Entrada y salida por archivos

Ricardo Pérez López

IES Doñana, curso 2025/2026

Generado el 2026/02/03 a las 20:37:00

Índice

1. Introducción

1.1. Introducción

Como ya sabemos, toda la comunicación con el exterior se lleva a cabo a través de **flujos**, que son secuencias de bytes o caracteres.

Por tanto, cuando queramos leer y/o escribir datos en un archivo, lo haremos también a través de un flujo de bytes o de caracteres.

Para ello, deberemos seguir los siguientes pasos, en este orden:

1. Abrir el archivo en el modo adecuado con `open`.
2. Realizar las operaciones deseadas sobre el archivo.
3. Cerrar el archivo con `close`.

Mientras el archivo está abierto, disponemos de un **puntero** que *apunta* a la posición actual de lectura o escritura, es decir, a la posición donde se hará la siguiente operación de lectura o escritura con el archivo.

Ese puntero indica la posición actual dentro del flujo de caracteres o bytes a través del cual accedemos al archivo.

2. Operaciones de apertura y cierre

2.1. open

La función `open` abre un archivo y devuelve un objeto que lo representa.

Su signatura es:

```
open(nombre: str [, modo: str]) -> archivo
```

El *nombre* es una cadena que contiene el nombre del archivo a abrir.

El *modo* es otra cadena que contiene caracteres que describen de qué forma se va a usar el archivo.

El valor devuelto es un objeto que representa al archivo abierto y cuyo tipo depende del modo en el que se ha abierto el archivo.

Hay dos modos principales de abrir un archivo: en **modo texto** (en el que se leen y escriben *cadenas*) y en **modo binario** (en el que se leen y escriben *bytes*) y luego existen modos secundarios en función de lo que se vaya a hacer con el archivo.

Los valores posibles de *modo* aparecen en las siguientes tablas.

Modo texto

Modo	Significado	El puntero se coloca...
`'r'`	Abre sólo para lectura de texto.	Al principio.
`'r+'`	Abre para lectura/escritura de texto.	Al principio.
`'w'`	Abre sólo para escritura de texto. Vacía y sobreescribe el archivo si ya existe. Si no existe, lo crea y lo abre sólo para escritura.	Al principio.
`'w+'`	Abre para lectura/escritura de texto. Vacía y sobreescribe el archivo si ya existe. Si no existe, lo crea y lo abre para lectura/escritura.	Al principio.

Modo	Significado	El puntero se coloca…
`'a'`	Abre para añadir texto. Si el archivo no existe, lo crea y lo abre sólo para escritura.	Al final si el archivo ya existe.
`'a+'`	Abre para lectura/añadir en modo texto. Si el archivo no existe, lo crea y lo abre para lectura/escritura.	Al final si el archivo ya existe.

Modo binario

Modo	Significado	El puntero se coloca…
`'rb'`	Abre sólo para lectura en binario.	Al principio.
`'rb+'`	Abre para lectura/escritura en binario.	Al principio.
`'wb'`	Abre sólo para escritura en binario. Vacía y sobreescribe el archivo si ya existe. Si no existe, lo crea y lo abre sólo para escritura.	Al principio.
`'wb+'`	Abre para lectura/escritura en binario. Vacía y sobreescribe el archivo si ya existe. Si no existe, lo crea y lo abre para lectura/escritura.	Al principio.
`'ab'`	Abre para añadir en binario. Si el archivo no existe, lo crea y lo abre sólo para escritura.	Al final si el archivo ya existe.
`'ab+'`	Abre para lectura/añadir en binario. Si el archivo no existe, lo crea y lo abre para lectura/escritura.	Al final si el archivo ya existe.

