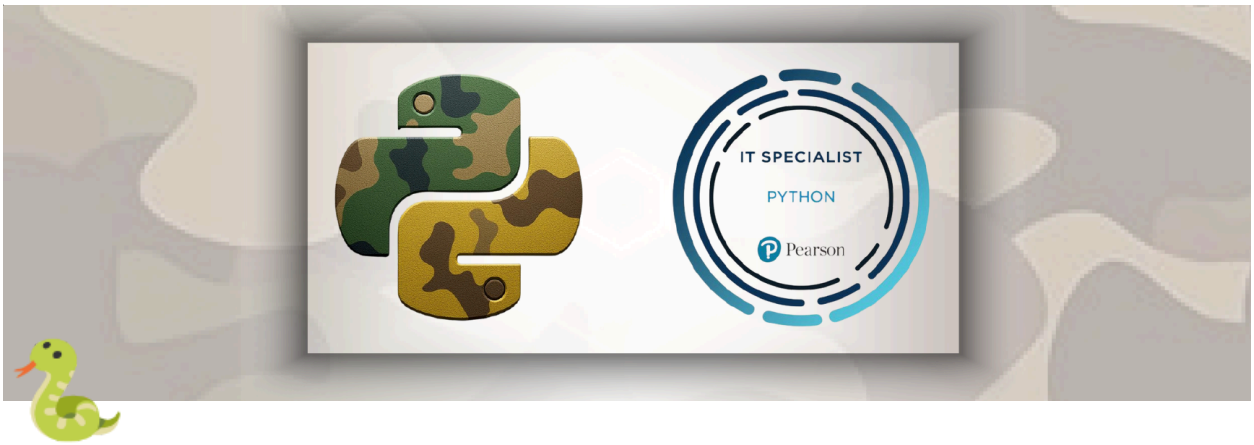

🐍 Clase 11 - Ficheros: parte 2

1. Check existence, delete y with statement

1.1 Sintaxis

```
# Verificar existencia
import os

os.path.exists("archivo.txt")          # True/False
os.path.isfile("archivo.txt")          # True solo si es arch
ivo (no carpeta)
os.path.isdir("carpeta")               # True si es carpeta

# Eliminar archivo
os.remove("archivo_a_borrar.txt")

# with statement (recomendado siempre)
with open("fichero.txt", "r", encoding="utf-8") as f:
    # aquí se usa f
# <- archivo cerrado automáticamente
```

1.2 Ejemplos básicos

Ejemplo 1.2.1

```
# Ejemplo 1 – Verificar existencia
import os
if os.path.exists("datos.txt"):
    print("El archivo existe")
else:
    print("No se encuentra datos.txt")
```

Ejemplo 1.2.2

```
# Ejemplo 2 – Eliminar si existe
# vamos a crear un fichero que luego borraremos
with open("temporal.txt", "w", encoding="utf-8") as f:
    f.write("temporal 1\ntemporal 2\ntemporal 3")

import os
if os.path.exists("temporal.txt"):
    os.remove("temporal.txt")
    print("Archivo temporal eliminado")
```

Ejemplo 1.2.3

```
# Ejemplo 3 – Combinación with + exists
import os

nombre = "prueba.txt"
if not os.path.exists(nombre):
    with open(nombre, "w", encoding="utf-8") as f:
        f.write("Archivo creado automáticamente\\n")
    print("Archivo creado")
else:
    print("El archivo ya existía")
```

1.3 Ejercicios resueltos

Ejercicio 1.3.1

> Si `backup.txt` existe, elimínalo y luego crea uno nuevo con contenido.

```
import os

if os.path.exists("backup.txt"):
    os.remove("backup.txt")
    print("Backup anterior eliminado")

with open("backup.txt", "w", encoding="utf-8") as f:
    f.write("Nueva copia de seguridad\n")
print("Backup creado")
```

Ejercicio 1.3.2

> Comprueba si existe `config.json`. Si no existe, crea uno con dos líneas por defecto.

```
import os

if not os.path.exists("config.json"):
    with open("config.json", "w", encoding="utf-8") as f:
        f.write('{\n')
        f.write('  "theme": "dark"\n')
        f.write('}\n')
    print("Archivo config.json creado con valores por defect
o")
else:
    print("config.json ya existe")
```

Ejercicio 1.3.3

> Limpia todos los archivos `.tmp` que existan en el directorio actual (versión básica).

```
import os

for archivo in os.listdir("."):
    if archivo.endswith(".tmp"):
        try:
            os.remove(archivo)
            print(f"Eliminado: {archivo}")
        except:
            print(f"No se pudo eliminar: {archivo}")
```

1.4 Ejercicios propuestos

Ejercicio 1.4.1

> Si el archivo `cache.dat` existe y tiene más de 1.000.000 bytes, elimínalo.

