

Python cómo escribir en un archivo - abrir, leer, escribir y otras funciones de archivos explicadas

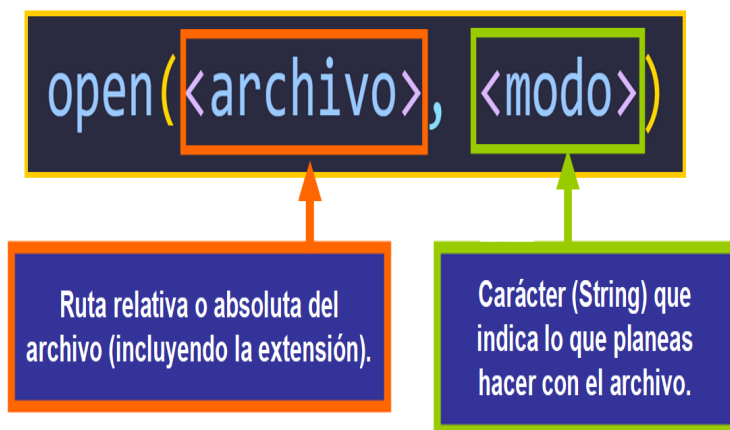
- Cómo abrir un archivo.
- Cómo leer un archivo.
- Cómo crear un archivo.
- Cómo modificar un archivo.
- Cómo cerrar un archivo.
- Cómo abrir archivos para realizar varias operaciones.
- Cómo trabajar con métodos de objetos archivo.
- Cómo eliminar archivos.
- Cómo trabajar con "context managers" (gestores de contexto) y por qué son útiles.
- Cómo manejar excepciones que podrían ocurrir cuando trabajas con archivos.
- ¡Y más!

¡Comencemos! ✨

◆ Trabajar con archivos: sintaxis básica

Una de las funciones más importantes que necesitarás usar a medida que trabajas con archivos en Python es `open()`, una función incorporada (built-in) que abre un archivo y permite que tu programa tenga acceso a él.

Esta es la sintaxis básica:



💡 **Dato:** estos son los dos argumentos más comúnmente usados para llamar a esta función. Existen seis argumentos adicionales que son opcionales. Para aprender más sobre ellos, por favor lee [este artículo](#) en la documentación.

Primer parámetro: file (archivo)

El primer parámetro de la función `open()` es `file` (archivo), la ruta (path) absoluta o relativa del archivo con el cual estás intentando trabajar.

Normalmente usamos la ruta relativa, la cual indica dónde está ubicado el archivo en relación a la ubicación del archivo de Python (script) que llama a la función `open()`.

Por ejemplo, la ruta en esta llamada a la función `open()`:

```
Python
open("nombres.txt") # La ruta relativa es "nombres.txt"
```

Solo contiene el nombre del archivo. Esto puede ser usado cuando el archivo que estás intentando abrir está ubicado en el mismo directorio o carpeta que el script de Python, de esta forma:



code

Type: Python Source File



nombres

Pero si el archivo de texto está dentro de otra carpeta, de esta forma:



datos



code

Type: Python Source File

Entonces necesitamos usar una ruta específica para indicarle a la función que el archivo de texto está dentro de otra carpeta:

Por lo tanto, esta sería la ruta para este ejemplo:

```
Python  
open("datos/nombres.txt")
```

Nota que estamos escribiendo `datos/` al principio (el nombre de la carpeta seguido de un `/`) y luego `nombres.txt` (el nombre del archivo con su extensión).

💡 **Datos:** las tres letras `.txt` ubicadas a la derecha del nombre del archivo en `nombres.txt` es la extensión del archivo (el tipo de archivo). En este caso, `.txt` indica que es un archivo de texto.

Segundo parámetro: mode (modo)

El segundo parámetro de la función `open()` es `mode` (modo), una cadena de caracteres conformada por un solo carácter. Ese único carácter básicamente le dice a Python lo que planeas hacer con el archivo en tu programa.


