Manejo y uso de Ficheros en Python

Introducción

El manejo de ficheros es una competencia fundamental en Python para leer, procesar, transformar y persistir datos. Python ofrece dos enfoques complementarios: la API clásica basada en la función open() y la abstracción orientada a objetos de pathlib, que simplifica el trabajo con rutas y mejora la portabilidad entre sistemas operativos.

En esta guía abordaremos lectura y escritura en modo texto y binario, codificaciones y saltos de línea, recorridos eficientes sin cargar el archivo completo en memoria (streaming), y operaciones comunes como copiar, mover, comprimir o calcular hashes. También veremos metadatos (tamaño, fechas, permisos) ~~y técnicas de escritura segura (atómica); mapeo de memoria con mmap y~~ patrones de diseño útiles.

Prestaremos especial atención a la robustez: manejo de excepciones, validación de rutas para prevenir path traversal, diferencias entre Windows y Linux (permisos y fin de línea), y consideraciones de rendimiento como el tamaño de bloque y el buffer de E/S.

La guía incluye ejemplos listos para usar y pequeñas recetas que podrás incorporar directamente en tus proyectos o prácticas docentes.

La base de datos que vamos a usar en Python es SQLITE. Se almacena todo en un único fichero.

Rutas y pathlib

pathlib proporciona la clase Path, con operadores y métodos expresivos para construir y manipular rutas.

```
from pathlib import Path

base = Path.home() / "datos"
logs = base / "logs" / "app.log"

if not logs.parent.exists():
    logs.parent.mkdir(parents=True, exist_ok=True)

for p in base.rglob("*.txt"):
    print(p.name, p.stat().st_size)
```

Carpeta en la que se ejecuta el programador.

Sobrecarga de operadores: los operadores hacen cosas diferentes en base a los datos que están tomándose.

Si no existe, lo crea

De todos los archivo de ese tipo, te imprime nombre y tamaño en bytes.

Operaciones útiles con Path:

```
p = Path("notas.txt")
print(p.resolve())        # Ruta absoluta
print(p.exists())         # ¿Existe?
print(p.is_file(), p.is_dir())
print(p.suffix, p.stem)   # '.txt', 'notas'
print(p.with_suffix('.bak'))
```

stem es el nombre sin el sufijo

```
p.rename("notas_old.txt")
```

File(java) = Path(Python)

Abrir archivos y modos

La función open() admite modos de apertura:

```
# Texto
open('archivo.txt', 'r', encoding='utf-8')   # lectura
open('archivo.txt', 'w', encoding='utf-8')   # escritura (sobrescribe)
open('archivo.txt', 'a', encoding='utf-8')   # anexar
```

append, añadiendo cosas por el final

```
# Binario
open('datos.bin', 'rb')   # lectura
open('datos.bin', 'wb')   # escritura
open('datos.bin', 'ab')   # anexar
```

Lectura línea a línea (streaming) en texto:

```
with open('poema.txt', encoding='utf-8') as f:
    for linea in f:
        print(linea.rstrip())
```

Si hay un error, la variable f (el nombre lógico del open) se encarga de cerrar el fichero.

Lectura en binario con offsets (seek/tell):

```
with open('datos.bin', 'rb') as f:
    f.seek(100)                 # desplazar a offset 100
    chunk = f.read(16)          # leer 16 bytes
    print(f.tell(), chunk.hex())
```

Te desplaza al byte 100 del fichero

Convierte la secuencia de bytes a hexadecimal

```
with open("salida.txt", "w", encoding="utf-8") as f:
    f.write("resultado\n")
```

Si salida ya existe, nos lo cargamos.

```
with open("imagen.png", "rb") as f:
    datos = f.read()
```

Si con read no pones nada, lees todo el fichero.

La elección de encoding es crítica; utf-8 es el estándar recomendado.

