open('valores_en_columna.txt', 'r')  # Apertura

lista_enteros = []
for linea in fich_ent:
    lista_enteros.append(int(linea))

print(lista_enteros)

fich_ent.close()  # Cierre
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Es muy importante darnos cuenta de que el éxito en la lectura del fichero se sustenta en el hecho de que conocemos de antemano su **formato**. En este caso, que el fichero está formado por líneas en las cuales hay un único número entero. Si ese formato no se cumple, tendremos problemas.

Veamos qué ocurre si pretendemos utilizar el mismo código con el fichero *valores_en_fila.txt*.


```
%reset -f
# Leyendo del fichero "valores_en_fila.txt" línea a línea
fich_ent = open('valores_en_fila.txt', 'r')  # Apertura

lista_enteros = []
for linea in fich_ent:
    lista_enteros.append(int(linea))

print(lista_enteros)
```

```
fich_ent.close()  # Cierre
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_8452\3874021872.py in <module>
      5 lista_enteros = []
      6 for linea in fich_ent:
----> 7     lista_enteros.append(int(linea))
      8
      9 print(lista_enteros)

ValueError: invalid literal for int() with base 10: '1 2 3 4 5 6 7 8 9
10\n'
```

Afortunadamente, tenemos una *bonita* excepción:

```
ValueError: invalid literal for int() with base 10: '1 2 3 4 5 6 7 8 9
10\n'
```

El análisis del mensaje nos da luz acerca del problema. La primera línea del fichero es la cadena de caracteres '1 2 3 4 5 6 7 8 9 10\n', que Python lógicamente es incapaz de convertir a un entero vía la función int(). Observad la presencia del carácter nueva línea \n al final.

Decimos afortunadamente porque no hay nada mejor que, de forma automática, el motor de ejecución de Python nos informe de que algo estamos haciendo mal. Imaginad que la ejecución no diese error y que en una importante variable de nuestra aplicación cargásemos un valor espurio.

Veremos más adelante cómo leer este fichero.

Manejo de excepciones con ficheros

Aviso

Este apartado requiere haber estudiado el tema **Manejo de excepciones**.

En el caso del uso de **ficheros**, la capacidad de responder de forma consistente a errores que se puedan producir es muy importante. Hay que tener en cuenta que, cuando se trata de ficheros, existen una serie de elementos *externos* que dependen del *hardware* y del **Sistema Operativo** que pueden fallar y que están fuera del control del programador.

Por ejemplo:

- El fichero que se pretende *abrir* para *lectura* no existe.
- Se pretende *abrir* para *escritura* un fichero en un medio físico (dispositivo) o lógico (carpeta) *protegido* contra escritura.

Otros fallos pueden ser debidos a:

- La estructura o formato que se espera del fichero no es la realmente existente.

Acabamos de ver el ejemplo más arriba con el fichero *valores_en_fila.txt*.

- Durante la lectura del fichero, se produce otro tipo de excepción, como una división por 0, etc.

Vamos a generar *artificialmente* una excepción IndexError y a capturarla debidamente. Lo lograremos creando de inicio una lista de 9 elementos, cuando nuestro fichero tiene realmente 10.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  línea a línea
try:
    fich_ent = open('valores_en_columna.txt', 'r')
    # Línea "artificial" para generar un error de índice en nuestro
ejemplo
    # Cambiando 9 por 10 el programa no genera error
    num_valores = 9
    lista_enteros = [0]*num_valores
    for i, linea in enumerate(fich_ent):  # Nótese el usu de
enumerate()
        lista_enteros[i] = int(linea)
except IndexError as error:
    print(error)
else:
    print(lista_enteros)
finally:
    print('Cerramos el fichero.')
    fich_ent.close()  # Cierre
```

```
list assignment index out of range
Cerramos el fichero.
```

Aquí vemos entrar en acción al bloque finally, un bloque concebido para ser destinado a cerrar recursos abiertos, una especie de *coche escoba*. Tanto si existe excepción como si no, el fichero está abierto y debemos cerrarlo.

Vamos ahora a provocar otro error típico: nos equivocamos al transcribir el nombre. En el siguiente ejemplo, hemos empleado colunna en lugar de columna.

