# 10 Archivos y excepciones



Ahora que dominas las habilidades básicas necesarias para escribir programas organizados y fáciles de usar, es hora de pensar en hacer que tus programas sean aún más relevantes y utilizables. En este capítulo, aprenderás a trabajar con archivos para que tus programas puedan analizar rápidamente muchos datos.

Aprenderás a gestionar errores para que tus programas no se bloqueen cuando se encuentren con situaciones inesperadas. Aprenderás sobre *exceptions*, que son objetos especiales que Python crea para gestionar los errores que surgen mientras se ejecuta un programa. También conocerás el módulo json, que te permite guardar los datos del usuario para que no se pierdan cuando tu programa deje de ejecutarse.

Aprender a trabajar con archivos y a guardar datos facilitará el uso de tus programas. Los usuarios podrán elegir qué datos introducir y cuándo hacerlo. La gente podrá ejecutar tu programa, hacer algún trabajo, y luego cerrar el programa y continuar donde lo dejaron. Aprender a manejar excepciones te ayudará a enfrentarte a situaciones en las que los archivos no existen y a lidiar con otros problemas que pueden hacer que tus programas se bloqueen. Esto hará que tus programas sean más robustos cuando se encuentren con datos erróneos, tanto si proceden de errores inocentes de como de intentos maliciosos de romper tus programas. Con las habilidades que aprenderás en este capítulo, conseguirás que tus programas sean más aplicables, utilizables y estables.

## Lectura de un archivo

Los archivos de texto contienen una cantidad increíble de datos. Los archivos de texto pueden contener datos meteorológicos, datos de tráfico, datos socioeconómicos, obras literarias y mucho más. La lectura desde un archivo es especialmente útil en aplicaciones de análisis de datos, pero también es aplicable a cualquier situación en la que quieras analizar o modificar información almacenada en un archivo. Por ejemplo, puedes escribir un programa que lea el contenido de un archivo de texto y lo reescriba con un formato que permita visualizarlo en un navegador.

Cuando quieras trabajar con la información de un archivo de texto, el primer paso es leer el archivo en la memoria. A continuación, puedes trabajar con todo el contenido del archivo a la vez o línea por línea.

### Leer el contenido de un fichero

Para empezar, necesitamos un archivo con algunas líneas de texto. Empecemos con un archivo que contenga *pi* con 30 decimales, con 10 decimales por línea:

**pi\_digits.txt**

3.1415926535  
 8979323846  
 2643383279

Para probar tú mismo los siguientes ejemplos, puedes introducir estas líneas en un editor y guardar el archivo como *pi\_digits.txt*, o puedes descargar el archivo de los recursos del libro a través de [https://ehmatthes.github.io/pcc\_3e.](https://ehmatthes.github.io/pcc_3e) Guarda el archivo en el mismo directorio donde guardarás los programas de este capítulo.

Aquí tienes un programa que abre este archivo, lo lee e imprime su contenido en la pantalla:

**file\_reader.py**

from pathlib import Path  
  
❶ path = Path('pi\_digits.txt')  
❷ contents = path.read\_text()  
print(contents)

Para trabajar con el contenido de un archivo, necesitamos indicarle a Python la ruta al archivo. Un *path* es la ubicación exacta de un archivo o carpeta en un sistema. Python proporciona un módulo llamado pathlib que facilita el trabajo con archivos y directorios, independientemente del sistema operativo con el que tú o los usuarios de tu programa estéis trabajando. Un módulo que proporciona una funcionalidad específica como ésta suele denominarse *library*, de ahí el nombre pathlib.

Empezaremos importando la clase Path de pathlib. Hay muchas cosas que puedes hacer con un objeto Path que apunte a un archivo. Por ejemplo, puedes comprobar que el archivo existe antes de trabajar con él, leer su contenido o escribir nuevos datos en él. Aquí, construimos un objeto Path que representa el archivo *pi\_digits.txt*, que asignamos a la variable path ❶. Como este archivo está guardado en el mismo directorio que el archivo *.py* que estamos escribiendo, el nombre del archivo es todo lo que Path necesita para acceder al archivo.

## Nota

VS Code busca los archivos en la carpeta que se abrió más recientemente. Si utilizas VS Code, empieza abriendo la carpeta donde guardas los programas de este capítulo. Por ejemplo, si estás guardando los archivos de tus programas en una carpeta llamada *chapter\_10*, pulsa CTRL-O (⌘-O en macOS), y abre esa carpeta.

Una vez que tenemos un objeto Path que representa a *pi\_digits.txt*, utilizamos el método read\_text() para leer todo el contenido del archivo ❷. El contenido del archivo se devuelve como una única cadena, que asignamos a la variable contents. Cuando imprimimos el valor de contents, vemos todo el contenido del archivo de texto:

3.1415926535  
 8979323846  
 2643383279

La única diferencia entre esta salida y el archivo original es la línea en blanco adicional al final de la salida. La línea en blanco aparece porque read\_text() devuelve una cadena vacía cuando llega al final del archivo; esta cadena vacía aparece como una línea en blanco.

