Entrada y Salida

Archivos

Cuando un programa se está ejecutando los datos están en la memoria, pero cuando el programa termina los datos se pierden.

Para almacenar los datos de forma permanente se hace uso de **archivos**. Cada archivo se identifica con un nombre único dentro de directorio o carpeta en que se encuentre. Por ejemplo dentro la carpeta *Documentos* puede existir solo un archivo con el nombre *Apuntes.txt*.

Los archivos se utilizan para organizar los datos e intercambiarlos para distintos fines. El modo de trabajar con archivos es como trabajar con libros, se pueden abrir, leer, escribir y cerrar. Además se puede leer en orden o secuencialmente o yendo a un lugar específico.

> Nota
>
> Toda la organización de las computadoras está basada en archivos y directorios.

Abriendo un Archivo

En python para abrir un archivo utilizamos la función `open`

```
ruta_archivo = "alumnos.txt"
archivo = open(ruta_archivo)
```

Esta función intentara abrir el archivo "alumnos.txt" y si tiene éxito en la variable archivo quedara un tipo de dato que nos á manipularlo.

Leyendo un Archivo

La operación más frecuente con los archivos es leerlos de forma secuencial

```
archivo = open(ruta_archivo)
línea = archivo.readline()

while línea != '':
  # hacer algo con la línea
  línea = archivo.readline()

archivo.close()
```

Este último bloque de código lee todas las líneas (renglones) del archivo hasta que no queden más.

La variable `archivo`, que mencionamos más arriba como un "tipo de dato que nos permitirá manipularlo" guarda cuál es la siguiente posición que debe leer y cuando se ejecuta `archivo.readline()` lee esa posición y avanza una posición más.

La función `close()` cierra el archivo, esta operación es importante para mantener la consistencia de la información. Volveremos más adelante sobre este tema.

Ejemplo "poema.txt"

```
En un lugar de la Mancha,
de cuyo nombre no quiero acordarme,
no ha mucho tiempo que vivía un hidalgo
de los de lanza en astillero.
```

En el ejemplo anterior leímos el archivo línea por línea, pero existe otra forma de leer un archivo. Veamos otro ejemplo.

```
archivo = open(ruta_archivo)
líneas = archivo.readlines()
archivo.close()

for línea in líneas:
  # hacer algo con la línea
  print(línea)
```

```
línea número 0
línea número 1
línea número 2
línea número 3
línea número 4
```

¿Que diferencias hay entre el ejemplo de más arriba y éste?

La diferencia principal y que condiciona el resto de los cambios es que en lugar de leer línea por línea utilizamos la función `readlines()`. Esta función lee *todo* el contenido del archivo y devuelve una lista donde cada elemento de la lista es un renglón. Por otro lado se llama a la función `close()` inmediatamente después de leer todo el archivo. ¿Por qué? ¿Te animás a analizar todas las diferencias?

Resumen

- `read()`: Lee todo el archivo y lo devuelve como una cadena de texto. El archivo se guarda en memoria, por lo que puede resultar pesado leer todo el archivo con `read()` si el mismo es muy largo.
- `readline()`: Lee una línea del archivo y la devuelve como una cadena de texto. Cuando se llega al final del archivo, devuelve una cadena vacía.
- `readlines()`: Lee todas las líneas del archivo y las devuelve como una lista de cadenas de texto. Las líneas se guardan en memoria, por lo que puede resultar pesado leer todas las líneas juntas si el mismo el archivo es muy largo.

Tip

Decidir usar read, readline o readlines dependerá de nuestro archivo, su tamaño y la información que contenga. Debemos analizar cada caso para decidir qué método de lectura queremos usar.

Escribiendo en un archivo

Python también tiene métodos para escribir archivos, los más comunes son:

- `write()`: Escribe una cadena de texto en el archivo.
- `writelines()`: Escribe una lista de cadenas de texto en el archivo.

```
ruta_archivo_nuevo = "saludo.txt"
archivo = open(ruta_archivo_nuevo, 'w')

archivo.write("Hola!\n")
archivo.writelines(["¿Cómo estás?\n", "Espero que bien.\n"])
archivo.close()
```

Tip

En este ejemplo se puede ver el uso de `\n`. Este caracter es lo que indica a los medios de salida de información que lo que se escribió finaliza con una nueva línea. Ninguno de los métodos de escritura agrega automáticamente un salto de línea al final de lo que se escribe, a menos que se lo indiquemos explícitamente.
Cuando leemos un archivo tenemos que tener en cuenta que el último caracter de cada línea va a ser `\n`

Pero no todos los archivos pueden ser escritos, por ejemplo los archivos que se encuentran en modo lectura (‘r’). ¿De qué depende? Depende del tipo de acceso con el que se abrió el archivo.