Los modos disponibles son:

- Read ("**r**") (Leer)
- Append ("**a**") (Agregar)
- Write ("**w**") (Escribir)
- Create ("**x**") (Crear)

También puedes abrir el archivo en:

- Text mode ("**t**") (Modo texto)
- Binary mode ("**b**") (Modo binario)

Para usar los modos de texto o binario, debes añadir estos caracteres al modo principal. Por ejemplo: "**wb**" significa "escribir en modo binario".

 **Dato:** los modos asignados por defecto son read ("**r**") (leer) y text ("**t**") (texto), lo cual significa "abrir para leer texto" ("**rt**"), así que no necesitas especificarlos en `open()` si deseas usarlos porque se asignan automáticamente. Puedes simplemente escribir `open(<archivo>)`.

¿Por qué modos?

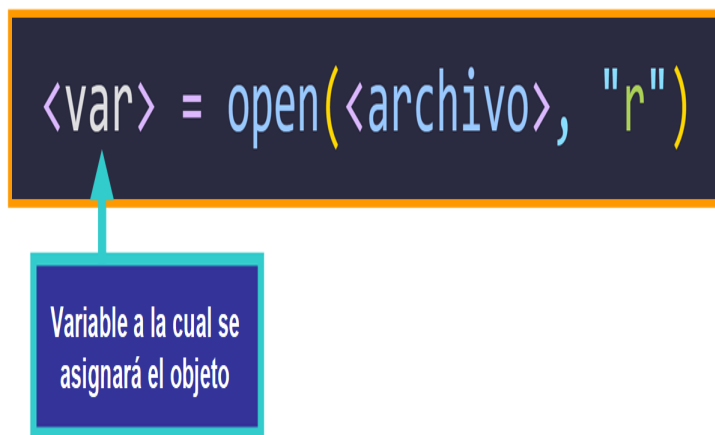
En realidad, tiene sentido que Python solo otorgue ciertos permisos en base a lo que planeas hacer con el archivo. ¿Cierto? ¿Por qué Python debería permitirle a tu programa hacer más de lo necesario? Esta es la razón por la cual los modos existen.

Piénsalo... permitirle a un programa hacer más de lo necesario puede ser problemático. Por ejemplo, si solo necesitas leer el contenido de un archivo, puede ser peligroso permitirle a tu programa modificarlo inesperadamente, lo cual podría introducir "bugs" (errores).

◆ Cómo leer un archivo

Ahora que sabes más sobre los argumentos que recibe la función `open()`, veamos cómo puedes abrir un archivo y guardarlo en una variable para usarlo en tu programa.

Esta es la sintaxis básica:



Estamos asignando el valor retornado a una variable. Por ejemplo:

```
Python
archivo_nombres = open("data/nombres.txt", "r")
```

Sé que te debes estar preguntando: ¿qué tipo de valor retorna `open()`?

La respuesta es.... un objeto archivo.

Hablemos un poco sobre ellos.

Objetos archivo

Según la [documentación de Python](#), un objeto archivo es:

Un objeto que expone una API orientada a archivos (con métodos como `read()` o `write()`) al objeto subyacente.

Esto básicamente nos dice que un objeto archivo es un objeto que nos permite trabajar e interactuar con archivos en nuestros programas de Python.

Los objetos archivo tienen atributos, tales como:

- **name**: el nombre del archivo.
- **closed**: `True` si el archivo está cerrado. `False` si está abierto.
- **mode**: el modo usado para abrir el archivo.



Por ejemplo:

```
Python
f = open("datos/nombres.txt", "a")
print(f.mode) # Resultado: "a"
```

Ahora veamos cómo podemos acceder al contenido de un archivo a través de un objeto archivo.

Métodos para leer un archivo

Para poder trabajar con objetos archivo, necesitamos tener una forma de "interactuar" con ellos en nuestro programa y eso es exactamente lo que hacen los métodos. Veamos algunos de ellos.

Read()

El primer método que debes aprender es `read()`, el cual retorna todo el contenido del archivo como una cadena de caracteres.