```
%reset -f
try:
```

```
    fich_ent = open('valores_en_colunna.txt', 'r')
    # Línea "artificial" para generar un error de índice en nuestro
ejemplo
    # Cambiando 9 por 10 el programa no genera error
    num_valores = 9
    lista_enteros = [0]*num_valores
    for i, linea in enumerate(fich_ent):
        lista_enteros[i] = int(linea)
except IndexError as error:
    print(error)
else:
    print(lista_enteros)
finally:
    print('Cerramos el fichero.')
    fich_ent.close()  # Cierre
```

```
Cerramos el fichero.
---------------------------------------------------------------------------
-----
FileNotFoundError                         Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_8452\2102069716.py in <module>
      2 try:
----> 3     fich_ent = open('valores_en_colunna.txt', 'r')
      4     # Línea "artificial" para generar un error de índice en
nuestro ejemplo

FileNotFoundError: [Errno 2] No such file or directory:
'valores_en_colunna.txt'

During handling of the above exception, another exception occurred:

NameError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_8452\2102069716.py in <module>
     14 finally:
     15     print('Cerramos el fichero.')
---> 16     fich_ent.close()  # Cierre

NameError: name 'fich_ent' is not defined
```

¡Tenemos dos excepciones!

- La excepción FileNotFoundError es bastante descriptiva del tipo de error cometido.
- La excepción NameError nos está diciendo que no conoce la existencia de la variable fich_ent. Pero, ¡la tenemos definida en la línea fich_ent = open('valores_en_colunna.txt','r')! Raro ¿no?

Vamos, de momento, a manejar la excepción FileNotFoundError, cuyo origen tenemos claro.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  línea a línea
try:
    fich_ent = open('valores_en_colunna.txt', 'r')
    # Línea "artificial" para generar un error de índice en nuestro
ejemplo
    # Cambiando 9 por 10 el programa no genera error
    num_valores = 9
    lista_enteros = [0]*num_valores
    for i, linea in enumerate(fich_ent):
        lista_enteros[i] = int(linea)
except (IndexError, FileNotFoundError) as error:
    print(error)
else:
    print(lista_enteros)
finally:
    print('Cerramos el fichero.')
    fich_ent.close()  # Cierre
```

```
[Errno 2] No such file or directory: 'valores_en_colunna.txt'
Cerramos el fichero.
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
~\AppData\Local\Temp\ipykernel_8452\1302000543.py in <module>
     15 finally:
     16     print('Cerramos el fichero.')
---> 17     fich_ent.close()  # Cierre

NameError: name 'fich_ent' is not defined
```

Seguimos teniendo la excepción NameError.

¿Qué está pasando?

El problema es que la excepción FileNotFoundError se genera en la parte derecha de la asignación a la variable fich_ent, al intentar abrir un fichero que no existe.
Y **la asignación nunca llega a producirse** y, por tanto, la variable fich_ent no llega a estar definida nunca.

Nuestro finally tal y como está programado no nos ayuda. Podríamos solventar el problema a través de la utilización de código más verboso, pero hay mejores opciones.

El administrador de contextos with

A partir de la versión 2.6 de Python se introdujo una *nueva estructura* de control de flujo, la construcción with. La estructura with ha sido *diseñada* específicamente para lidiar con código donde se manejan **objetos** que utilizan **recursos** externos. Por ello, Python define a la estructura with como un **administrador de contextos** (**context manager**).

El concepto de **contexto** se utiliza en informática para referirse al conjunto de datos utilizados por un **recurso** que deben ser guardados para permitir una posterior reutilización.

Además de los ficheros, un **administrador de contextos** como with puede trabajar con otros objetos, tales como aquellos dedicados a gestionar conexiones a red, bases de datos, etc. Con todos estos recursos, se van produciendo una serie de pasos, generándose nuevos *estados*. Los recursos son *adquiridos* y deben ser *liberados* o *cerrados* aún en presencia de **excepciones**.

La construcción with crea un *contexto* que ante la presencia de posibles excepciones maneja el *recurso* que representa el fichero. Usando with ya no es necesario *cerrar explícitamente* el fichero utilizando close(): el **administrador de contexto** creado con with se ocupa de todos estos detalles *tras las bambalinas*.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  línea a línea
with open('valores_en_columna.txt', 'r') as fich_ent:
    lista_enteros = []
    for linea in fich_ent:
        lista_enteros.append(int(linea))
print(lista_enteros)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

El manejo de excepciones usando with

Aviso

Este apartado requiere haber estudiado el tema **Manejo de excepciones**.