Tipos de acceso

Cuando se abre un archivo hay que especificar para qué lo estamos abriendo, las opciones en general son: leer o escribir. Por defecto, si no especificamos nada, tal como vimos en los ejemplos anteriores, se abre para leer.

operación/modo	r	w	a	r+	w+
leer	si	no	no	si	si
escribir	no	si	si	si	si
posición inicial	inicio	inicio	fin	inicio	inicio
observaciones	Si el archivo no existe da error	Si el archivo existe borra el contenido, sino lo crea	Si el archivo no existe lo crea	Si el archivo no existe da error	Si el archivo existe borra el contenido, sino lo crea
caso de uso	Leer un archivo	Iniciar un nuevo archivo	Agregar más lineas a un archivo existente	Agregar, editar y leer	Agregar, editar y leer

Figura 1: Resumen de los tipos de acceso con los que se puede abrir un archivo.

Veamos ejemplos de los casos más comunes

Ejemplo Write (w)

```
ruta_archivo_nuevo = "alumnos_nuevo.txt"
archivo = open(ruta_archivo_nuevo, 'w')

for x in range(5):
  # hacer algo con la línea
  archivo.write(f"línea número {x} \n")

archivo.close()
```

Ejemplo Read (r)

```
archivo = open(ruta_archivo_nuevo, 'r')
líneas = archivo.readlines()
archivo.close()

for línea in líneas:
  # hacer algo con la línea
  print(línea)
```

Ejemplo Append (a)

```
archivo = open(ruta_archivo_nuevo, 'a')
archivo.write("línea número 5 \n") # agrega una nueva línea al final del archivo
archivo.close()
```

Close

Al terminar de trabajar con un archivo, es importante cerrarlo, por diversos motivos: en algunos sistemas los archivos sólo pueden ser abiertos de a un programa por la vez; en otros, lo que se haya escrito no se guardará realmente hasta no cerrar el archivo.

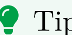
```
archivo = open(ruta_archivo)
líneas = archivo.readlines()
archivo.close()
```

> Advertencia
>
> Cuando abrimos un archivo, queremos dejarlo abierto siempre la **menor cantidad de tiempo posible**. Si abrimos un archivo y no lo cerramos, estamos ocupando recursos del sistema que podrían ser utilizados por otros programas. Por lo que tenemos que pensar muy bien la forma de armar nuestro código para que los archivos se abran y cierren sólo cuando los necesitamos usar.

Una forma de asegurarse de que un archivo se cierre es utilizar la sentencia `with`. Esta sentencia se encarga de cerrar el archivo automáticamente al finalizar el bloque de código que se le pasa.

```
with open(ruta_archivo) as archivo:
  líneas = archivo.readlines()

# Acá el archivo ya se cerró sólo
```

 Tip

¿Te animás a probar que pasa si intentas escribir en un archivo que fue abierto para lectura ('r') y a leer en uno que fue abierto para escritura ('w')?

Ejemplos

Veamos un ejemplo en el que trabajaremos con dos archivos.

> **Ejemplo** Contar la cantidad de palabras de un archivo de texto y guardar el resultado en un nuevo archivo llamado "resultado.txt"

Ejemplo de texto.txt:

```
Hola mundo
Este es un archivo de prueba
con varias líneas de texto
```

```python
# Abrimos el archivo de texto
def contar_palabras(ruta_texto): # La función puede recibir "texto.txt"
  archivo = open(ruta_texto, 'r')
  líneas = archivo.readlines()
  archivo.close()

  # Contamos las palabras de cada línea
  total_palabras = 0
  for línea in líneas:
    palabras = línea.strip('\n').split(" ")
    total_palabras += len(palabras)

  # Guardamos el resultado en un nuevo archivo
  ruta_archivo_resultado = "resultado.txt"
  archivo = open(ruta_archivo_resultado, 'w')
  archivo.write(f"Total de palabras: {total_palabras}")
  archivo.close()
```