Podemos eliminar la línea en blanco adicional utilizando rstrip() en la cadena contents:

from pathlib import Path  
  
path = Path('pi\_digits.txt')  
contents = path.read\_text()  
contents = contents.rstrip()  
print(contents)

Recuerda del Capítulo 2 que el método rstrip() de Python elimina, o quita, cualquier carácter de espacio en blanco de la parte derecha de una cadena. Ahora la salida coincide exactamente con el contenido del archivo original:

3.1415926535  
 8979323846  
 2643383279

Podemos eliminar el carácter de nueva línea final cuando leamos el contenido del archivo, aplicando el método rstrip() inmediatamente después de llamar a read\_text():

contents = path.read\_text().rstrip()

Esta línea indica a Python que llame al método read\_text() en el archivo con el que estamos trabajando. Luego aplica el método rstrip() a la cadena que devuelve read\_text(). La cadena limpiada se asigna entonces a la variable contents. Este método se denomina *method chaining*, y lo verás utilizado a menudo en programación.

### Rutas de archivos relativas y absolutas

Cuando pasas un nombre de archivo simple como *pi\_digits.txt* a Path, Python busca en el directorio donde está almacenado el archivo que se está ejecutando en ese momento (es decir, tu archivo de programa *.py* ).

A veces, dependiendo de cómo organices tu trabajo, el archivo que quieres abrir no estará en el mismo directorio que tu archivo de programa. Por ejemplo, podrías almacenar tus archivos de programa en una carpeta llamada *python\_work*; dentro de *python\_work*, podrías tener otra carpeta llamada *text\_files* para distinguir tus archivos de programa de los archivos de texto que manipulan. Aunque *text\_files* esté en *python\_work*, pasar a Path el nombre de un archivo en *text\_files* no funcionará, porque Python sólo buscará en *python\_work* y se detendrá ahí; no seguirá y buscará en *text\_files.*. Para conseguir que Python abra archivos de un directorio distinto de aquel en el que está almacenado tu archivo de programa, necesitas proporcionar la ruta correcta.

Hay dos formas principales de especificar rutas en programación. Un *relative file path* le dice a Python que busque una ubicación determinada relativa al directorio donde está almacenado el archivo de programa que se está ejecutando en ese momento. Como *text\_files* está dentro de *python\_work*, necesitamos construir una ruta que empiece por el directorio *text\_files*, y termine por el nombre del archivo. He aquí cómo construir esta ruta:

path = Path('text\_files/filename.txt')

También puedes decirle a Python exactamente dónde está el archivo en tu ordenador, independientemente de dónde esté almacenado el programa que se está ejecutando. Esto se llama *absolute file path*. Puedes utilizar una ruta absoluta si una relativa no funciona. Por ejemplo, si has puesto *text\_files* en alguna carpeta distinta de *python\_work*, entonces pasar a Path la ruta 'text\_files/nombrearchivo.txt' no funcionará porque Python sólo buscará esa ubicación dentro de *python\_work*. Tendrás que escribir una ruta absoluta para aclarar dónde quieres que Python busque.

Las rutas absolutas suelen ser más largas que las relativas, porque empiezan en la carpeta raíz de tu sistema:

path = Path('/home/eric/data\_files/text\_files/filename.txt')

Utilizando rutas absolutas, puedes leer archivos desde cualquier ubicación de tu sistema. Por ahora, lo más fácil es almacenar los archivos en el mismo directorio que los archivos de tu programa, o en una carpeta como *text\_files* dentro del directorio que almacena los archivos de tu programa.

## Nota

Los sistemas Windows utilizan una barra invertida (\) en lugar de una barra inclinada (/) al mostrar las rutas de los archivos, pero debes utilizar barras inclinadas en tu código, incluso en Windows. La biblioteca pathlib utilizará automáticamente la representación correcta de la ruta cuando interactúe con tu sistema o con el de cualquier usuario.

### Acceder a las líneas de un archivo

Cuando trabajes con un archivo, a menudo querrás examinar cada línea del mismo. Puede que busques cierta información en el archivo, o puede que quieras modificar el texto del archivo de alguna manera. Por ejemplo, puede que quieras leer un archivo de datos meteorológicos y trabajar con cualquier línea que incluya la palabra *sunny* en la descripción del tiempo de ese día. En un informe de noticias, podrías buscar cualquier línea con la etiqueta <headline> y reescribir esa línea con un tipo de formato específico.