Aquí tenemos un ejemplo:

```
Python
f = open("data/nombres.txt")
print(f.read())
```

El resultado es:

```
Python
Nora
Gino
Timmy
William
```

Puedes usar la función `type()` para confirmar que el valor retornado por `f.read()` es una cadena de caracteres:

```
Unset
print(type(f.read()))

# Output
<class 'str'>
```

¡Sí, es una cadena de caracteres!

En este caso, se mostró todo el archivo porque no especificamos un número máximo de bytes, pero también podemos hacerlo:

Aquí tenemos un ejemplo:

```
Python
f = open("data/nombres.txt")
print(f.read(3))
```

El valor retornado se limitará a este número de caracteres:

```
Unset
Nor
```

💡 **Importante:** debes **cerrar** el archivo luego de que la tarea ha sido completada para liberar los recursos asociados al archivo. Para hacerlo, debes llamar al método `close()` de esta forma:

```
<var_objeto>.close()
```

El objeto archivo
que se va a cerrar

Readline() vs. Readlines()

Puedes leer un archivo línea por línea con estos dos métodos. Son ligeramente diferentes, así que veámoslos en detalle.

`readline()` lee una línea del archivo. Mantiene el carácter de salto de línea (`\n`) al final de la cadena de caracteres.

💡 **Dato:** Opcionalmente, puedes pasar el tamaño (size), el número máximo de caracteres que deseas incluir en el resultado.

```
<var_objeto>.readline([tamaño])
```

El objeto archivo
que se va a leer

(Opcional) Cuántos
bytes se leerán

Por ejemplo:


```
Python
f = open("data/nombres.txt")
print(f.readline())
f.close()
```

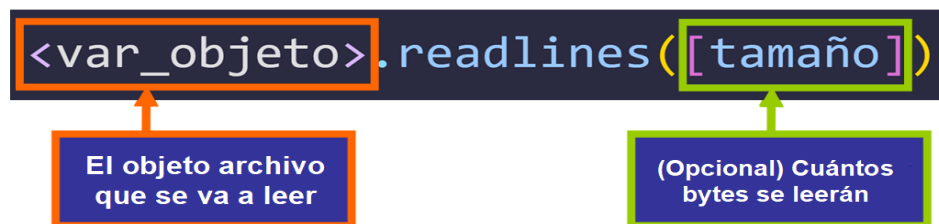
El resultado es:

```
Python
Nora
```

Esta es la primera línea del archivo.

En cambio, `readlines()` retorna una lista que contiene todas las líneas del archivo como elementos individuales de la lista (cadenas de caracteres).

Esta es la sintaxis:



Por ejemplo:

```
Python
f = open("data/nombres.txt")
print(f.readlines())
f.close()
```

El resultado es:

```
Python
['Nora\n', 'Gino\n', 'Timmy\n', 'William']
```

Nota que cada cadena de caracteres termina con un carácter de salto de línea `\n`, excepto la última.



Dato: Puedes obtener la misma lista con `list(f)`.

Puedes trabajar con esta lista en tu programa asignándola a una variable o usando un ciclo:

```
Python
f = open("data/nombres.txt")

for line in f.readlines():
    # Hacer algo con cada línea del archivo.

f.close()
```

También podemos iterar sobre `f` directamente (sobre el objeto archivo) en un ciclo:

```
Python
f = open("data/nombres.txt", "r")

for line in f:
    # Hacer algo con cada línea del archivo.

f.close()
```

Estos son los principales métodos que usamos en Python para leer objetos archivo. Ahora veamos cómo puedes crear archivos.

◆ Cómo crear un archivo

Si necesitas crear un archivo de forma dinámica usando Python, puedes hacerlo con el modo `"x"`.

Veamos cómo. Esta es la sintaxis básica:

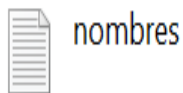
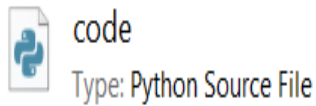
```
<variable> = open("<archivo>.<extensión>", "x")
```



Crear el archivo

Puedes escribir el nombre del archivo para crearlo en el directorio en el que estás actualmente o especificar una ruta para crear el archivo en una carpeta diferente.