Salida esperada (ejemplo):

```
Archivo cache.dat demasiado grande → eliminado
```

Ejercicio 1.4.2

> Comprueba si existe `ranking.txt`. Si no existe → créalo con la línea "Posición 1: Nadie". Si existe → agrega la línea "Nueva entrada registrada"

Salida esperada (segunda ejecución):

```
ranking.txt ya existía → nueva entrada añadida
```

Ejercicio 1.4.3

> Antes de escribir en `resultados.txt`, verifica que no exista. Si existe, elimínalo primero y avisa al usuario.

Salida esperada (si existía):

```
Archivo resultados.txt encontrado → eliminado para nueva ejec
ución
```

1.5 Caso de estudio propuesto

Caso: Gestor de notas simple con protección

El programa debe:

1. Preguntar nombre de la nota (ej: reunion-2025-07.txt)
2. Verificar si ya existe
3. Si existe → preguntar si desea sobreescribir (s/n)
4. Si no existe o si el usuario acepta → pedir contenido (varias líneas) hasta escribir "FIN"
5. Guardar el contenido usando `with`

Salida esperada (ejemplo de interacción):

```
Nombre de la nota: reunion-equipo
El archivo ya existe. ¿Desea sobreescribir? (s/n): s
Escribe el contenido (escribe FIN para terminar):

[usuario escribe varias líneas]
FIN

Nota guardada correctamente en reunion-equipo.txt
```

---

2. Manejando el puntero

> El **puntero de archivo** marca la posición actual de lectura/escritura.

💡 Métodos esenciales:
`f.tell()` → posición actual (en bytes). `f.seek(offset, whence=0)` → mueve el puntero:

- `whence=0` (inicio, *default*): desplaza desde el inicio.
- `whence=1` (posición actual): desplaza relativo a la posición actual.
- `whence=2` (final): desplaza relativo al final (offsets negativos).

- `f.tell()` → posición actual (en bytes).
- `f.seek(offset, whence=0)` → mueve el puntero:
  - `whence=0` (inicio, *default*): desplaza desde el inicio.
  - `whence=1` (posición actual): desplaza relativo a la posición actual.
  - `whence=2` (final): desplaza relativo al final (offsets negativos).


```
from io import open

texto = "Una línea con texto\nOtra línea con texto"

# Ruta donde crearemos el fichero, w indica escritura (punter
o al principio)
fichero = open('fichero1.txt','w')

# Escribimos el texto
fichero.write(texto)

# Cerramos el fichero
fichero.close()
```

Es posible posicionar el puntero en el fichero manualmente usando el método *seek* e indicando un número de caracteres para luego leer una cantidad de caracteres con el método *read*:

```
fichero = open('fichero1.txt','r')
fichero.seek(0)   # Puntero al principio
fichero.read(10)  # Leemos 10 carácteres
```

Para posicionar el puntero justo al inicio de la segunda línea, podríamos ponerlo justo en la longitud de la primera:

```
fichero = open('fichero1.txt','r')
fichero.seek(0)

# Leemos la primera línea y situamos el puntero al principio
de la segunda
fichero.seek( len(fichero.readline()) )

# Leemos todo lo que queda del puntero hasta el final
fichero.read()
```

Ejemplo resuelto 1 — Releer una sección desde una marca

> **Objetivo:** leer los primeros 10 caracteres, saltar 5, leer los siguientes 8.

```
from io import StringIO

contenido = "Python intermedio: manejo de ficheros.\nLínea
2.\nLínea 3."
f = StringIO(contenido) # Este es el equivalente a open()

primero_10 = f.read(10)            # 'Python int'
pos = f.tell()                     # 10
f.seek(pos + 5, 0)                 # salta 5 desde inicio (po
s=10 → 15)
siguientes_8 = f.read(8)           # lee 8 chars

print("primero_10:", primero_10)
print("tell_inicial:", pos)
```

```
print("después_seek:", f.tell())
print("siguientes_8:", siguientes_8)
```