Ficheros de texto: patrones de lectura y escritura

Patrones habituales: leer todo, leer por líneas y escribir con formato.

```
from pathlib import Path

texto = Path("poema.txt").read_text(encoding="utf-8")
```

crea el path y leemos el texto

```
with open("grande.log", encoding="utf-8") as f:
```

por defecto, modo lectura y modo texto; pero siempre ponerlo

```
    for linea in f:
        if "ERROR" in linea:
            print(linea.rstrip())

Path("reporte.txt").write_text("\n".join(f"Item {i}" for i in range(10)),
encoding="utf-8")
```

Solo imprime aquellas líneas que contiene la palabra ERROR

Ficheros binarios, posicionamiento y ficheros grandes

Para binarios, usa rb/wb. Los métodos seek() y tell() permiten navegación aleatoria. Procesar por bloques mejora memoria y rendimiento.

```
tam_bloque = 1024 * 1024
with open("origen.bin", "rb") as fin, open("destino.bin", "wb") as fout:
    while True:
        bloque = fin.read(tam_bloque)
        if not bloque:
            break
        fout.write(bloque)

with open("datos.bin", "rb") as f:
    f.seek(100)
    chunk = f.read(16)
    pos = f.tell()
    print(pos, chunk)
```

1 megabyte Leer ficheros en trozos de un mega.

Codificaciones y saltos de línea

Trabaja siempre con encoding explícito (UTF-8) para evitar sorpresas. Controla los saltos de línea con newline si es necesario.

```
# Escritura con UTF-8 y fin de línea estándar
with open('salida.txt', 'w', encoding='utf-8', newline='\n') as f:
    f.write('Primera línea\nSegunda línea\n')
```

Escritura atómica y copias de seguridad

Para evitar archivos corruptos ante fallos intermedios, escribe en un temporal y reemplaza con os.replace().

```
import os, tempfile
from pathlib import Path

dest = Path('config.json')
tmp = tempfile.NamedTemporaryFile('w', delete=False, encoding='utf-8')
try:
    tmp.write('{"ok": true}\n')
```

```
    tmp.flush()
    os.replace(tmp.name, dest)
finally:
    try: os.unlink(tmp.name)
    except FileNotFoundError: pass
```

Metadatos, permisos y tiempos

Puedes acceder a tamaño, fechas y permisos con stat(). En Unix, chmod permite marcar ejecutables; en Windows el modelo de permisos es distinto.

```
from pathlib import Path
import stat
p = Path('script.sh')
st = p.stat()
print('bytes=', st.st_size, 'mtime=', st.st_mtime)

# Hacer ejecutable en Unix
p.chmod(st.st_mode | stat.S_IXUSR)
```

Copiar, mover, eliminar y directorios

```
from pathlib import Path
import shutil

src = Path('origen.bin')
dst = Path('copia.bin')

# Copiar binario
with src.open('rb') as fsrc, dst.open('wb') as fdst:
    shutil.copyfileobj(fsrc, fdst, length=1024*1024)  # 1 MiB por
bloque

# Mover/renombrar
dst.rename('copia_final.bin')

# Directorios
Path('out').mkdir(parents=True, exist_ok=True)
shutil.rmtree('carpeta_temporal', ignore_errors=True)
```

CSV y JSON (librería estándar)

```
import csv, json
from pathlib import Path
```

```
# CSV -> JSON
rows = []
with open('alumnos.csv', newline='', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    for row in reader:
        rows.append(row)
Path('alumnos.json').write_text(json.dumps(rows, ensure_ascii=False,
indent=2), encoding='utf-8')

# JSON -> CSV (cabeceras fijas)
with open('ventas.json', encoding='utf-8') as f:
    data = json.load(f)
fieldnames = ['id','producto','precio','cantidad']
with open('ventas.csv','w',newline='',encoding='utf-8') as f:
    w = csv.DictWriter(f, fieldnames=fieldnames)
    w.writeheader()
    for d in data:
        w.writerow({k: d.get(k,'') for k in fieldnames})
```