Puedes utilizar el método splitlines() para convertir una cadena larga en un conjunto de líneas, y luego utilizar un bucle for para examinar cada línea de un archivo, de una en una:

**file\_reader.py**

from pathlib import Path  
  
path = Path('pi\_digits.txt')  
❶ contents = path.read\_text()  
  
❷ lines = contents.splitlines()  
for line in lines:  
 print(line)

Comenzamos leyendo todo el contenido del archivo, como hicimos anteriormente ❶. Si piensas trabajar con las líneas individuales de un archivo, no es necesario que elimines ningún espacio en blanco al leer el archivo. El método splitlines() devuelve una lista de todas las líneas del archivo, y asignamos esta lista a la variable lines ❷. A continuación, hacemos un bucle sobre estas líneas e imprimimos cada una de ellas:

3.1415926535  
 8979323846  
 2643383279

Como no hemos modificado ninguna de las líneas, la salida coincide exactamente con el fichero de texto original.

### Trabajar con el contenido de un archivo

Después de leer el contenido de un archivo en la memoria, puedes hacer lo que quieras con esos datos, así que vamos a explorar brevemente los dígitos de *pi*. Primero, intentaremos construir una única cadena que contenga todos los dígitos del archivo sin espacios en blanco:

**pi\_string.py**

from pathlib import Path  
  
path = Path('pi\_digits.txt')  
contents = path.read\_text()  
  
lines = contents.splitlines()  
pi\_string = ''  
❶ for line in lines:  
 pi\_string += line  
  
print(pi\_string)  
print(len(pi\_string))

Empezamos leyendo el archivo y almacenando cada línea de dígitos en una lista, igual que hicimos en el ejemplo anterior. A continuación, creamos una variable, pi\_string, para guardar los dígitos de *pi*. Escribimos un bucle que añade cada línea de dígitos a pi\_string ❶. Imprimimos esta cadena y también mostramos su longitud:

3.1415926535 8979323846 2643383279  
36

La variable pi\_string contiene los espacios en blanco que había a la izquierda de los dígitos de cada línea, pero podemos deshacernos de ellos utilizando lstrip() en cada línea:

--snip--  
for line in lines:  
 pi\_string += line.lstrip()  
  
print(pi\_string)  
print(len(pi\_string))

Ahora tenemos una cadena que contiene *pi* con 30 decimales. La cadena tiene 32 caracteres porque también incluye el 3 inicial y un punto decimal:

3.141592653589793238462643383279  
32

## Nota

Cuando Python lee de un archivo de texto, interpreta todo el texto del archivo como una cadena. Si lees un número y quieres trabajar con ese valor en un contexto numérico, tendrás que convertirlo en un entero utilizando la función int() o en un flotante utilizando la función float().

### Archivos grandes: Un millón de dígitos

Hasta ahora, nos hemos centrado en analizar un archivo de texto que sólo contiene tres líneas, pero el código de estos ejemplos funcionaría igual de bien en archivos mucho más grandes. Si empezamos con un archivo de texto que contiene *pi* con 1.000.000 de decimales, en lugar de sólo 30, podemos crear una única cadena que contenga todos estos dígitos. No necesitamos cambiar nuestro programa en absoluto, salvo para pasarle un archivo diferente. También imprimiremos sólo los 50 primeros decimales, para no tener que ver pasar un millón de dígitos en el terminal:

**pi\_string.py**

from pathlib import Path  
  
path = Path('pi\_million\_digits.txt')  
contents = path.read\_text()  
  
lines = contents.splitlines()  
pi\_string = ''  
for line in lines:  
 pi\_string += line.lstrip()  
  
print(f"{pi\_string[:52]}...")  
print(len(pi\_string))

La salida muestra que, efectivamente, tenemos una cadena que contiene *pi* con 1.000.000 de decimales:

3.14159265358979323846264338327950288419716939937510...  
1000002

Python no tiene ningún límite inherente a la cantidad de datos con los que puedes trabajar; puedes trabajar con tantos datos como pueda manejar la memoria de tu sistema.

## Nota

Para ejecutar este programa (y muchos de los ejemplos que siguen), necesitarás descargar los recursos disponibles en [https://ehmatthes.github.io/pcc\_3e.](https://ehmatthes.github.io/pcc_3e)

### ¿Tu cumpleaños está contenido en Pi?

Siempre he tenido curiosidad por saber si mi cumpleaños aparece en algún lugar de los dígitos de *pi*. Utilicemos el programa que acabamos de escribir para averiguar si el cumpleaños de alguien aparece en algún lugar del primer millón de dígitos de *pi*. Podemos hacerlo expresando cada cumpleaños como una cadena de dígitos y viendo si esa cadena aparece en algún lugar de pi\_string:

**pi\_birthday.py**

--snip--  
for line in lines:  
 pi\_string += line.strip()  
  
birthday = input("Enter your birthday, in the form mmddyy: ")  
if birthday in pi\_string:  
 print("Your birthday appears in the first million digits of pi!")  
else:  
 print("Your birthday does not appear in the first million digits of pi.")

Primero pedimos la fecha de nacimiento del usuario y luego comprobamos si esa cadena está en pi\_string. Vamos a probarlo:

Enter your birthdate, in the form mmddyy: 120372  
Your birthday appears in the first million digits of pi!