💡 Ejercicio propuesto. Rehacer el código anterior pero con `with open`

Ejemplo resuelto 2 — Leer los últimos N bytes (binario)

> **Objetivo:** extraer un *footer* de 16 bytes de un archivo binario.

```
# Create a bin file
content = b"Este es un archivo binario de prueba.\nContiene d
atos aleatorios:\n" + bytes(range(0, 256))
filename = "archivo.bin"
with open(filename, "wb") as f:
    f.write(content)

filename, len(content)
```

```
# patrón típico
ruta = "archivo.bin"
N = 16
with open(ruta, "rb") as f:
    f.seek(-N, 2)     # desde el final
    tail = f.read(N)

print("Footer (hex):", tail.hex())
```

💡 `seek(-N, 2)` posiciona el puntero N bytes antes del final.

3. Módulo pickle

Este módulo nos permite almacenar fácilmente colecciones y objetos en ficheros binarios abstrayendo todo la parte de escritura y lectura binaria.

Escritura de colecciones

```
import pickle

# Podemos guardar lo que queramos, listas, diccionarios, tupl
as...
lista = [1,2,3,4,5]

# Escritura en modo binario, vacía el fichero si existe
fichero = open('lista.pckl','wb')

# Escribe la colección en el fichero
pickle.dump(lista, fichero)

fichero.close()
```

Lectura de colecciones

```
# Lectura en modo binario
fichero = open('lista.pckl','rb')

# Cargamos los datos del fichero
lista_fichero = pickle.load(fichero)
print(lista_fichero)

fichero.close()
```

Persistencia de objetos

Para guardar objetos lo haremos dentro de una colección. La lógica sería la siguiente:

1. Crear una colección.
2. Introducir los objetos en la colección.
3. Guardar la colección haciendo un dump.
4. Recuperar la colección haciendo un load.
5. Seguir trabajando con nuestros objetos.

4. Ficheros CSV

CSV: Valores separados por comas (Comma Separated Values)

Documentación: https://docs.python.org/3/library/csv.html

🧩 4.1. El módulo estándar `csv`

Python incluye el módulo `csv` en la biblioteca estándar.

No necesitas instalar nada y te ofrece:

- Lectura robusta de archivos CSV.
- Escritura correcta de filas, delimitadores y comillas.
- Soporte para encabezados, dialectos, delimitadores personalizados, quoting, etc.

Usarlo es preferible a leer CSV "a mano" con `split()` porque el módulo **maneja correctamente**:

- Campos entrecomillados
- Delimitadores dentro de campos
- Líneas con saltos "irregulares"
- Escapado de caracteres
- Diferencias entre sistemas operativos

🪟 2. Apertura recomendada (especialmente en Windows)

En Windows, si abres un CSV con:

```
open("archivo.csv")
```

es probable que al escribir obtengas **líneas en blanco extra** debido al manejo distinto de saltos de línea del sistema operativo.

💡 **Forma correcta:**

```
open("archivo.csv", "r", encoding="utf-8", newline="")
```

> 💡 **¿Por qué `newline=""`?**
>
> - Indica al módulo `csv` que gestione él mismo los saltos de línea.
> - Evita dobles saltos como `\r\n` → `\r\r\n`, que generan líneas vacías.
> - Funciona perfecto tanto para lectura como para escritura.