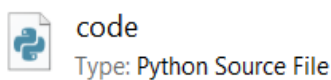
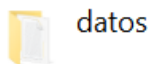
Aquí tenemos un ejemplo. Este es mi directorio actual de trabajo (working directory):



Si ejecuto esta línea de código:

```
Python
f = open("archivo_nuevo.txt", "x")
```

Se crea un archivo con este nombre:



Con este modo, puedes crear un archivo y luego añadir contenido de forma dinámica usando métodos que aprenderás en tan solo un segundo.

💡 **Dato:** El archivo inicialmente estará vacío.

Algo curioso es que si intentas ejecutar esta línea de código nuevamente y ya existe un archivo con ese mismo nombre, verás un error:

```
Python
Traceback (most recent call last):
  File "<path>", line 8, in <module>
    f = open("archivo_nuevo.txt", "x")
FileExistsError: [Errno 17] File exists: 'archivo_nuevo.txt'
```

De acuerdo con la [documentación de Python](#), esta excepción (error durante la ejecución) se:

Genera cuando se intenta crear un archivo o directorio que ya existe.

Texto original en inglés:

Raised when trying to create a file or directory which already exists.

Ahora que ya sabes cómo crear un archivo, veamos cómo puedes modificarlo.

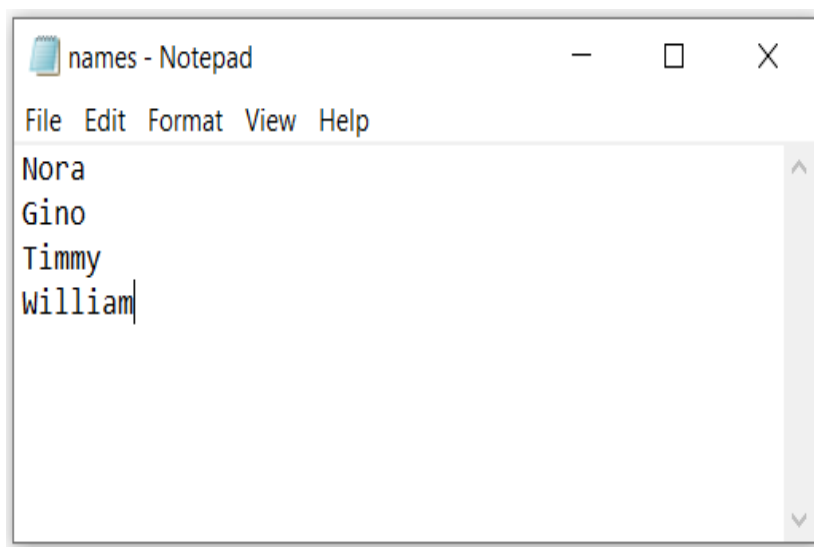
◆ Cómo modificar un archivo

Para modificar (cambiar el contenido) de un archivo, debes usar el método `write()`. Existen dos alternativas (append o write) en base al modo que escojas para abrir el archivo. Veámos estas alternativas en detalle.

Append

"Append" significa agregar algo al final de una estructura o valor. El modo `"a"` (append) te permite abrir un archivo para agregar contenido al final del contenido existente.

Por ejemplo, si tenemos este archivo:



Y queremos agregarle una línea nueva, podemos abrir el archivo usando el modo "a" (append) y luego llamar al método `write()` pasando el contenido que queremos agregar como argumento.

Esta es la sintaxis básica para llamar al método `write()`:

```
<var_objeto>.write("Cadena de caracteres")
```