¡Mi cumpleaños sí aparece en los dígitos de *pi*! Una vez que hayas leído un archivo, puedes analizar su contenido de cualquier forma que puedas imaginar.

## Pruébalo tú mismo

10-1. Aprender Python: Abre un archivo en blanco en tu editor de texto y escribe unas líneas resumiendo lo que has aprendido sobre Python hasta ahora. Comienza cada línea con la frase *In Python you can. . .* . Guarda el archivo como *learning\_python.txt* en el mismo directorio que tus ejercicios de este capítulo. Escribe un programa que lea el archivo e imprima lo que has escrito dos veces: imprime el contenido una vez leyendo todo el archivo, y otra almacenando las líneas en una lista y haciendo un bucle sobre cada línea.

10-2.Aprendizaje de C: Puedes utilizar el método replace() para sustituir cualquier palabra de una cadena por otra diferente. Aquí tienes un ejemplo rápido que muestra cómo sustituir 'dog' por 'cat' en una frase:

>>> message = "I really like dogs."  
>>> message.replace('dog', 'cat')  
'I really like cats.'

Lee cada línea del archivo que acabas de crear, *learning\_python.txt*, y sustituye la palabra *Python* por el nombre de otro idioma, como *C*. Imprime cada línea modificada en la pantalla.

10-3. Código más sencillo: El programa *file\_reader.py* de esta sección utiliza una variable temporal, lines, para mostrar cómo funciona splitlines(). Puedes omitir la variable temporal y hacer un bucle directamente sobre la lista que devuelve splitlines():

for line in contents.splitlines():

Elimina la variable temporal de cada uno de los programas de esta sección, para hacerlos más concisos.

## Escribir en un archivo

Una de las formas más sencillas de guardar datos es escribirlos en un archivo. Cuando escribes texto en un archivo, la salida seguirá estando disponible después de que cierres el terminal que contiene la salida de tu programa. Puedes examinar la salida después de que un programa termine de ejecutarse, y también puedes compartir los archivos de salida con otras personas. También puedes escribir programas que vuelvan a leer el texto en la memoria y trabajar con él más tarde.

### Escribir una sola línea

Una vez que tengas definida una ruta, puedes escribir en un archivo utilizando el método write\_text(). Para ver cómo funciona, vamos a escribir un mensaje sencillo y almacenarlo en un archivo en lugar de imprimirlo en la pantalla:

**write\_message.py**

from pathlib import Path  
  
path = Path('programming.txt')  
path.write\_text("I love programming.")

El método write\_text() toma un único argumento: la cadena que quieres escribir en el archivo. Este programa no tiene salida de terminal, pero si abres el archivo *programming.txt*, verás una línea:

**programming.txt**

I love programming.

Este archivo se comporta como cualquier otro archivo de tu ordenador. Puedes abrirlo, escribir nuevo texto en él, copiar de él, pegar en él, etc.

## Nota

Python sólo puede escribir cadenas en un archivo de texto. Si quieres almacenar datos numéricos en un archivo de texto, tendrás que convertirlos primero a formato de cadena utilizando la función str().

### Escribir varias líneas

El método write\_text() hace algunas cosas entre bastidores. Si el archivo al que apunta path no existe, lo crea. Además, después de escribir la cadena en el archivo, se asegura de que el archivo se cierra correctamente. Los archivos que no se cierran correctamente pueden hacer que falten datos o que se corrompan.

Para escribir más de una línea en un archivo, tienes que construir una cadena que contenga todo el contenido del archivo, y luego llamar a write\_text() con esa cadena. Vamos a escribir varias líneas en el archivo *programming.txt*:

from pathlib import Path  
  
contents = "I love programming.\n"  
contents += "I love creating new games.\n"  
contents += "I also love working with data.\n"  
  
path = Path('programming.txt')  
path.write\_text(contents)

Definimos una variable llamada contents que contendrá todo el contenido del archivo. En la siguiente línea, utilizamos el operador += para añadir a esta cadena. Puedes hacer esto tantas veces como necesites, para construir cadenas de cualquier longitud. En este caso incluimos caracteres de nueva línea al final de cada línea, para asegurarnos de que cada declaración aparece en su propia línea.

Si ejecutas esto y luego abres *programming.txt*, verás cada una de estas líneas en el archivo de texto:

I love programming.  
I love creating new games.  
I also love working with data.

También puedes utilizar espacios, caracteres de tabulación y líneas en blanco para dar formato a tu salida, igual que has estado haciendo con la salida basada en terminal. No hay límite para la longitud de tus cadenas, y así es como se crean muchos documentos generados por ordenador.

## Nota

Ten cuidado al llamar a write\_text() sobre un objeto ruta. Si el archivo ya existe, write\_text() borrará el contenido actual del archivo y escribirá nuevo contenido en él. Más adelante en este capítulo, aprenderás a comprobar si un archivo existe utilizando pathlib.