Resumen básico:

- Si no se pone `'b'` (modo binario), se entiende que es `'t'` (modo texto).
- El modo predeterminado es `'r'` (abrir para lectura en modo texto, sinónimo de `'rt'`).
- Los modos `'a'`, `'ab'`, `'a+'` y `'ab+'` abren el archivo si ya existía previamente, o lo crean nuevo si no existía.
- Los modos `'w'`, `'wb'`, `'w+'` y `'wb+'` abren el archivo y lo vacía (borra su contenido) si ya existía previamente, o lo crean nuevo si no existía.
- Los modos `'r+'` y `'rb+'` abren el archivo sin borrarlo.
- El modo `'x'` abre el archivo en modo exclusivo, produciendo un error si el archivo ya existía.

Normalmente, los archivos se abren en **modo texto**, lo que significa que se leen y se escriben cadenas (valores de tipo `str`) desde y hacia el archivo, las cuales se codifican según una codificación específica que depende de la plataforma.

Por ejemplo, los saltos de línea se escriben como `\n` en Unix o `\r\n` en Windows, y se leen siempre como `\n`.

Al añadir una `b` en el modo se abre el archivo en **modo binario**. En tal caso, los datos se leen y se escriben en forma de objetos de tipo `bytes`.

El modo binario es el que debe usarse cuando se trabaje con archivos que no contengan texto (datos binarios *crudos*).

Ejemplo:

```
f = open('salida.txt', 'w')
```

El tipo de dato que devuelve `open` depende de cómo se ha abierto el archivo:

- Si se ha abierto en **modo texto**, devuelve un `io.TextIOWrapper`.
- Si se ha abierto en **modo binario**, entonces depende:
  - En modo **sólo lectura**, devuelve un `io.BufferedReader`.
  - En modo **sólo escritura o añadiendo al final**, devuelve un `io.BufferedWriter`.
  - En modo **lectura/escritura**, devuelve un `io.BufferedRandom`.

`io` es el módulo que contiene los elementos básicos para manipular flujos.

2.2. close

El método `close` cierra un archivo previamente abierto por `open`, finalizando la sesión de trabajo con el mismo.

Su signatura es:

```
⟨archivo⟩.close()
```

Siempre hay que cerrar un archivo previamente abierto para:

- asegurarse de que los cambios realizados se vuelcan al archivo a través del sistema operativo, y
- liberar inmediatamente los recursos del sistema que pudiera estar consumiendo.

Una vez que se ha cerrado el archivo ya no se podrá seguir usando:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Podemos comprobar si un archivo ya se ha cerrado consultando su atributo `closed`:

```
>>> f = open('archivo.txt', 'r')
>>> f.closed
False
>>> f.close()
```

```
>>> f.closed
True
```

Observar que **no es un método** (no lleva paréntesis), **sino un *campo*** que contiene un valor lógico que el propio objeto modifica al cambiar su estado de abierto a cerrado o viceversa.

2.3. Gestores de contexto

A veces un programa necesita trabajar con recursos externos:

- Archivos locales.
- Conexiones a bases de datos.
- Conexiones de red.

Trabajar con esos recursos siempre implica los siguientes pasos:

1. Abrir el recurso (solicitar la apertura o la conexión al sistema operativo).
2. Usar el recurso.
3. Cerrar el recurso (solicitar su cierre o su desconexión al sistema operativo).

Por ejemplo, al trabajar con archivos hay que:

1. Abrir el archivo con `f = open(...)`.
2. Usar el archivo con `f.read(...)`, `f.write(...)`, etc.
3. Cerrar el archivo con `f.close()`.

La **cantidad de recursos abiertos** al mismo tiempo está **limitada** por el sistema operativo o el intérprete.

Por ejemplo, si intentamos abrir demasiados archivos a la vez, el intérprete nos devolverá el error: `OSError: [Errno 24] Too many open files`.

Además, cada recurso abierto consume, a su vez, recursos del sistema operativo o del intérprete (memoria, descriptores internos, etcétera).

Por ello, es importante **acordarse de cerrar el recurso** una vez hayamos terminado de trabajar con él, para que el sistema operativo o el intérprete pueda liberar los recursos que está consumiendo y éstos se puedan reutilizar.