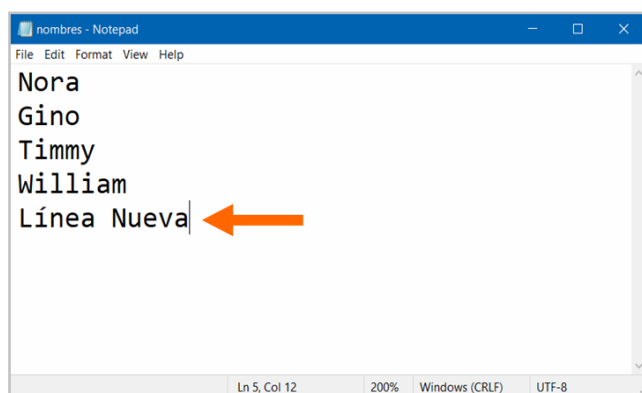
Contenido que deseas
agregar al archivo

Aquí tenemos un ejemplo:

```
Python
f = open("datos/nombres.txt", "a")
f.write("\nLínea Nueva")
f.close()
```

💡 **Dato:** Nota que estoy agregando `\n` antes de la línea para indicar que mi intención es que la línea nueva se agregue como una línea separada y no como una continuación de la línea actual.

Este es el archivo luego de ejecutar el programa:



💡 **Dato:** Es posible que la línea nueva no aparezca en el archivo hasta que `f.close()` se ejecute.

Write

También puedes eliminar todo el contenido de un archivo y reemplazarlo completamente con contenido nuevo. Puedes hacerlo con el método `write()` si abres el archivo en el modo `"w"` (write).

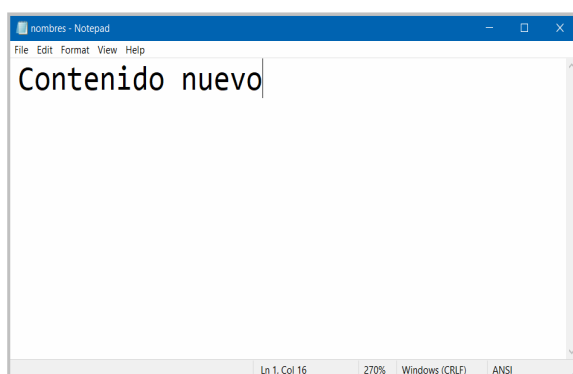
Aquí tenemos este archivo de texto:



Si ejecuto este programa:

```
Python
f = open("datos/nombres.txt", "w")
f.write("Contenido nuevo")
f.close()
```

Este es el resultado:



Como puedes ver, abrir el archivo con el modo `"w"` y luego llamar al método `write()` reemplaza el contenido del archivo.

💡 **Dato:** El método `write()` retorna el número de caracteres que fueron agregados al archivo.

Si deseas escribir varias líneas a la vez, puedes llamar al método `writelines()`, el cual toma una lista de cadenas de caracteres como argumentos. Cada cadena de caracteres representa una línea que se agregará al archivo.

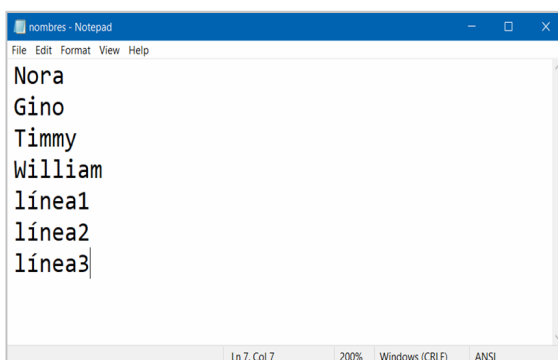
Aquí tenemos un ejemplo. Este es el estado inicial del archivo:



Si ejecutamos este programa:

```
Python
f = open("datos/nombres.txt", "a")
f.writelines(["\nlínea1", "\nlínea2", "\nlínea3"])
f.close()
```

Las líneas se agregan al final del archivo:



Abrir un archivo para varias operaciones

Ahora ya sabes cómo crear, leer y agregar contenido a un archivo, pero ¿cómo puedes realizar más de una operación con el mismo archivo en el mismo programa? Veamos qué ocurre si intentamos hacerlo con los modos que hemos aprendido hasta el momento.