## Pruébalo tú mismo

10-4. Invitado: Escribe un programa que pregunte al usuario su nombre. Cuando responda, escribe su nombre en un archivo llamado *guest.txt*.

10-5. Libro de visitas: Escribe un bucle while que pregunte a los usuarios por su nombre. Recoge todos los nombres introducidos y escríbelos en un archivo llamado *guest\_book.txt*. Asegúrate de que cada entrada aparece en una nueva línea del archivo.

## Excepciones

Python utiliza objetos especiales llamados *exceptions* para gestionar los errores que surgen durante la ejecución de un programa. Cada vez que se produce un error que hace que Python no sepa qué hacer a continuación, crea un objeto de excepción. Si escribes código que maneje la excepción, el programa seguirá ejecutándose. Si no manejas la excepción, el programa se detendrá y mostrará un mensaje *traceback*, que incluye un informe de la excepción que se ha producido.

Las excepciones se manejan con los bloques try-except. Un bloque *try*-*except* pide a Python que haga algo, pero también le dice qué hacer si se produce una excepción. Si utilizas los bloques try-except, tus programas seguirán ejecutándose aunque las cosas empiecen a ir mal. En lugar de rastreos, que pueden resultar confusos de leer para los usuarios, éstos verán mensajes de error amigables que tú has escrito.

### Manejo de la excepción ZeroDivisionError

Veamos un error sencillo que hace que Python lance una excepción. Probablemente sepas que es imposible dividir un número por cero, pero pidámosle a Python que lo haga de todos modos:

**division\_calculator.py**

print(5/0)

Python no puede hacerlo, así que obtenemos un traceback:

Traceback (most recent call last):  
 File "division\_calculator.py", line 1, in <module>  
 print(5/0)  
 ~^~  
❶ ZeroDivisionError: division by zero

El error notificado en el traceback, ZeroDivisionError, es un objeto de excepción ❶. Python crea este tipo de objeto en respuesta a una situación en la que no puede hacer lo que le pedimos. Cuando esto ocurre, Python detiene el programa y nos indica el tipo de excepción que se ha producido. Podemos utilizar esta información para modificar nuestro programa. Le diremos a Python qué hacer cuando se produzca este tipo de excepción; así, si vuelve a ocurrir, estaremos preparados.

### Utilizar bloques try-except

Cuando creas que se puede producir un error, puedes escribir un bloque try-except para manejar la excepción que se pueda producir. Le dices a Python que intente ejecutar algún código, y le dices qué hacer si el código produce un tipo concreto de excepción.

Este es el aspecto de un bloque try-except para manejar la excepción ZeroDivisionError:

try:  
 print(5/0)  
except ZeroDivisionError:  
 print("You can't divide by zero!")

Colocamos print(5/0), la línea que causó el error, dentro de un bloque try. Si el código del bloque try funciona, Python se salta el bloque except. Si el código del bloque try provoca un error, Python busca un bloque except cuyo error coincida con el que se ha producido, y ejecuta el código de ese bloque.

En este ejemplo, el código del bloque try produce un ZeroDivisionError, por lo que Python busca un bloque except que le indique cómo responder. Python ejecuta entonces el código de ese bloque, y el usuario ve un mensaje de error amigable en lugar de un rastreo:

You can't divide by zero!

Si hubiera más código después del bloque try-except, el programa seguiría ejecutándose porque le hemos dicho a Python cómo manejar el error. Veamos un ejemplo en el que la captura de un error puede permitir que un programa siga ejecutándose.

### Utilizar excepciones para evitar fallos

Manejar los errores correctamente es especialmente importante cuando el programa tiene más trabajo que hacer después de que se produzca el error. Esto ocurre a menudo en los programas que piden a los usuarios que introduzcan datos. Si el programa responde adecuadamente a las entradas no válidas, puede pedir más entradas válidas en lugar de bloquearse.

Creemos una calculadora sencilla que sólo haga divisiones:

**division\_calculator.py**

print("Give me two numbers, and I'll divide them.")  
print("Enter 'q' to quit.")  
  
while True:  
❶ first\_number = input("\nFirst number: ")  
 if first\_number == 'q':  
 break  
❷ second\_number = input("Second number: ")  
 if second\_number == 'q':  
 break  
❸ answer = int(first\_number) / int(second\_number)  
 print(answer)

Este programa pide al usuario que introduzca un first\_number ❶ y, si no introduce q para salir, un second\_number ❷. A continuación, dividimos estos dos números para obtener un answer ❸. Este programa no hace nada para manejar los errores, por lo que pedirle que divida por cero hace que se bloquee:

Give me two numbers, and I'll divide them.  
Enter 'q' to quit.  
  