Otra forma de resolverlo podría haber sido:

```python
# Abrimos el archivo de texto
def contar_palabras(ruta_texto): # La función puede recibir "texto.txt"
  with open(ruta_texto, 'r') as archivo: # usamos with open
    contenido = archivo.read()

  # Contamos las palabras del contenido completo
  palabras = contenido.split()
  total_palabras = len(palabras)

  # Guardamos el resultado en un nuevo archivo
  ruta_archivo_resultado = "resultado.txt"
  with open(ruta_archivo_resultado, 'w') as archivo_destino:
    archivo_destino.write(f"Total de palabras: {total_palabras}")
```

Veamos el contenido del archivo "resultado.txt"

```python
ruta_archivo = "resultado.txt"
archivo = open(ruta_archivo, 'r')
línea = archivo.readline() # o read
archivo.close()
print(línea)
```

```
Total de palabras: 12
```

En el ejemplo anterior hay al menos dos cosas que vale la pena remarcar: el uso de la función `split()`[1] nos permite separar el texto en una lista de palabras; por otro lado también utilizamos la función `strip()`[2], esto remueve el caracter de nueva línea `\n`.

Otra función muy útil es `read()`, que lee todo el contenido del archivo de una sola vez como un único string.

Tipos de archivos

En la sección anterior utilizamos para todos los archivos la extensión '.txt'. El uso de extensiones es una **convención**, una manera de nombrar las cosas que nos da una idea de lo que hay en el contenido del archivo.

Existen muchos tipos de archivos de texto. Los que vamos a ver en la materia son:

- `.txt`: Archivos de texto plano, sin formato especial.

[1] Split
[2] Strip

- `.csv`: Archivos de "comma separated values" (valores separados por comas). Son muy utilizados para almacenar datos tabulares. Los veremos en detalle en la **Unidad 6** cuando trabajemos con la biblioteca **Pandas**, que nos facilita enormemente su manipulación.

En esta unidad nos enfocamos en archivos de texto plano (`.txt`). Aprender a manipularlos correctamente nos da las bases para entender cómo funcionan todos los demás tipos de archivos de texto.

Conclusiones

- Para utilizar un archivo desde un programa, es necesario abrirlo, y cuando ya no se lo necesite, se lo debe cerrar.
- Las instrucciones más básicas para manejar un archivo son leer y escribir.
- Los archivos sirven para intercambiar información entre diversos programas o entre programas y humanos.
- Los archivos de texto se pueden procesar línea por línea o leyendo la totalidad del archivo y guardándolo en memoria. La elección entre una opción o la otra dependerá del tamaño del archivo. Si el archivo es muy grande, no suele ser eficiente guardarlo en memoria, porque el uso de recursos es muy grande. Otra limitación es la cantidad de veces que abrimos o cerramos el archivo, ya que queremos que sea la menor posible. El manejo de archivos requiere analizar cada caso particular y decidir la forma en la cual interactuaremos con el mismo.

Colab y Archivos

Para usar y crear archivos en Colab, se debe ingresar al último item del menú derecho, que tiene forma de carpeta.