Para ello, se puede usar un **`try ... finally`**:

```
f = open('hola.txt', 'w')
try:
    f.write('¡Hola, mundo!')
finally:
    f.close()
```

Esto garantiza que el archivo se cerrará aunque el `f.write(...)` levante una excepción.

Los gestores de contexto son un mecanismo más cómodo y elegante para trabajar con recursos y asegurarse de que se cierran al final.

Para ello, se usa la sentencia **`with ... as`**, cuya sintaxis es:

```
⟨gestor_contexto⟩ ::=
    with ⟨expresión⟩ [as ⟨identificador⟩]:
        ⟨sentencia⟩
```

El siguiente código es equivalente al anterior:

```
with open('hola.txt', 'w') as f:
    f.write('¡Hola, mundo!')
```

La sentencia **`with ... as`** es una estructura de control que hace lo siguiente:

1. Evalúa la ⟨*expresión*⟩, que deberá devolver un **gestor de recursos**.

   Los gestores de recursos son objetos que responden a los métodos `__enter__` y `__exit__`.
2. Llama al método `__enter__` sobre el objeto, el cual debe abrir y devolver el recurso.
3. Ese recurso se asigna a la variable del `identificador`.
4. Ejecuta la `sentencia` que, por supuesto, puede ser simple o compuesta.
5. Cuando termina de ejecutar la sentencia, llama al método `__exit__` sobre el objeto inicial, el cual se encargará de cerrar el recurso.

Por tanto, al salir de la estructura de control **`with ... as`**, se garantiza que el recurso asignado a `f` está cerrado.

Eso significa que, en el siguiente código, la última llamada al método `write` fallará al no estar abierto el recurso:

```
>>> with open('hola.txt', 'w') as f:
...     f.write('¡Hola, mundo!')
...
13
>>> f.write('Esto fallará')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

3. Operaciones de lectura

3.1. `read`

Para leer de un archivo, se puede usar el método `read` sobre el objeto que devuelve la función `open`.

Su signatura es:

```
⟨archivo⟩.read([tamaño: int]) -> str | bytes
```

El método devuelve una cadena (tipo `str`) si el archivo se abrió en modo texto, o un objeto de tipo `bytes` si se abrió en modo binario.

El archivo contiene un **puntero interno** que indica hasta dónde se ha leído en el mismo.

Cada vez que se llama al método `read`, se mueve ese puntero para que en posteriores llamadas se continúe leyendo desde ese punto.

Si se alcanza el final del archivo, se devuelve la cadena vacía (`''`).

El parámetro *tamaño* es opcional:

- Si se omite o es negativo, se devuelve todo lo que hay desde la posición actual del puntero hasta el final del archivo.
- En caso contrario, se leerán y devolverán *como mucho* tantos caracteres (en modo texto) o bytes (en modo binario) como se haya indicado.

Ejemplos de lectura de todo el archivo:

```
>>> f = open('entrada.txt', 'r')
>>> f.read()
'Este es el contenido del archivo.\n'
>>> f.read()
''
```

Ejemplos de lectura del archivo en varios trozos:

```
>>> f = open('entrada.txt', 'r')
>>> f.read(4)
'Este'
>>> f.read(4)
' es '
>>> f.read(4)
'el c'
>>> f.read()
'ontenido del archivo\n'
>>> f.read()
''
```

3.2. `readline`

El método `readline` también sirve para leer de un archivo y también se ejecuta sobre el objeto que devuelve `open`.

Su signatura es:

```
⟨archivo⟩.readline([tamaño: int]) -> str | bytes
```

`readline` devuelve una línea del archivo en forma de cadena (si el archivo se abrió en **modo texto**) o un valor de tipo `bytes` (si se abrió en **modo binario**), dejando el carácter de salto de línea (`\n`) al final.

El salto de línea sólo se omite cuando es la última línea del archivo y éste no acaba en salto de línea.

Si devuelve una cadena vacía (`''`), significa que se ha alcanzado el final del archivo.

Si se devuelve una cadena formada sólo por `\n`, significa que es una línea en blanco (una línea que sólo contiene un salto de línea).

El método empieza a leer desde la posición actual del puntero interno del archivo y cambia la posición del mismo.

El parámetro *tamaño* es opcional:

- Si se omite o es negativo, se devuelve todo desde la posición actual del puntero hasta el final de la línea.
- En caso contrario, se leerán y devolverán *como mucho* tantos caracteres (en modo texto) o bytes (en modo binario) como se haya indicado.

Ejemplos

```
>>> f = open('entrada.txt', 'r')
>>> f.readline()
'Esta es la primera línea.\n'
>>> f.readline()
'Esta es la segunda.\n'
>>> f.readline()
'Y esta es la tercera.\n'
>>> f.readline()
''
>>> f = open('entrada.txt', 'r')
>>> f.readline(4)
'Esta'
>>> f.readline(4)
' es '
>>> f.readline()
'la primera línea.\n'
>>> f.readline()
'Esta es la segunda.\n'
>>> f.readline()
'Y esta es la tercera.\n'
>>> f.readline()
''
```

3.3. readlines

El método `readlines` (en plural, no confundir con `readline` en singular) también sirve para leer de un archivo y también se ejecuta sobre el objeto que devuelve `open`, pero en lugar de su versión en singular, **lee *varias* líneas del archivo de una sola vez**.

Su signatura es:

```
⟨archivo⟩.readlines([tamaño: int]) -> list[str|bytes]
```

`readlines` devuelve una lista de cadenas de caracteres o de bytes (según como se haya abierto el archivo, en modo texto o binario).

El método empieza a leer desde la posición actual del puntero interno del archivo y cambia la posición del mismo.

Las líneas conservan el carácter de salto de línea (`\n`) al final. El salto de línea sólo se omite cuando es la última línea del archivo y éste no acaba en salto de línea.

Si devuelve una lista vacía (`[ ]`), significa que el archivo está vacío.

Si uno de los elementos está formada sólo por `\n`, significa que es una línea en blanco (una línea que sólo contiene un salto de línea).

El parámetro *tamaño* es opcional, y se puede usar para controlar la cantidad de líneas leídas:

- Si se omite o es negativo, se leerá desde la posición actual del puntero hasta el final del archivo, devolviendo cada línea separada en un elemento de la lista.
- En caso contrario, se leerán y devolverán el menor número de líneas que sean necesarias para leer el número de caracteres (en modo texto) o bytes (en modo binario) indicado, desde la posición actual del puntero.

Ejemplos

```
>>> f = open('entrada.txt', 'r')
>>> f.readlines()
['Esta es la primera línea.\n', 'Esta es la segunda.\n',
 'Y esta es la tercera.\n']
>>> f.readlines()
[]
>>> f = open('entrada.txt', 'r')
>>> f.readlines(5)
['Esta es la primera línea.\n']
>>> f = open('entrada.txt', 'r')
>>> f.read(4)
'Esta'
>>> f.readlines()
[' es la primera línea.\n', 'Esta es la segunda.\n',
 'Y esta es la tercera.\n']
```

3.4. Archivos como iterables

Existen iterables e iteradores incluso donde uno menos se lo podría esperar.

Por ejemplo, **los archivos abiertos también son iterables**, ya que se pueden recorrer línea a línea usando un iterador:

```
with open('archivo.txt') as f:
    for linea in f:
        print(linea)
```

Esta forma de recorrer los archivos, además de resultar simple y elegante, también resulta muy eficiente, ya que se va recuperando cada línea de una en una en lugar de todas a la vez.

4. Operaciones de escritura

4.1. `write`

El método `write` sirve para escribir en un archivo y se ejecuta sobre el objeto que devuelve `open` (y que representa al archivo abierto).

Su signatura es:

```
⟨archivo⟩.write(contenido: str | bytes) -> int
```

El método escribe el *contenido* en el ⟨*archivo*⟩. Ese contenido debe ser una *cadena* de caracteres si el archivo se abrió en **modo texto**, o un *valor de tipo* `bytes` si se abrió en **modo binario**.

Al escribir, modifica el puntero interno del archivo.

Devuelve el número de caracteres o de bytes que se han escrito, dependiendo de si se abrió en modo texto o en modo binario.

También se puede usar `print` para escribir en un archivo.

En la práctica, no hay mucha diferencia entre usar `print` y usar `write`.

Hacer:

```
>>> f = open('archivo.txt', 'r+')
>>> f.write('Hola Manolo\n')
```

equivale a hacer:

```
>>> f = open('archivo.txt', 'r+')
>>> print('Hola', 'Manolo', file=f)
```

Hay que tener en cuenta los separadores y los saltos de línea que introduce `print`.

`write` NO escribe el carácter de salto de línea al final, cosa que sí hace `print` (salvo que le digamos lo contrario).

`print` escribe en el flujo `sys.stdout` mientras no se diga lo contrario.

Si el archivo se ha abierto en un modo `'a'` o `'a+'`, el puntero empezará estando al **final del archivo** y la escritura **siempre** se realizará a partir de ese punto, aunque previamente hayamos leído algo o hayamos movido el puntero explícitamente mediante `seek`:

```
>>> f = open('entrada.txt', 'r')   # Abrimos el archivo para ver su contenido
>>> f.readlines()                  # Comprobamos que contiene las líneas originales
['Esta es la primera línea.\n',
 'Esta es la segunda.\n',
 'Y esta es la tercera.\n']
>>> f.close()
>>> f = open('entrada.txt', 'a+')  # Lo volvemos a abrir en lectura/añadir
>>> f.read(4)                      # El puntero está situado al final, y...
''                                 # ... si leemos algo, allí no hay nada
```

```
>>> f.write('Prueba\n')            # Escribimos siete caracteres al final
7
>>> f.close()                      # Cerramos el archivo para guardar los cambios
>>> f = open('entrada.txt', 'r')   # Volvemos a abrir el archivo
>>> f.readlines()                  # Comprobamos que se ha escrito al final...
['Esta es la primera línea.\n',    # ... del archivo, y no en la posición 4...
 'Esta es la segunda.\n',          # ... como cabría haber esperado
 'Y esta es la tercera.\n',
 'Prueba\n']
```

Si el archivo se ha abierto en un modo `'w'`, `'w+'` o `'r+'`, el puntero empezará estando al **principio del archivo** y la escritura se realizará en la posición del puntero.

Pero **si leemos algo del archivo antes de escribir en él, la escritura se hará *al final*** del archivo (como si lo hubiésemos abierto con un modo `'a+'`) a menos que primero movamos el puntero explícitamente mediante `seek`:

```
>>> f = open('entrada.txt', 'r')   # Abrimos el archivo para ver su contenido
>>> f.readlines()                  # Comprobamos que contiene las líneas originales
['Esta es la primera línea.\n',
 'Esta es la segunda.\n',
 'Y esta es la tercera.\n']
>>> f.close()
>>> f = open('entrada.txt', 'r+')  # Lo volvemos a abrir en lectura/escritura
>>> f.read(4)                      # Leemos cuatro caracteres (desde el principio)
'Esta'
>>> f.write('Prueba\n')            # Escribimos siete caracteres
7
>>> f.close()                      # Cerramos el archivo para guardar los cambios
>>> f = open('entrada.txt', 'r')   # Volvemos a abrir el archivo
>>> f.readlines()                  # Comprobamos que se ha escrito al final...
['Esta es la primera línea.\n',    # ... del archivo, y no en la posición 4...
 'Esta es la segunda.\n',          # ... como cabría haber esperado
 'Y esta es la tercera.\n',
 'Prueba\n']
```

4.2. `writelines`

El método `writelines` escribe una lista de líneas en un archivo, por lo que, en cierta forma, es el contrario de `readlines`.

Igualmente, se ejecuta sobre el objeto que devuelve `open` (y que representa al archivo abierto).

Su signatura es:

```
⟨archivo⟩.writelines(lineas: list[str|bytes]) -> None
```

El parámetro *lineas* es el contenido a escribir en el archivo, y debe ser una lista de *cadenas* si el archivo se abrió en **modo texto**, o de *valores de tipo* `bytes` si se abrió en **modo binario**.

Todo lo comentado anteriormente para el método `write` sobre su comportamiento en función del modo de apertura del archivo, se aplica también a `writelines`.

Ejemplos

```
>>> f = open('entrada.txt', 'r')
>>> f.readlines()
['Esta es la primera línea.\n', 'Esta es la segunda.\n',
 'Y esta es la tercera.\n']
>>> f.close()
>>> f = open('salida.txt', 'w')
>>> f.writelines(['Primera línea de salida.txt.\n','Segunda línea.\n'])
>>> f.close()
>>> f = open('salida.txt', 'r')
>>> f.readlines()
['Primera línea de salida.txt.\n','Segunda línea.\n']
```

Al igual que `write`, el método `writelines` NO escribe el **salto de línea** al final de cada cadena, así que tendremos que introducirlo nosotros mismos.

5. Otras operaciones

5.1. seek y tell

El método `seek` **sitúa el puntero interno** del archivo en una determinada posición.

El método `tell` **devuelve la posición actual** del puntero interno.

Sus signaturas son:

```
⟨archivo⟩.seek(offset: int) -> int
```

```
⟨archivo⟩.tell() -> int
```

El *offset* es la posición a la que se desea mover el puntero, empezando por 0 desde el comienzo del archivo.

Además de mover el puntero, el método `seek` devuelve la nueva posición del puntero.

Pero no olvidemos que si el archivo se ha abierto en modo *añadir* (`'a'` o `'a+'`), la escritura se hará siempre al final del archivo, sin importar cuál sea la posición actual del puntero.

Ejemplos

```
>>> f = open('entrada.txt', 'r+')    # Abre en modo lectura/escritura
>>> f.tell()                         # El puntero está al principio
0
>>> f.readline()                     # Lee una línea de texto
'Esta es la primera línea.\n'
>>> f.tell()                         # Se ha movido el puntero
27
>>> f.seek(0)                        # Vuelve a colocarlo al principio
0
>>> f.readline()                     # Por tanto, se lee la misma línea
```

```
'Esta es la primera línea.\n'
>>> f.seek(0)                          # Vuelve a colocarlo al principio
0
>>> f.write('Cambiar')                 # Escribe desde el principio
7
>>> f.tell()
7
>>> f.seek(0)
0
>>> f.readline()                       # Se ha cambiado la primera línea
'Cambiar la primera línea.\n'
```

```
>>> f = open('entrada.txt', 'a+')     # Abre en modo lectura/añadir
>>> f.tell()                          # El puntero está al final
69
>>> f.readline()                      # Allí no hay nada
''
>>> f.seek(0)                         # Movemos el puntero al principio
0
>>> f.tell()                          # El puntero se ha movido
0
>>> f.readline()                      # Por tanto, se lee la primera línea
'Esta es la primera línea.\n'
>>> f.tell()                          # El puntero se ha movido
27
>>> f.seek(0)                         # Vuelve a colocarlo al principio
0
>>> f.write('Prueba\n')               # Siempre se escribe al final
7
>>> f.tell()                          # El puntero está al final
76
>>> f.readlines()                     # Allí no hay nada
[]
>>> f.seek(0)                         # Movemos el puntero al principio
0
>>> f.readlines()                     # Leemos todas las líneas
['Esta es la primera línea.\n',
 'Esta es la segunda.\n',
 'Y esta es la tercera.\n',
 'Prueba\n']                         # Se ha escrito al final
```

Bibliografía

Python Software Foundation. n.d. *Sitio Web de Documentación de Python*. https://docs.python.org/3.