First number: 5  
Second number: 0  
Traceback (most recent call last):  
 File "division\_calculator.py", line 11, in <module>  
 answer = int(first\_number) / int(second\_number)  
 ~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~  
ZeroDivisionError: division by zero

Es malo que el programa se bloquee, pero tampoco es buena idea dejar que los usuarios vean las trazas. Los usuarios no técnicos se sentirán confundidos por ellos, y en un entorno malicioso, los atacantes aprenderán más de lo que tú quieres. Por ejemplo, sabrán el nombre del archivo de tu programa y verán una parte de tu código que no funciona correctamente. A veces, un atacante hábil puede utilizar esta información para determinar qué tipo de ataques utilizar contra tu código.

### El bloque else

Podemos hacer que este programa sea más resistente a los errores envolviendo la línea que podría producir errores en un bloque try-except. El error se produce en la línea que realiza la división, así que ahí es donde pondremos el bloque try-except. Este ejemplo también incluye un bloque else. Cualquier código que dependa de que el bloque try se ejecute correctamente va en el bloque else:

--snip--  
while True:  
 --snip--  
 if second\_number == 'q':  
 break  
❶ try:  
 answer = int(first\_number) / int(second\_number)  
❷ except ZeroDivisionError:  
 print("You can't divide by 0!")  
❸ else:  
 print(answer)

Pedimos a Python que intente completar la operación de división en un bloque try ❶, que incluye sólo el código que podría causar un error. Cualquier código que dependa del éxito del bloque try se añade al bloque else. En este caso, si la operación de división tiene éxito, utilizamos el bloque else para imprimir el resultado ❸.

El bloque except indica a Python cómo responder cuando surge un ❷ ZeroDivisionError . Si el bloque try no tiene éxito debido a un error de división por cero, imprimimos un mensaje amigable indicando al usuario cómo evitar este tipo de error. El programa sigue ejecutándose y el usuario nunca ve un rastreo:

Give me two numbers, and I'll divide them.  
Enter 'q' to quit.  
  
First number: 5  
Second number: 0  
You can't divide by 0!  
  
First number: 5  
Second number: 2  
2.5  
  
First number: q

El único código que debe ir en un bloque try es el código que pueda provocar el lanzamiento de una excepción. A veces tendrás código adicional que sólo debe ejecutarse si el bloque try ha tenido éxito; este código va en el bloque else. El bloque except indica a Python qué hacer en caso de que se produzca una determinada excepción al intentar ejecutar el código del bloque try.

Al anticiparte a las posibles fuentes de error, puedes escribir programas robustos que sigan ejecutándose aunque se encuentren con datos no válidos y recursos ausentes. Tu código será resistente a los errores inocentes del usuario y a los ataques maliciosos.

### Manejo de la Excepción FileNotFoundError

Un problema común cuando se trabaja con archivos es la gestión de archivos perdidos. El archivo que buscas puede estar en una ubicación diferente, el nombre del archivo puede estar mal escrito o puede no existir en absoluto. Puedes manejar todas estas situaciones con un bloque try-except.

Intentemos leer un archivo que no existe. El siguiente programa intenta leer el contenido de *Alice in Wonderland*, pero no he guardado el archivo *alice.txt* en el mismo directorio que *alice.py*:

**alice.py**

from pathlib import Path  
  
path = Path('alice.txt')  
contents = path.read\_text(encoding='utf-8')

Ten en cuenta que aquí estamos utilizando read\_text() de una forma ligeramente distinta a la que has visto antes. El argumento encoding es necesario cuando la codificación por defecto de tu sistema no coincide con la codificación del archivo que se está leyendo. Lo más probable es que esto ocurra al leer de un archivo que no se creó en tu sistema.

Python no puede leer de un archivo que falta, por lo que lanza una excepción:

Traceback (most recent call last):  
❶ File "alice.py", line 4, in <module>  
❷ contents = path.read\_text(encoding='utf-8')  
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
 File "/.../pathlib.py", line 1056, in read\_text  
 with self.open(mode='r', encoding=encoding, errors=errors) as f:  
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
 File "/.../pathlib.py", line 1042, in open  
 return io.open(self, mode, buffering, encoding, errors, newline)  
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
❸ FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'

Éste es un rastreo más largo que los que hemos visto anteriormente, así que veamos cómo dar sentido a rastreos más complejos. A menudo es mejor empezar por el final del rastreo. En la última línea, podemos ver que se lanzó una excepción FileNotFoundError ❸. Esto es importante porque nos dice qué tipo de excepción utilizar en el bloque except que vamos a escribir.

Mirando hacia atrás, cerca del principio del rastreo ❶, podemos ver que el error se produjo en la línea 4 del archivo *alice.py*. La siguiente línea muestra la línea de código que causó el error ❷. El resto del rastreo muestra código de las bibliotecas que intervienen en la apertura y lectura de archivos. Normalmente no es necesario que leas o comprendas todas estas líneas en un rastreo.

Para tratar el error que se está produciendo, el bloque try comenzará con la línea que se identificó como problemática en el rastreo. En nuestro ejemplo, es la línea que contiene read\_text():

from pathlib import Path  
  
path = Path('alice.txt')  
try:  
 contents = path.read\_text(encoding='utf-8')  
❶ except FileNotFoundError:  
 print(f"Sorry, the file {path} does not exist.")