El uso de with resuelve los problemas del ejemplo visto más arriba.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  línea a línea
try:
    with open('valores_en_columna.txt', 'r') as fich_ent:
        # Línea "artificial" para generar un error de índice en
nuestro ejemplo
        # Cambiando 9 por 10 el programa no genera error
```

```
        lista_enteros = [0]*10  # Línea "artificial" que genera un
error de índice en este ejemplo
        for i, linea in enumerate(fich_ent):
            lista_enteros[i] = int(linea)
except (FileNotFoundError, IndexError) as error:
    print(error)
else:
    print(lista_enteros)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

En la celda anterior puede probar ahora a cambiar el nombre del fichero o el número de elementos de la lista para ver cómo se manejan las excepciones con with.

Una vez analizadas las excepciones, la forma recomendada de leer el fichero `'valores_en_columna.txt'` con `with` sería:

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  línea a línea
try:
    with open('valores_en_columna.txt', 'r') as fich_ent:
        lista_enteros = []
        for linea in fich_ent:
            lista_enteros.append(int(linea))
except FileNotFoundError as error:
    print(error)
else:
    print(lista_enteros)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Vistas estas ventajas, usaremos a partir de ahora, la estructura `with`.

Otras formas de lectura en Python

Una alternativa a leer línea por línea es hacerlo en **un único paso**:

- usando el método readlines(). Se crea una lista con las cadenas de caracteres de cada una de las líneas.
- usando el método read(). Obtendremos una única cadena de caracteres unión de todas las líneas, incluidos los caracteres nueva línea.

Debemos señalar que estas funciones pueden usarse con un argumento indicando el número de caracteres a leer, opción que no estudiaremos.

Para hacer menos prolija la explicación, evitaremos el uso del manejo de excepciones en lo que sigue.

Ejemplo de uso de readlines()

Para el fichero que estamos manejando, podemos observar que el código es menos compacto que con el método de lectura iterada línea por línea.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  con readlines()

with open('valores_en_columna.txt', 'r') as fich_ent:
    lista_lineas = fich_ent.readlines()

print("Lista con las líneas del fichero\n{}".format(lista_lineas))
# Transformamos cada una de las líneas en el entero correspondiente
lista_enteros = [int(linea) for linea in lista_lineas]  # Lista por
comprensión
print("Lista con los enteros del fichero\n{}".format(lista_enteros))
```

```
Lista con las líneas del fichero
['1\n', '2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n', '9\n',
'10\n']
Lista con los enteros del fichero
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ejemplo de uso de `read()`

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  con read()
with open('valores_en_columna.txt', 'r') as fich_ent:
    lineas_unidas = fich_ent.read()

print("Una única cadena de caracteres correspondiente a todo el
fichero\n{}".format(lineas_unidas))
```

```
Una única cadena de caracteres correspondiente a todo el fichero
1
2
3
4
5
6
7
8
9
10
```

¡Atención! Lo que hemos sacado por pantalla es una única cadena de caracteres, la cadena '1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n', en cuyo interior hay caracteres nueva línea \n.

Si utilizamos esta alternativa, para extraer nuestra secuencia de números enteros podemos recurrir al método split() asociado a las cadenas de caracteres. El método split() extrae las *palabras* de la cadena de caracteres situadas entre espacios en blanco, que incluyen tabuladores, nuevas líneas, etc…

El método `split()` ya lo hemos usado cuando introducíamos por teclado valores *secuencialmente separados por espacios sin usar intro*.

```
%reset -f
# Leyendo del fichero "valores_en_columna.txt"  con read()
with open('valores_en_columna.txt', 'r') as fich_ent:
    lineas_unidas = fich_ent.read()

lista_enteros = [int(palabra) for palabra in lineas_unidas.split()]
print("Lista con los enteros del fichero en
columna\n{}".format(lista_enteros))
```