Si abres un archivo en modo `"r"` (leer) y luego tratas de modificarlo:

```
Python
f = open("datos/nombres.txt")
f.write("Contenido Nuevo") # Intentar escribir
f.close()
```

Verás este error:

```
Python
Traceback (most recent call last):
  File "<path>", line 9, in <module>
    f.write("Contenido Nuevo")
io.UnsupportedOperation: not writable
```

De igual forma, si abres un archivo en modo `"w"` mode (escribir) y luego intentas leer su contenido:

```
Python
f = open("datos/nombres.txt", "w")
print(f.readlines()) # Intentar leer
f.write("Contenido Nuevo")
f.close()
```

Verás el siguiente error:

```
Python
Traceback (most recent call last):
  File "<path>", line 14, in <module>
    print(f.readlines())
io.UnsupportedOperation: not readable
```

Lo mismo ocurrirá con el modo `"a"` (append).

¿Cómo podemos solucionar este problema?

Para leer el archivo y realizar otras operaciones en el mismo programa, debemos agregar el símbolo "+" al modo, de esta forma:

```
Python
f = open("datos/nombres.txt", "w+") # Leer + Escribir
```

```
Python
f = open("datos/nombres.txt", "a+") # Leer + Append (agregar al final)
```

```
Python
f = open("datos/nombres.txt", "r+") # Leer + Escribir
```

Muy útil ¿cierto? Esto es lo que normalmente necesitarás usar en tus programas, pero asegúrate de incluir solo los modos que necesitas para evitar problemas o errores (bugs).

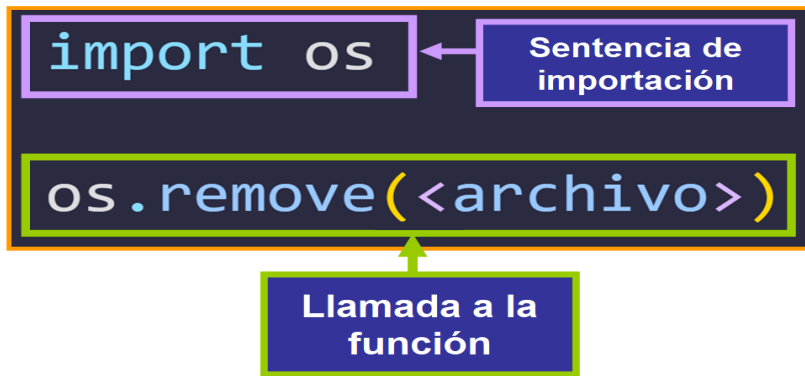
A veces ciertos archivos ya no son necesarios. Veamos cómo puedes eliminar archivos con Python.

◆ Cómo eliminar archivos

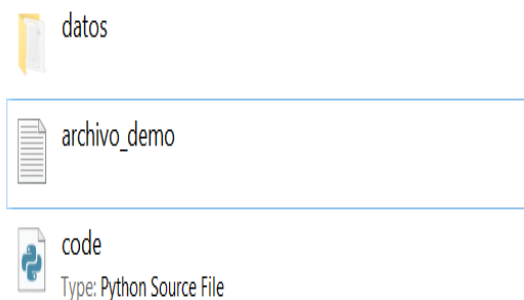
Para eliminar un archivo usando Python, debes importar un módulo llamado `os` que contiene funciones para interactuar con tu sistema operativo.

💡 **Dato:** Un módulo es un archivo de Python con variables, funciones y clases relacionadas.

Específicamente, necesitas la función `remove()`. Esta función toma la ubicación o la ruta (path) del archivo como argumento y elimina el archivo automáticamente.



Veamos un ejemplo. Queremos eliminar el archivo llamado `archivo_demo.txt`.



Para hacerlo, escribimos el siguiente código:

```
Python
import os
os.remove("archivo_demo.txt")
```

- La primera línea: `import os` se denomina una "sentencia de importación".
Esta sentencia se escribe al inicio del archivo y te permite tener acceso a las funciones y otros elementos definidos en el módulo `os`.
- La segunda línea: `os.remove("archivo_demo.txt")` elimina el archivo especificado por el argumento.

💡 **Dato:** Puedes usar una ruta (path) absoluta o relativa.

Ahora que ya sabes cómo eliminar archivos, veamos una herramienta interesante: los Gestores de Contexto (Context Managers).