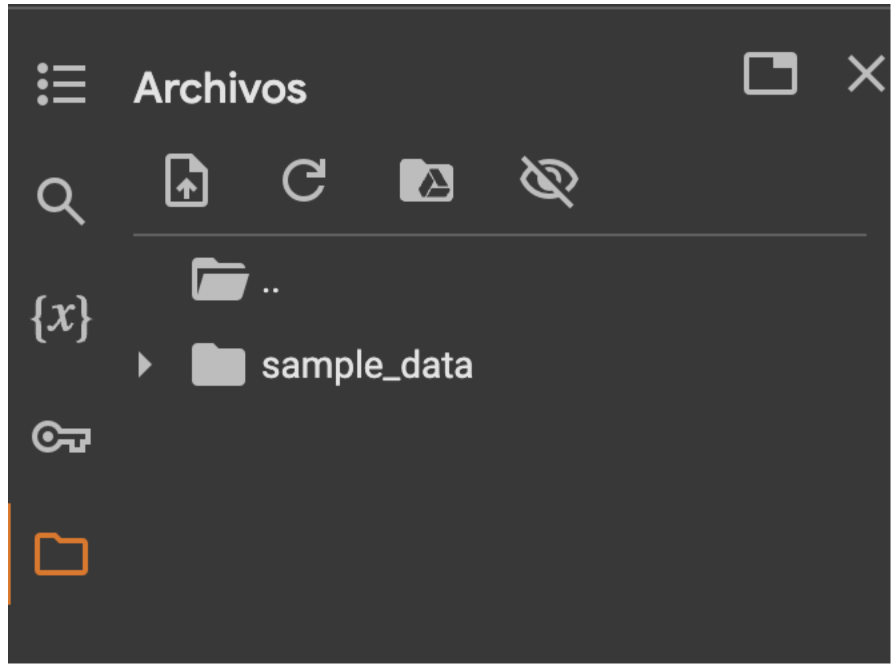


Figura 2: Menu Derecho

Allí, con el menú interno de archivos vamos a poder:

- Subir un archivo que tengamos en la computadora / celular
- Refrescar los archivos para ver el contenido actualizado
- Subir un archivo desde Drive
- Ver archivos ocultos (no vamos a usar esto)

Figura 3: Menú Archivos

También podemos crear un nuevo archivo usando el menú de "sample_data" (los tres puntitos) y seleccionando "Nuevo Archivo".

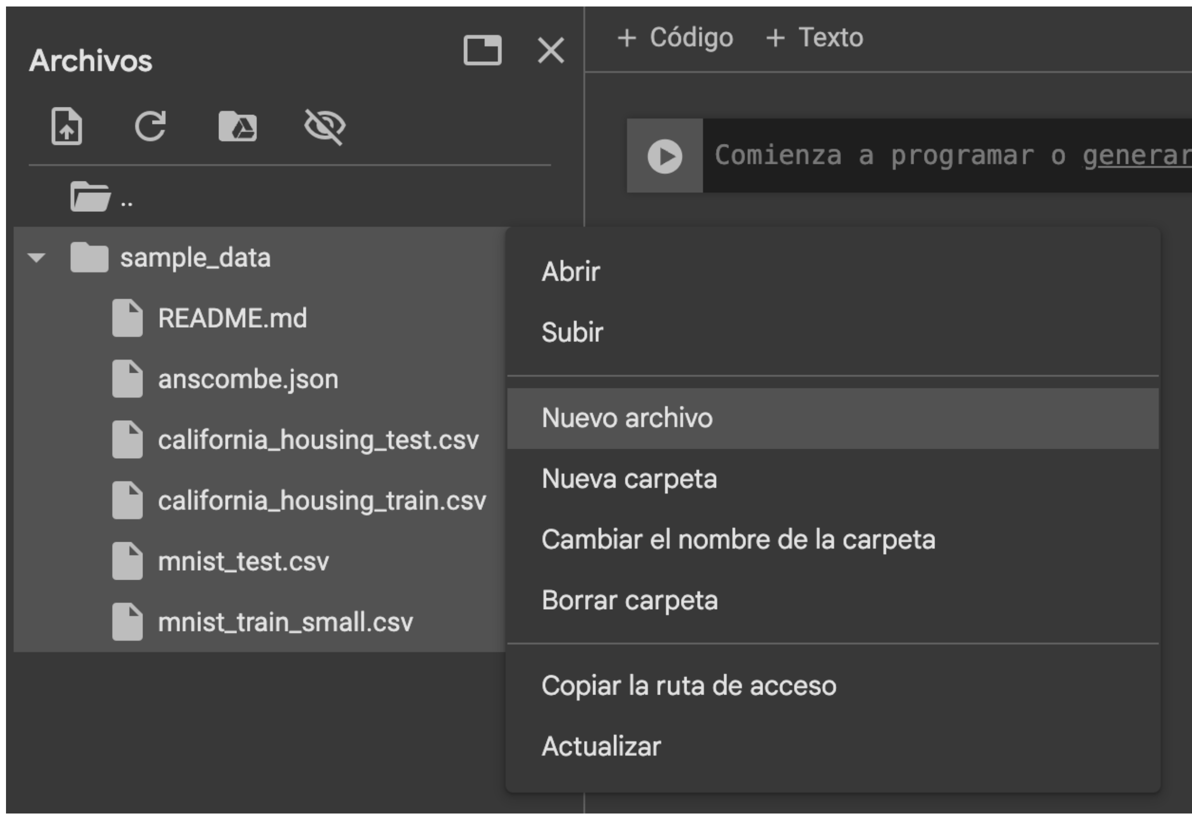
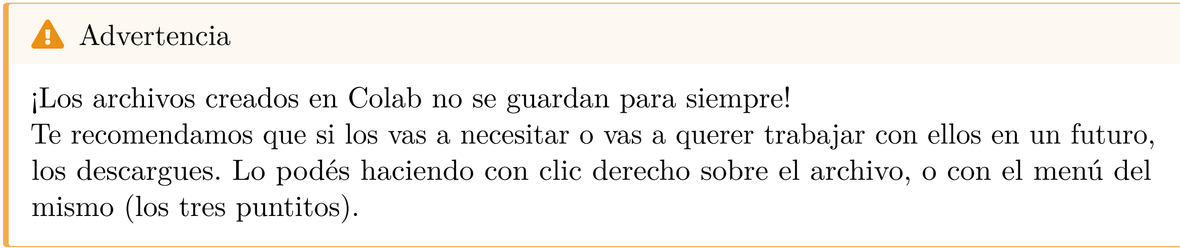