```
Lista con los enteros del fichero en columna
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Con este método, estamos en condiciones de procesar también nuestro archivo de números en una sola línea *valores_en_fila.txt*.

```
%reset -f
# Leyendo del fichero "valores_en_fila.txt"  con read()
with open('valores_en_fila.txt', 'r') as fich_ent:
    lineas_unidas = fich_ent.read()

lista_enteros = [int(palabra) for palabra in lineas_unidas.split()]
print("Lista con los enteros del fichero en
fila\n{}".format(lista_enteros))
```

```
Lista con los enteros del fichero en fila
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Debemos recalcar que existen más posibilidades de lectura de los ficheros, que pueden consultarse en la documentación de Python, pero, entre todas las opciones, la lectura línea por línea es probablemente la más fácil y más utilizada.

Leyendo un fichero con formato complejo

Los dos ejemplos de ficheros vistos hasta el momento tienen una estructura muy simple, lo que no significa que no sean útiles.

Las aplicaciones prácticas requieren ser capaces de **crear** y **leer** ficheros que tengan una *estructura compleja* conocida de tamaño arbitrario, con independencia

del número de datos que estos ficheros almacenen. En estos ficheros pueden mezclarse comentarios y datos tanto numéricos como cadenas de caracteres.

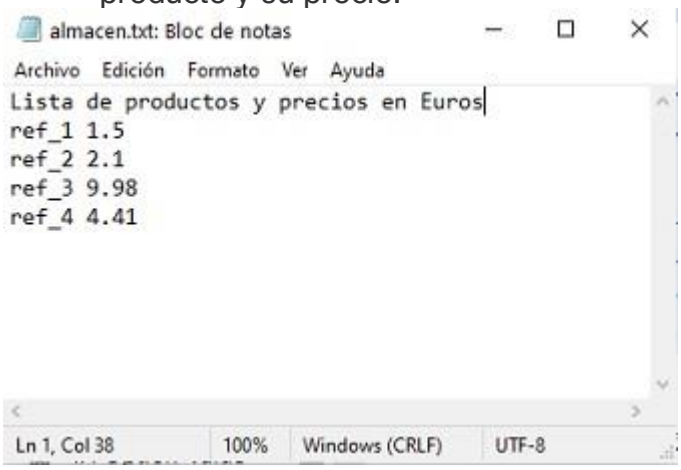
Para dar solo algunos ejemplos de los innumerables que podrían citarse, un fichero puede contener:

- los datos del censo de una ciudad o un país: nombre y apellidos, DNI, dirección y edad de los votantes.
- la sucesión de temperaturas recogidas por un sensor en determinado lugar.
- las filas y columnas de una matriz de dimensiones arbitrarias.
- los datos, organizados en número arbitrario de ejes, de un estudio epidemiológico, etc.

Todos los problemas tienen en común que el **significado de los datos** y **la estructura básica** de la organización de los mismos **tienen que ser conocidos por el programador** que diseña la aplicación de lectura y/o escritura del fichero. El *volumen* total de los datos (cantidad de personas, número de lecturas del sensor, dimensiones de la matriz, numero de ejes y cantidad de elementos por eje) es, en general, desconocido.

La figura siguiente muestra el fichero *almacen.txt*, cuyo formato es algo más complejo, formado por:

- Una línea inicial que es un comentario explicativo del contenido del fichero, en general, útil solo para el usuario que lo abre directamente desde un editor de textos.
- Una serie de líneas que contienen, cada una de ellas, una referencia de un producto y su precio.



Veamos un posible fragmento para su lectura. Lo relevante:

- Leemos la primera línea y la obviamos
- Para cada una de las siguientes líneas, extraemos con split() la referencia y el precio.

Es importante darnos cuenta también aquí del concepto de **flujo de datos**. Tras leer con readline() la línea del comentario, ésta ya está extraída de él. De hecho, un **puntero** oculto estará ahora *señalando* a la siguiente línea, al siguiente **byte** a extraer, correspondiente a la línea donde se encuentra ref_1 y 1.5. Por eso, el bucle for que itera a continuación sobre el objeto fichero fich lo hace desde esa línea, no desde el principio del fichero.

```
# Leyendo el fichero "almacen.txt"
with open('almacen.txt', 'r') as fich:
    fich.readline()  # Leemos la primera línea, pero la obviamos
    producto = []
    precio = []
    for linea in fich:
        palabras = linea.split()
        producto.append(palabras[0])
        precio.append(float(palabras[1]))

print(producto)
print(precio)
```

```
['ref_1', 'ref_2', 'ref_3', 'ref_4']
[1.5, 2.1, 9.98, 4.41]
```

Vamos a introducir en el siguiente fragmento dos mejoras:

- Leemos el fichero a través de una función, encapsulando su código
- Creamos una lista de tuplas con la pareja (referencia, precio)