📖 3. Lectura de CSV

✔️ Opción A: `csv.reader`

Produce **listas**, una por cada fila:

```
import csv

with open("archivo.csv", encoding="utf-8", newline="") as f:
    reader = csv.reader(f, delimiter=",")
    for fila in reader:
        print(fila)
```

Una fila como:

```
1,Ana,8.5
```

Se convierte en:

```
['1', 'Ana', '8.5']
```

Es decir:

- No hay tipos (todo es `str`)
- No hay nombres de columnas
- Es rápido y simple

✔️ Opción B: `csv.DictReader`

Convierte cada fila en un **diccionario usando el encabezado como claves**:

```
import csv

with open("archivo.csv", encoding="utf-8", newline="") as f:
    reader = csv.DictReader(f)
    for fila in reader:
        print(fila)
```

Ejemplo:

```
{'id': '1', 'nombre': 'Ana', 'nota': '8.5'}
```

Ventajas:

- Accedes por nombre: `fila["nota"]`
- Más legible y seguro cuando hay muchas columnas
- Ideal para procesamiento de datos (ETL)

🖊️ 4. Escritura de CSV

✔️ Opción A: `csv.writer`

Escribe **listas** como filas:

```
import csv

with open("salida.csv", "w", encoding="utf-8", newline="") as
f:
    writer = csv.writer(
        f,
        delimiter=",",
        quoting=csv.QUOTE_MINIMAL,   # controla uso de comill
as
        lineterminator="\n"          # evita problemas cross-
plataforma
    )
    writer.writerow(["id", "nombre", "nota"])
    writer.writerow([1, "Ana", 8.5])
```

✔️ Opción B: `csv.DictWriter`

Igual que Reader, pero escribiendo **diccionarios**:

```
writer = csv.DictWriter(f, fieldnames=["id","nombre","nota"])
writer.writeheader()     # escribe encabezado
writer.writerow({"id":1, "nombre":"Ana", "nota":8.5})
```

Ventajas:

- Orden consistente de columnas
- Evita errores por columnas desordenadas
- Más expresivo, más seguro

🧱 5. Quoting (manejo de comillas)

`csv` debe decidir cuándo poner campos entre comillas.

Los modos más importantes son:

- `csv.QUOTE_MINIMAL` (recomendado)

  Solo pone comillas si es necesario:

  - Si el campo contiene el delimitador (p. ej., una coma)
  - Si contiene saltos de línea
  - Si contiene comillas

- `csv.QUOTE_ALL`

  Cada campo se escribe siempre entre comillas.

  Útil para sistemas muy estrictos o CSV "compatibles con Excel".

- `csv.QUOTE_NONE`

  No usa comillas nunca (raro, riesgo alto de errores).

Ejemplo:

Si un campo contiene:

```
Hola, mundo
```

`QUOTE_MINIMAL` producirá:

```
"Hola, mundo"
```

🔍 6. Detectar automáticamente el dialecto (`Sniffer`)

A veces no sabes:

- si el CSV usa `,` o `;`
- si usa comillas simples o dobles
- si tiene encabezado
- si usa quoting estricto o no

Python permite **detectar el formato automáticamente**:

```
import csv

with open("archivo.csv", encoding="utf-8") as f:
    sample = f.read(2048)
    dialecto = csv.Sniffer().sniff(sample)
    f.seek(0)   # volvemos al inicio
    reader = csv.reader(f, dialecto)
```

También puede adivinar si tiene encabezado:

```
csv.Sniffer().has_header(sample)
```

Útil para archivos generados por otras aplicaciones, especialmente Excel o sistemas legacy.

Escritura de listas en CSV

```
import csv

contactos = [
    ("Manuel", "Desarrollador Web", "manuel@ejemplo.com"),
    ("Lorena", "Gestora de proyectos", "lorena@ejemplo.com"),
    ("Javier", "Analista de datos", "javier@ejemplo.com"),
    ("Marta", "Experta en Python", "marta@ejemplo.com")
]

with open("contactos.csv", "w", newline="\n") as csvfile:
    writer = csv.writer(csvfile, delimiter=",")
    for contacto in contactos:
        writer.writerow(contacto)
```

Lectura de listas en CSV

```
with open("contactos.csv", newline="\n") as csvfile:
    reader = csv.reader(csvfile, delimiter=",")
    for nombre, empleo, email in reader:
        print(nombre, empleo, email)
```

Escritura de diccionarios en CSV

```
import csv

contactos = [
    ("Manuel", "Desarrollador Web", "manuel@ejemplo.com"),
    ("Lorena", "Gestora de proyectos", "lorena@ejemplo.com"),
    ("Javier", "Analista de datos", "javier@ejemplo.com"),
    ("Marta", "Experta en Python", "marta@ejemplo.com")
]

with open("contactos.csv", "w", newline="\n") as csvfile:
    campos = ["nombre", "empleo", "email"]
    writer = csv.DictWriter(csvfile, fieldnames=campos)
    writer.writeheader()
    for nombre, empleo, email in contactos:
        writer.writerow({
            "nombre": nombre, "empleo": empleo, "email": emai
l
        })
```