Figura 4: Nuevo archivo en sample data

El archivo luego hay que arrastrarlo *fuera* de la carpeta de sample data para poder usar como ruta su nombre directamente.

De esta forma, podemos abrirlo usando como ruta su nombre: `open("archivo_nuevo")`.

Si hacemos doble clic en el archivo, podemos verlo. Si además el archivo es del tipo txt, podemos editarlo. Los archivos CSV no pueden editarse, pero un truco sería cambiarle la extensión a txt, editarlo, y luego volver a cambiarle la extensión a CSV. No es lo ideal, pero nos permite realizar cambios manuales en un archivo para poder trabajar con mayor comodidad.



⚠ Advertencia

¡Los archivos creados en Colab no se guardan para siempre!
Te recomendamos que si los vas a necesitar o vas a querer trabajar con ellos en un futuro, los descargues. Lo podés haciendo con clic derecho sobre el archivo, o con el menú del mismo (los tres puntitos).

Con estos tips, ya podrías manejarte correctamente para hacer la guía de archivos en Colab.

Manejo de errores

Cuando cometemos un error de tipeo o utilizamos mal una sentencia el intérprete nos muestra un error de sintaxis. En la práctica lo vemos como un `SintaxisError`, este tipo de errores se los llama errores sintácticos, la manera de resolverlo es revisar la sintaxis y corregirlo.

Ejemplo: Función mal definida

```
deff incrementar(n):
  return n + 1
```

```
 File ...., line 1
    deff incrementar(n):
         ^^^^^^^^^^^
SyntaxError: invalid syntax
```

Cuando un programa se está ejecutando y ocurre un error se crea una excepción, normalmente el programa detiene su ejecución y se imprime un mensaje. Este tipo de errores se los llama **errores de ejecución**, vamos a ver como manejarlos.

Ejemplo: División por cero

```
dividendo = 10
divisor = 0
resultado = dividendo/divisor
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
File ...
      1 dividendo = 10
      2 divisor = 0
----> 3 resultado = dividendo/divisor
ZeroDivisionError: division by zero
```

Ejemplo: Acceso a un elemento que no existe

```
lista = ["a","b"]
segundo_elemento = lista[2]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
File /...
      1 lista = ["a","b"]
----> 2 segundo_elemento = lista[2]
IndexError: list index out of range
```

Ejemplo: Abrir un archivo que no existe

```
archivo = open("archivo_falso.txt","r")
```

```
FileNotFoundError                         Traceback (most recent call last)
File ...
----> 1 archivo = open("archivo_falso.txt","r")
FileNotFoundError: [Errno 2] No such file or directory: 'archivo_falso.txt'
```

En cada caso el mensaje de error tiene dos partes, la primera indica el tipo de error:

- `ZeroDivisionError`
- `IndexError`
- `FileNotFoundError`

La segunda tiene una descripción:

- `division by zero`
- `list index out of range`
- `No such file or directory`

Además nos da información contextual que puede indicar en la ejecución de qué línea se dio el error:

- línea 3: `----> 3 resultado = dividendo/divisor`.
- línea 2: `----> 2 segundo_elemento = lista[2]`.
- línea 1: `----> 1 archivo = open("archivo_falso.txt","r")`.