```
%reset -f
def lee_fichero(nombre):
    with open(nombre, 'r') as fich:
        fich.readline()  # Leemos la primera línea pero la obviamos
        listado_precios = []
        for linea in fich:
            palabras = linea.split()
            listado_precios.append((palabras[0], float(palabras[1])))

    return listado_precios


# Programa principal
lista = lee_fichero('almacen.txt')  # Prueba a introducir un nombre
erróneo
print(lista)
```

```
[('ref_1', 1.5), ('ref_2', 2.1), ('ref_3', 9.98), ('ref_4', 4.41)]
```

Escritura en un fichero

La escritura en un fichero es comparativamente algo más simple que la lectura. No en vano, es el programador el que controla cómo es la estructura del fichero. Además, puede *despreocuparse* de elementos tales como los caracteres nueva línea, espacios en blanco, etc.

Aunque puede usarse la función print() para escribir en un fichero, el método relevante es write(). A diferencia de print(), el método write() no añade un carácter nueva línea por defecto al escribir en el fichero.

Si necesitamos añadir una línea, no tenemos más que añadir a la cadena correspondiente el carácter \n.

Otro aspecto importante es ser conscientes de lo que ocurre si abrimos un fichero para escritura usando el modo de apertura 'w':

- Si el fichero no existe, se creará uno nuevo con ese nombre
- Si el fichero ya existe, **¡se borrará su contenido!**

En el siguiente fragmento, vamos a crear un fichero *almacen_clon.txt*, idéntico a *almacen.txt*, usando todas las herramientas que ya conocemos.

```
%reset -f
def escribe_fichero(nombre, comentario, lista):
    with open(nombre, 'w') as fich:
        fich.write(comentario + '\n')  # Escribimos el comentario
        for x in lista:
            fich.write('{} {}\n'.format(x[0], x[1]))



# Programa principal
comentario = 'Lista de productos y precios en Euros'
listado_precios = [('ref_1', 1.5), ('ref_2', 2.1), ('ref_3', 9.98),
('ref_4', 4.41)]
escribe_fichero('almacen_clon.txt', comentario, listado_precios)
```

En lugar de utilizar el método write() se podría haber utilizado la propia función print() que hemos venido utilizando para la salida por la consola (asociada normalmente a la pantalla).

Para que print() escriba en un fichero diferente a la consola, que es el *fichero por defecto*, se puede utilizar el parámetro file de print() de la manera que se muestra:

```
print(cadena, file=fich)
```

Como ya sabemos, la función print() sí incluye automáticamente el cambio de línea al final de la cadena procesada.

La versión de la función escribe_fichero() usando print() sería la siguiente:

```
%reset -f
def escribe_fichero(nombre, comentario, lista):
    with open(nombre, 'w') as fich:
        print(comentario, file=fich)  # Escribimos el comentario
        for x in lista:
            print(x[0], x[1], file=fich)
```

Manejo de archivos en JavaScript

El manejo de archivos incluye diferentes operaciones como **la creación**, **lectura**, **actualización**, **renombrando** y **eliminando**. Tenemos que acceder a los archivos del sistema, lo que no nos es posible escribir desde cero. Entonces, NodeJS proporciona un módulo llamado **fs (sistema de archivos)** para el manejo de archivos.

Veamos diferentes métodos del **fs** módulo.

fs.open()

El método fs.open() tomará dos argumentos **camino** e **modo**.

La **camino** se utiliza para localizar el archivo.

El argumento **modo** se utiliza para abrir el archivo en diferentes modos como **adjuntar, escribir,** e **leyendo.**

Si abre cualquier archivo en un modo específico, entonces puede realizar solo un tipo de operación correspondiente al modo que ha pasado al método. Veamos la lista de modos y operaciones correspondientes.

Moda	**Operación**
'r'	Abre un archivo en modo lectura
'un'	Abre un archivo en modo adjunto
'w'	Abre un archivo en modo de escritura
'a +'	Abre un archivo en modo de adición y lectura
'w +'	Abre un archivo en modo de escritura y lectura
'r +'	Abre un archivo en modo lectura y escritura

Si el archivo no existe en la ruta dada, creará un nuevo archivo vacío. Veamos el código para abrir un archivo en diferentes modos.

```
const fs = require("fs");

fs.open("sample.txt", "w", (err, file) => {
  if (err) throw err;
  console.log(file);
});
```

El método fs.open() arrojará un error si el archivo no existe al abrir en **lectura** modo. Creará un nuevo archivo vacío en **la escritura** e **adjuntando** Modos.

Podemos realizar diferentes operaciones sobre el archivo abierto. Escribiremos un programa completo al final de este tutorial después de aprender algunos métodos más esenciales del **fs** módulo.

fs.appendFile()

El método fs.appendFile() se utiliza para agregar el contenido al final del archivo. Si el archivo no existe en la ruta dada, creará uno nuevo. Agregue algo de contenido al archivo usando el siguiente código.