En este ejemplo, el código del bloque try produce un FileNotFoundError, así que escribimos un bloque except que coincide con ese error ❶. Python ejecuta entonces el código de ese bloque cuando no encuentra el archivo, y el resultado es un mensaje de error amigable en lugar de un rastreo:

Sorry, the file alice.txt does not exist.

El programa no tiene nada más que hacer si el archivo no existe, así que ésta es toda la salida que vemos. Continuemos con este ejemplo y veamos cómo puede ayudar el manejo de excepciones cuando trabajas con más de un archivo.

### Analizar texto

Puedes analizar archivos de texto que contengan libros enteros. Muchas obras clásicas de la literatura están disponibles como simples archivos de texto porque son de dominio público. Los textos utilizados en esta sección proceden del Proyecto Gutenberg[(https://gutenberg.org)](https://gutenberg.org). El Proyecto Gutenberg mantiene una colección de obras literarias que están disponibles en el dominio público, y es un gran recurso si te interesa trabajar con textos literarios en tus proyectos de programación.

Extraigamos el texto de *Alice in Wonderland* e intentemos contar el número de palabras del texto. Para ello, utilizaremos el método de cadena split(), que por defecto divide una cadena siempre que encuentra algún espacio en blanco:

from pathlib import Path  
  
path = Path('alice.txt')  
try:  
 contents = path.read\_text(encoding='utf-8')  
except FileNotFoundError:  
 print(f"Sorry, the file {path} does not exist.")  
else:  
 # Count the approximate number of words in the file:  
❶ words = contents.split()  
❷ num\_words = len(words)  
 print(f"The file {path} has about {num\_words} words.")

He movido el archivo *alice.txt* al directorio correcto, así que el bloque try funcionará esta vez. Tomamos la cadena contents, que ahora contiene todo el texto de *Alice in Wonderland* como una cadena larga, y utilizamos split() para producir una lista de todas las palabras del libro ❶. Utilizando len() en esta lista ❷ obtenemos una buena aproximación al número de palabras del texto original. Por último, imprimimos una sentencia que informa de cuántas palabras se han encontrado en el archivo. Este código se coloca en el bloque else porque sólo funciona si el código del bloque try se ejecutó correctamente.

La salida nos dice cuántas palabras hay en *alice.txt*:

The file alice.txt has about 29594 words.

El recuento es un poco alto porque el editor proporciona información adicional en el archivo de texto utilizado aquí, pero es una buena aproximación a la longitud de *Alice in Wonderland*.

### Trabajar con varios archivos

Añadamos más libros para analizar, pero antes traslademos el grueso de este programa a una función llamada count\_words(). Esto facilitará la ejecución del análisis para varios libros:

**word\_count.py**

from pathlib import Path  
  
def count\_words(path):  
❶ """Count the approximate number of words in a file."""  
 try:  
 contents = path.read\_text(encoding='utf-8')  
 except FileNotFoundError:  
 print(f"Sorry, the file {path} does not exist.")  
 else:  
 # Count the approximate number of words in the file:  
 words = contents.split()  
 num\_words = len(words)  
 print(f"The file {path} has about {num\_words} words.")  
  
path = Path('alice.txt')  
count\_words(path)

La mayor parte de este código no ha cambiado. Sólo ha sido sangrado y trasladado al cuerpo de count\_words(). Es una buena costumbre mantener los comentarios actualizados cuando modificas un programa, así que el comentario también se ha cambiado a un docstring y se ha reformulado ligeramente ❶.

Ahora podemos escribir un bucle corto para contar las palabras de cualquier texto que queramos analizar. Para ello, almacenamos los nombres de los archivos que queremos analizar en una lista y, a continuación, llamamos a count\_words() para cada archivo de la lista. Intentaremos contar las palabras de *Alice in Wonderland*, *Siddhartha*, *Moby Dick*, y *Little Women*, todos ellos de dominio público. He dejado intencionadamente *siddhartha.txt* fuera del directorio que contiene *word\_count.py*, para que podamos ver lo bien que nuestro programa gestiona un archivo que falta:

from pathlib import Path  
  
def count\_words(filename):  
 --snip--  
  
filenames = ['alice.txt', 'siddhartha.txt', 'moby\_dick.txt',   
 'little\_women.txt']  
for filename in filenames:  
❶ path = Path(filename)  
 count\_words(path)

Los nombres de los archivos se almacenan como simples cadenas. A continuación, cada cadena se convierte en un objeto Path ❶, antes de la llamada a count\_words(). El archivo *siddhartha.txt* que falta no tiene ningún efecto sobre el resto de la ejecución del programa:

The file alice.txt has about 29594 words.  
Sorry, the file siddhartha.txt does not exist.  
The file moby\_dick.txt has about 215864 words.  
The file little\_women.txt has about 189142 words.