En algunas ocasiones es parte del programa manejar operaciones que puedan lanzar este tipo de excepciones sin que el programa detenga su ejecución, para estos casos Python nos provee las sentencias `try`y `except`.

Ejemplo

```python
dividendo = 10
divisor = 0
try:
  resultado = dividendo/divisor
except ZeroDivisionError:
  print("No se puede dividir por cero.")
```

```
No se puede dividir por cero.
```

Como se ve en el ejemplo se "envuelve" la operación que puede generar ese tipo de excepción para que lo que resulte de esa operación se pueda controlar. Como vimos más arriba hay distintos tipos de excepciones, la lista completa se puede ver en excepciones.

Tip

Es **extremadamente importante** que el bloque **`try` sólo tenga dentro** la porción de código que **tiene posibilidad de romper**. No es correcto colocar todo nuestro código dentro del try, porque si tenemos varios posibles puntos de falla, deberían tratarse por separado. Adicionalmente, es una mala práctica tener demasiado código dentro de un bloque try, cuando es innecesario.

Validaciones

Las validaciones son técnicas que permiten asegurar que los valores con los que se vaya a operar estén dentro de determinado conjunto de posibilidades o que tengan ciertas características.

Si bien quien invoca una función debe preocuparse de cumplir con las precondiciones de ésta, si las validaciones están hechas correctamente pueden devolver información valiosa para que el invocante pueda actuar en consecuencia.

También se debe tener en cuenta qué hará nuestro código cuando una validación falle, ya que queremos darle información al invocante que le sirva para procesar el error. El error producido tiene que ser fácilmente reconocible.

Ejemplo:

```python
def convertir_a_int(ingreso_usuario):
  try:
    return int(ingreso_usuario)
  except ValueError:
    print("El valor ingresado no es un número")
```

```
convertir_a_int("cuarenta")
```

```
El valor ingresado no es un número
```

> **Nota**
>
> En Python también tenemos una forma de *arrojar* nosotros un error, es decir, hacer que la función falle y devuelva un mensaje de error. Esto se hace con la sentencia `raise`. No lo vemos en la materia, pero se puede leer más en la documentación de Python.
> Arrojar o levantar excepciones es un tema complejo, porque además de arrojarlas, debemos ser capaces de capturarlas y manejarlas. Es por esto que no se incluye en el temario de la materia.

Varias Excepciones

Una misma línea podría arrojar varios tipos de excepciones distintos. Existe una forma de atajar varios casos de error sobre un mismo try, pero no lo vemos en la materia. Cada par try-except debería hacer referencia a un sólo tipo de error.

Conclusiones

- El manejo de errores es un parte fundamental en el desarrollo de software, tan importante como la funciónalidad que se está programando.
- Los errores que se dan en tiempo de ejecución los podemos "atrapar" con el bloque `try`.
- Hay tipos de excepciones que describen distintos tipos de errores de ejecución.

Bonus Track: Tipos de Errores

A continuación se presenta una tabla con los errores más comunes que se pueden encontrar al programar en Python.

Tipo de error	Descripción	Más Información
SyntaxError	Error de sintaxis	Suele ser un error de tipeo o de uso incorrecto de una sentencia
ZeroDivisionError	División por cero	Se produce cuando se intenta dividir por cero

Tipo de error	Descripción	Más Información
NameError	Variable no definida	Se produce cuando se intenta utilizar una variable que no fue definida
IndexError	Índice fuera de rango	Se produce cuando se intenta acceder a un elemento de una secuencia, que no existe
FileNotFoundError	Archivo no encontrado	Se produce cuando se intenta abrir un archivo que no existe
TypeError	Tipo de dato incorrecto	Se produce cuando se intenta realizar una operación con un tipo de dato incorrecto
ValueError	Valor incorrecto	Se produce cuando se intenta realizar una operación con un valor incorrecto
KeyError	Clave no encontrada	Se produce cuando se intenta acceder a un elemento de un diccionario que no existe
IOError	Error de entrada/salida	Se produce cuando se intenta realizar una operación de entrada/salida que no se puede realizar (por ejemplo, intentar acceder a un archivo)