◆ Gestores de contexto

Los gestores de contexto son estructuras en Python que te ayudarán muchísimo al trabajar con archivos. Si usas gestores de contexto, no necesitarás recordar cerrar el archivo al final de tu programa y tendrás acceso al archivo solamente en la parte específica del programa que escojas, lo cual disminuye el riesgo de errores (bugs).

Sintaxis

Este es un ejemplo de un gestor de contexto usado para trabajar con archivos:



💡 **Dato:** El cuerpo de un gestor de contexto debe estar indentado, al igual que indentamos el cuerpo de los ciclos, de las funciones y de las clases. Si el código no está indentado, no se considerará parte del gestor de contexto.

El archivo se cierra automáticamente cuando se completa la ejecución del cuerpo del gestor de contexto:

```
Python
with open("<ruta del archivo>", "<modo>") as <variable>:
    # Trabajando con el archivo...

# ¡El archivo está cerrado aquí!
```

Ejemplo

Aquí tenemos un ejemplo:

Python

```
with open("datos/nombres.txt", "r+") as f:  
    print(f.readlines())
```

Este gestor de contexto abre el archivo `nombres.txt` para operaciones de lectura y escritura y asigna el objeto archivo a la variable `f`. Esta variable se usa en el cuerpo del gestor de contexto para trabajar con el objeto archivo.

Intentar leerlo nuevamente

Luego de haber completado la ejecución del cuerpo del gestor de contexto, el archivo se cierra automáticamente, así que no puede ser leído si no se abre nuevamente.

Aquí tenemos una línea de código que intenta leer el archivo luego de cerrarlo:

Python

```
with open("datos/nombres.txt", "r+") as f:  
    print(f.readlines())  
  
print(f.readlines()) # Intentar leer el archivo nuevamente, fuera del gestor de contexto.
```

Veámos qué ocurre:

Python

```
Traceback (most recent call last):  
  File "<path>", line 21, in <module>  
    print(f.readlines())  
ValueError: I/O operation on closed file.
```

Se genera este error porque estamos intentando leer un archivo que ya fue cerrado.

Increíble, ¿cierto?

El gestor de contexto hace todo el trabajo pesado por nosotros, es fácil de leer y muy conciso.

◆ Cómo manejar excepciones al trabajar con archivos

Cuando trabajas con archivos pueden ocurrir errores. Quizás porque no tendremos los permisos necesarios para modificar o leer un archivo, o quizás porque el archivo no existe.

Como programadores, necesitamos predecir estas posibles circunstancias y manejarlas en el programa para evitar cierres o problemas inesperados que pueden afectar la experiencia del usuario.

Veamos algunas de las excepciones (errores durante la ejecución del programa) más comunes que puedes encontrar al trabajar con archivos:

FileNotFoundError

De acuerdo con la [documentación de Python](#), esta excepción:

Ocurre cuando un archivo o directorio es solicitado pero no existe.

Texto original en inglés:

Raised when a file or directory is requested but doesn't exist.

Por ejemplo, si el archivo que intentas abrir no existe en tu directorio actual de trabajo (current working directory):


```
Python
f = open("nombres.txt")
```

Verás el siguiente error:

```
Python
Traceback (most recent call last):
  File "<ruta>", line 8, in <módulo>
    f = open("nombres.txt")
FileNotFoundError: [Errno 2] No such file or directory: 'nombres.txt'
```

Veamos este error línea por línea:

- `File "<ruta>", line 8, in <módulo>.` Esta línea te dice que el error fue generado cuando el código en el archivo ubicado en `<ruta>` se estaba ejecutando. Específicamente, cuando la línea 8 estaba siendo ejecutada en `<módulo>`.
- `f = open("nombres.txt").` Esta es la línea que generó el error.
- `FileNotFoundError: [Errno 2] No such file or directory: 'nombres.txt' .` Esta línea dice que ocurrió una excepción `FileNotFoundError` porque el archivo o directorio `nombres.txt` no existe.

 **Dato:** Los mensajes de error son muy descriptivos en Python ¿cierto? Esta es una ventaja buenísima durante el proceso de "debugging" o depuración (el proceso para identificar errores o "bugs" en nuestro programa).