Compresión y empaquetado: gzip, zipfile, tarfile

```
import gzip, shutil, zipfile, tarfile
from pathlib import Path

# gzip streaming
with open('grande.log','rb') as fin, gzip.open('grande.log.gz','wb') as
fout:
    shutil.copyfileobj(fin, fout, 1024*1024)

# ZIP preservando estructura relativa
base = Path('carpeta')
with zipfile.ZipFile('backup.zip','w',
compression=zipfile.ZIP_DEFLATED) as z:
    for p in base.rglob('*'):
        if p.is_file():
            z.write(p, arcname=p.relative_to(base))

# TAR.GZ con exclusión
with tarfile.open('proyecto.tar.gz','w:gz') as tar:
    for p in Path('proyecto').rglob('*'):
        if p.suffix != '.tmp':
            tar.add(p, arcname=p.relative_to('proyecto'))
```

Binarios estructurados y mmap

struct permite empaquetar/desempaquetar datos binarios con formatos definidos; mmap mapea el fichero a memoria para búsquedas rápidas.

```
import struct, mmap

# struct: escribir entero y float en binario (little-endian)
with open('datos.bin','wb') as f:
    f.write(struct.pack('<if', 42, 3.14))

with open('datos.bin','rb') as f:
    entero, real = struct.unpack('<if', f.read(8))
    print(entero, real)

# mmap: buscar firma ZIP en archivo grande
with open('archivo_grande.bin','rb') as f:
    mm = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ)
    sig = b'\x50\x4B\x03\x04'
    idx = mm.find(sig)
    print('offset', idx)
    mm.close()
```

Bloqueo por fichero y rotación de logs

Como alternativa simple y portable, usa lockfiles (con limitaciones). La rotación por tamaño o fecha puede implementarse con rename + compresión.

```
from pathlib import Path
import time, os

lock = Path('tarea.lock')
while lock.exists():
    time.sleep(0.2)
try:
    lock.write_text('locked', encoding='utf-8')
    # Trabajo crítico...
    pass
finally:
    with contextlib.suppress(FileNotFoundError):
        lock.unlink()

# Rotación por tamaño
log = Path('app.log')
if log.exists() and log.stat().st_size > 5*1024*1024:
    rot = Path('app.log.1')
    if not rot.exists():
```

```
        os.replace(log, rot)
```

Errores comunes y buenas prácticas

• Usar siempre context managers: with open(...): garantiza cierre incluso con excepciones.

• Declarar siempre encoding al abrir en texto para portabilidad (UTF-8).

• Evitar leer archivos enormes de una vez; usar bloques o iteradores.

• Escritura atómica para configuraciones o datos críticos (temporal + os.replace).

• Validar rutas de entrada del usuario para evitar path traversal.

• Para rendimiento, ajustar tamaño de bloque (1–8 MiB) en copias grandes.

• Documentar diferencias de permisos y fin de línea entre sistemas.

Recetas rápidas

```
# Tamaño "humano"
def human(n):
    for u in ('B','KB','MB','GB','TB'):
        if n < 1024:
            return f'{n:.0f} {u}'
        n /= 1024
    return f'{n:.0f} PB'

# Hash SHA-256 por bloques
import hashlib
def sha256_file(path, block=1024*1024):
    h = hashlib.sha256()
    with open(path,'rb') as f:
        for chunk in iter(lambda: f.read(block), b''):
            h.update(chunk)
    return h.hexdigest()
```

Conclusión

Python ofrece un ecosistema sólido para el manejo de ficheros que cubre desde tareas simples hasta escenarios de alto rendimiento.

El uso de pathlib simplifica las rutas y mejora la claridad del código, mientras que open() y la librería estándar permiten trabajar con múltiples formatos.

Adoptar buenas prácticas (encoding explícito, streaming, escritura atómica, validación de rutas) incrementa la robustez y seguridad de tus aplicaciones.

Esta guía puede servirte como material de referencia y base para prácticas, exámenes o proyectos de integración.