Lectura de diccionarios en CSV

```
with open("contactos.csv", newline="\n") as csvfile:
    reader = csv.DictReader(csvfile)
    for contacto in reader:
```

```
        print(contacto["nombre"], contacto["empleo"], contact
o["email"])
```

Leer y calcular promedio por columna

> **Objetivo:** leer un CSV con encabezado y calcular el promedio de la columna `nota`.

Suposición de datos (`alumnos_notas.csv`):

```
id,nombre,nota
1,Ana,8.5
2,Luis,7.0
3,Marta,9.25
```

Solucion:

```
import csv
from pathlib import Path

ruta = Path("alumnos_notas.csv")

# Lectura del CSV y conversión de tipos
with ruta.open("r", encoding="utf-8", newline="") as f:
    reader = csv.DictReader(f)
    notas = [float(row["nota"]) for row in reader]

prom = sum(notas) / len(notas)
print(f"Promedio: {prom:.2f}")
```

Filtrar y escribir un nuevo CSV

> **Objetivo:** filtrar filas con `nota ≥ 8` y escribir un archivo nuevo con las seleccionadas.

Entrada (`alumnos_notas.csv`):

```
id,nombre,nota
1,Ana,8.5
2,Luis,7.0
3,Marta,9.25
4,Jose,5.5
```

Solución

```
import csv
from pathlib import Path

entrada = Path("alumnos_notas.csv")
salida = Path("alumnos_filtrados.csv")

# Leer y filtrar
with entrada.open("r", encoding="utf-8", newline="") as f_in:
    reader = csv.DictReader(f_in)
    filtradas = [r for r in reader if float(r["nota"]) >= 8.
0]

# Escribir resultado
campos = ["id", "nombre", "nota"]
with salida.open("w", encoding="utf-8", newline="") as f_out:
    writer = csv.DictWriter(f_out, fieldnames=campos, lineter
minator="\n")
    writer.writeheader()
    writer.writerows(filtradas)

print(f"Escrito: {salida.resolve()}")
```

5. Ficheros JSON

JSON: Notación de objeto de JavaScript (JavaScript Object Notation)

Documentación: https://docs.python.org/3/library/json.html

5.1. 📦 Módulo estándar `json`

Python incluye el módulo `json` en su biblioteca estándar. Esto significa:

- No necesitas instalar nada.
- Funciona igual en Linux, Windows y macOS.
- Está pensado para interoperar con APIs, archivos de configuración, ETL, etc.

🔄 Conversión JSON ↔ Python

De JSON → Python

Se usa cuando lees un archivo JSON o una cadena JSON que contiene datos estructurados.

- `json.loads(s)`
  Convierte un *string* JSON en un objeto Python.
- `json.load(f)`
  Lee JSON desde un archivo abierto con `open()`.

Ejemplo conceptual:

```
json.loads('{"a": 1, "b": true}')
```

Produce:

```
{"a": 1, "b": True}
```

> Observa cómo `true` (JSON) se convierte en `True` (Python).

De Python → JSON

Útil para guardar datos en archivo, enviar datos a APIs o generar configuraciones.

- `json.dumps(obj)`
  Convierte un objeto Python en una cadena JSON.

- `json.dump(obj, f)`
  Escribe el JSON directamente en un archivo.

Ejemplo:

```
json.dumps({"x": 1})
```

Produce:

```
{"x": 1}
```

5.2. 🖊️ Formateo y compatibilidad

Esta parte es fundamental para trabajar con archivos JSON "reales" (no solo ejemplos pequeños).

- 🔹 `indent=2`

Hace que el JSON generado sea *legible*, con saltos de línea y sangría.

```
json.dumps(obj, indent=2)
```

Sin `indent` → una sola línea.
Con `indent=2` → ideal para logs, configuraciones, pipelines mantenibles.

ensure_ascii=False

Controla si se fuerzan los caracteres a ASCII.