PermissionError

Esta es otra excepción común al trabajar con archivos. Según la [documentación oficial de Python](#), esta excepción:

Ocurre cuando se intenta ejecutar una operación sin los permisos de acceso adecuados - por ejemplo, permisos del sistema de archivos.

Texto original en inglés:

Raised when trying to run an operation without the adequate access rights - for example filesystem permissions.

La excepción ocurre cuando intentas leer o modificar un archivo pero no tienes los permisos necesarios para acceder a ese archivo.

Si intentas hacerlo, verás el siguiente error:

```
Python
Traceback (most recent call last):
  File "<ruta>", line 8, in <module>
    f = open("<ruta del archivo>")
PermissionError: [Errno 13] Permission denied: 'datos'
```

IsADirectoryError

Según la [documentación oficial de Python](#), esta excepción:

Ocurre cuando una operación de archivos se intenta ejecutar en un directorio.

Texto original en inglés:

Raised when a file operation is requested on a directory.

Esta excepción en particular ocurre cuando intentas abrir o trabajar con un directorio en lugar de un archivo, así que debes tener mucho cuidado al momento de escoger la ruta que pasarás como argumento a los métodos de objetos archivo.

Cómo manejar excepciones

Para manejar estas excepciones, puedes usar una sentencia **try/except**. Con esta sentencia, puedes indicarle a tu programa qué hacer en caso de que algo ocurra.

Esta es la sintaxis básica:


```
Unset
try:
    # Intenta ejecutar este código.
except <tipo_de_excepción>:
    # Si ocurre una excepción de este tipo, detén el proceso inmediatamente y salta a este
    bloque de código.
```

Aquí puedes ver un ejemplo con `FileNotFoundError`:

```
Python
try:
    f = open("nombres.txt")
except FileNotFoundError:
    print("El archivo no existe.")
```

Este código básicamente dice:

- Intenta abrir el archivo `nombres.txt`.
- Si una excepción `FileNotFoundError` ocurre, ¡no colapses! Solo muéstrale al usuario un mensaje describiendo lo que ocurrió.

 **Dato:** Puedes escoger cómo manejar la situación escribiendo el código apropiado en el bloque `except`. Quizás puedas crear un archivo nuevo si no existe en la ruta indicada.

Para cerrar el archivo automáticamente luego de la tarea (independientemente de si ocurrió una excepción o no en el bloque `try`) puedes agregar la cláusula `finally`.

```
Unset
try:
    # Intenta ejecutar este código.
except <tipo_de_excepción>:
    # Si ocurre una excepción de este tipo, detén el proceso inmediatamente y salta a este
    bloque de código.
finally:
    # Haz esto luego de ejecutar el código, incluso si ocurrió una excepción.
```

Este es un ejemplo:

```
Python
try:
    f = open("nombres.txt")
except FileNotFoundError:
    print("El archivo no existe.")
finally:
    f.close()
```


Hay muchas formas de personalizar la sentencia `try/except/finally` e incluso puedes añadir una cláusula `else` para ejecutar un bloque de código solamente si no ocurren excepciones en el bloque `try`.

◆ En Resumen

- Puedes crear, leer, escribir y eliminar archivos usando Python.
- Los objetos archivo tienen su propio conjunto de métodos que podemos usar para trabajar con ellos en nuestros programas.
- Los gestores de contexto te ayudan a trabajar con archivos. Una vez que se completa la ejecución del cuerpo de un gestor de contexto, el archivo correspondiente se cierra automáticamente.
- El manejo de excepciones es clave en Python. Excepciones comunes cuando trabajas con archivos incluyen `FileNotFoundError`, `PermissionError` and `IsADirectoryError`. Pueden detectarse y manejarse con la sentencia `try/except/else/finally`.

autor: **Estefania Cassingena Navone**

<https://www.freecodecamp.org/espanol/news/python-como-escribir-en-un-archivo-abrir-leer-escribir-y-otras-funciones-de-archivos-explicadas/>