```
const fs = require("fs");

fs.appendFile("sample.txt", "Appending content", (err) => {
  if (err) throw err;
  console.log("Completed!");
});
```

fs.writeFile()

El método fs.writeFile() se utiliza para escribir el contenido en el archivo. Si el archivo no existe en la ruta dada, creará uno nuevo. Pruebe el siguiente código para escribir el contenido en un archivo.

```
const fs = require("fs");

fs.writeFile("sample.txt", "Writing content", (err) => {
  if (err) throw err;
  console.log("Completed!");
});
```

fs.readFile()

El método fs.readFile() se utiliza para leer el contenido de un archivo. Lanzará un error si el archivo no existe en la ruta dada. Examine el siguiente código para el método.

```
const fs = require("fs");

fs.readFile("sample.txt", (err, data) => {
  if (err) throw err;
  console.log(data.toString());
});
```

fs.unlink()

El método fs.unlink() se utiliza para eliminar el archivo. Lanzará un error si el archivo no existe en la ruta dada. Eche un vistazo al código.

```
const fs = require("fs");

fs.unlink("sample.txt", (err) => {
  if (err) throw err;
  console.log("File deleted!");
});
```

fs.rename()

El método fs.rename() se utiliza para cambiar el nombre del archivo. Lanzará un error si el archivo no existe en la ruta dada. Cambie el nombre del siguiente archivo con el siguiente código.

```
const fs = require("fs");

fs.rename("sample.txt", "sample_one.txt", (err) => {
  if (err) throw err;
  console.log("File renamed!");
});
```

Misceláneo

Con estos conocimientos de diferentes métodos de manejo de archivos de la **fs (sistema de archivos)** módulo. Puede realizar la mayoría de las operaciones de archivo utilizando los métodos que ha visto en este tutorial. Como prometimos, veamos un script de ejemplo que abre un archivo y lee su contenido usando el fs.open() e fs.readFile() métodos respectivamente.

```
const fs = require("fs");

fs.open("sample.txt", "r", (err, file) => {
  if (err) throw err;
```

```
  fs.readFile(file, (err, data) => {
    if (err) throw err;
    console.log(data.toString());
  });
});
```

Conclusiones

Al hacer esta investigación me di cuenta de que los lenguajes diferentes a c++ no cuentan con un una documentación tan unificada o bien indexada.

Java, con su enfoque orientado a objetos, proporciona una variedad de clases y métodos en el paquete **java.io** para realizar operaciones de lectura y escritura de archivos. El uso de streams y readers/ writers facilita la manipulación de diferentes tipos de datos y asegura un control preciso sobre la lectura y escritura. Además, el manejo de excepciones es esencial en Java para tratar posibles problemas durante las operaciones de archivo, como la no disponibilidad del archivo o errores de E/S.

Java, orientado a objetos, ofrece un paquete java.io con clases y métodos para operaciones de archivos. El uso de streams y readers/writers facilita la lectura/escritura de datos. La gestión de excepciones es fundamental para manejar contratiempos.

Python se destaca por su sintaxis legible. El contexto with y el "duck typing" simplifican la manipulación de archivos. Funciones como open() y métodos como read(), write(), junto a módulos como os y shutil, son herramientas esenciales.

JavaScript, inicialmente para navegadores, evolucionó con Node.js para el manejo de archivos en servidores. El módulo fs ofrece funciones asincrónicas/sincrónicas. Dado su enfoque en eventos, comprender operaciones asincrónicas y su manejo es vital.

En resumen, sin importar el lenguaje, el manejo de archivos es crucial. Java, Python y JavaScript brindan formas distintas pero válidas de abordar esta tarea.

Referencias

- Ficheros — Fundamentos de programación en Python. (s. f.). https://www2.eii.uva.es/fund_inf/python/notebooks/10_Ficheros/Ficheros.html
- *Gestión de archivos en java*. (s. f.). http://jmoral.es/blog/IO-java
- Shaik, H. K. (2021). Manejo de archivos en JavaScript. *Geekflare*. https://geekflare.com/es/handling-files-in-javascript/