- `ensure_ascii=True` (por defecto)

  Convierte caracteres como "á, ñ, ü" en secuencias Unicode:

  → `"Jos\u00e9"`

- `ensure_ascii=False`

  Mantiene los caracteres tal cual:

  → `"José"`

Esto es crucial para:

- Archivos con acentos
- APIs internacionales
- Documentos que deben ser legibles por humanos

🔹 `sort_keys=True` (JSON determinista)

Ordena alfabéticamente las claves del diccionario antes de generar el JSON.

```
json.dumps(obj, sort_keys=True)
```

¿Por qué es útil?

- Facilita comparaciones (`diff`)
- Garantiza salidas idénticas en entornos diferentes

5.3. 🔥 Diferencias entre tipos JSON y Python

JSON NO es un espejo perfecto de Python. Algunas estructuras **no existen** en JSON.

💡 Conversión automática:

Python	JSON
`dict`	objeto JSON `{}`
`list`	arreglo `[]`
`tuple`	**lista** `[]` (los tupla NO existen en JSON)
`str`	string JSON
`int/float/bool`	números / booleanos JSON
`None`	`null`

❌ Tipos NO soportados por JSON:

- `set`
- `tuple` (se convierten en `list`)
- `datetime`

- `bytes`
- Objetos personalizados (clases)

Si pasas uno de estos a `json.dumps()` sin ayuda → **Error**.

Ejercicio 1: Escritura de datos en JSON

```
import json

contactos = [
    ("Manuel", "Desarrollador Web", "manuel@ejemplo.com"),
    ("Lorena", "Gestora de proyectos", "lorena@ejemplo.com"),
    ("Javier", "Analista de datos", "javier@ejemplo.com"),
    ("Marta", "Experta en Python", "marta@ejemplo.com")
]

datos = []

for nombre, empleo, email in contactos:
    datos.append({"nombre":nombre, "empleo":empleo, "email":e
mail})

with open("contactos.json", "w") as jsonfile:
    json.dump(datos, jsonfile)
```

Ejercicio 2: Lectura de datos en JSON

```
with open("contactos.json") as jsonfile:
    datos = json.load(jsonfile)
    for contacto in datos:
        print(contacto["nombre"], contacto["empleo"], contact
o["email"])
```

Ejercicio 3: Leer, filtrar y hacer *pretty-print*

Contenido de entrada (archivo `usuarios.json`):

```
[
  {"id": 1, "nombre": "Ana",   "activo": true,  "nota": 8.5},
  {"id": 2, "nombre": "Luis",  "activo": false, "nota": 7.0},
  {"id": 3, "nombre": "Marta", "activo": true,  "nota": 9.25}
]
```

Solución:

```
import json
from pathlib import Path

ruta = Path("usuarios.json")

# Leer el archivo JSON
with ruta.open("r", encoding="utf-8") as f:
    datos = json.load(f)

# Filtrar usuarios activos
activos = []
for d in datos:
    if d.get("activo") is True:
        activos.append(d)

# Pretty-print con acentos correctos
print(json.dumps(activos, ensure_ascii=False, indent=2, sort_
keys=True))
```

Salida esperada:

```
[
  {
    "activo": true,
    "id": 1,
    "nombre": "Ana",
```

```
    "nota": 8.5
  },
  {
    "activo": true,
    "id": 3,
    "nombre": "Marta",
    "nota": 9.25
  }
]
```

Ejercicio 4: Actualizar estructura y guardar a archivo

```
import json
from pathlib import Path

ruta_salida = Path("perfil.json")

# Perfil inicial
perfil = {
    "usuario": "juan",
    "roles": ["viewer"],
    "preferencias": {"tema": "claro", "lang": "es"}
}

# Actualizar datos
perfil["roles"].append("editor")
perfil["preferencias"]["tema"] = "oscuro"

# Guardar en archivo JSON
with ruta_salida.open("w", encoding="utf-8") as f:
    json.dump(perfil, f, ensure_ascii=False, indent=2, sort_k
eys=True)

# Volver a cargar para verificar
with ruta_salida.open("r", encoding="utf-8") as f:
```

```
    verificado = json.load(f)

print(verificado["roles"])
print(verificado["preferencias"]["tema"])
```