Utilizar el bloque try-except en este ejemplo proporciona dos ventajas significativas. Evitamos que nuestros usuarios vean un rastreo, y dejamos que el programa siga analizando los textos que es capaz de encontrar. Si no atrapamos el FileNotFoundError que plantea *siddhartha.txt*, el usuario vería un rastreo completo, y el programa dejaría de ejecutarse tras intentar analizar *Siddhartha*. Nunca analizaría *Moby Dick* ni *Little Women*.

### Fallar en silencio

En el ejemplo anterior, informamos a nuestros usuarios de que uno de los archivos no estaba disponible. Pero no es necesario que informes de todas las excepciones que detectes. A veces, querrás que el programa falle silenciosamente cuando se produzca una excepción y continúe como si no hubiera pasado nada. Para hacer que un programa falle silenciosamente, escribe un bloque try como de costumbre, pero dile explícitamente a Python que no haga nada en el bloque except. Python tiene una sentencia pass que le dice que no haga nada en un bloque:

def count\_words(path):  
 """Count the approximate number of words in a file."""  
 try:  
 --snip--  
 except FileNotFoundError:  
 pass  
 else:  
 --snip--

La única diferencia entre este listado y el anterior es la sentencia pass en el bloque except. Ahora, cuando se lanza un FileNotFoundError, el código del bloque except se ejecuta, pero no ocurre nada. No se produce ningún rastreo, y no hay ninguna salida en respuesta al error que se ha producido. Los usuarios ven el recuento de palabras de cada archivo que existe, pero no ven ninguna indicación de que no se ha encontrado un archivo:

The file alice.txt has about 29594 words.  
The file moby\_dick.txt has about 215864 words.  
The file little\_women.txt has about 189142 words.

La declaración pass también actúa como marcador de posición. Es un recordatorio de que estás eligiendo no hacer nada en un punto concreto de la ejecución de tu programa y que puede que quieras hacer algo allí más adelante. Por ejemplo, en este programa podríamos decidir escribir cualquier nombre de archivo que falte en un archivo llamado *missing\_files.txt*. Nuestros usuarios no verían este archivo, pero nosotros podríamos leerlo y ocuparnos de cualquier texto que faltara.

### Decidir de qué errores informar

¿Cómo sabes cuándo informar de un error a tus usuarios y cuándo dejar que tu programa falle silenciosamente? Si los usuarios saben qué textos se supone que deben analizarse, es posible que aprecien un mensaje que les informe de por qué no se analizaron algunos textos. Si los usuarios esperan ver algunos resultados, pero no saben qué libros se supone que deben analizarse, puede que no necesiten saber que algunos textos no estaban disponibles. Dar a los usuarios información que no están buscando puede disminuir la usabilidad de tu programa. Las estructuras de gestión de errores de Python te dan un control preciso sobre cuánto compartir con los usuarios cuando las cosas van mal; tú decides cuánta información compartir.

El código bien escrito y correctamente probado no es muy propenso a errores internos, como errores sintácticos o lógicos. Pero cada vez que tu programa depende de algo externo, como la entrada de datos del usuario, la existencia de un archivo o la disponibilidad de una conexión de red, existe la posibilidad de que se produzca una excepción. Un poco de experiencia te ayudará a saber dónde incluir bloques de gestión de excepciones en tu programa y cuánto informar a los usuarios sobre los errores que surjan.

## Pruébalo tú mismo

10-6. Suma: Un problema común cuando se solicita una entrada numérica ocurre cuando la gente proporciona texto en lugar de números. Cuando intentes convertir la entrada en un int, obtendrás un ValueError. Escribe un programa que pida dos números. Súmalos e imprime el resultado. Atrapa el ValueError si alguno de los valores de entrada no es un número, e imprime un mensaje de error amigable. Prueba tu programa introduciendo dos números y luego introduciendo un texto en lugar de un número.

10-7. Calculadora de sumas: Envuelve tu código del Ejercicio 10-5 en un bucle while para que el usuario pueda seguir introduciendo números, aunque se equivoque e introduzca texto en lugar de un número.

10-8. Gatos y Perros: Crea dos archivos, *cats.txt* y *dogs.txt*. Guarda al menos tres nombres de gatos en el primer fichero y tres nombres de perros en el segundo. Escribe un programa que intente leer estos archivos e imprima su contenido en la pantalla. Envuelve tu código en un bloque try-except para detectar el error FileNotFound, e imprime un mensaje amigable si falta un archivo. Mueve uno de los archivos a una ubicación diferente de tu sistema, y asegúrate de que el código del bloque except se ejecuta correctamente.

10-9. Perros y gatos silenciosos: Modifica tu bloque except del Ejercicio 10-7 para que falle silenciosamente si falta alguno de los archivos.

10-10. Palabras comunes: Visita el Proyecto Gutenberg[(https://gutenberg.org)](https://gutenberg.org) y encuentra algunos textos que te gustaría analizar. Descarga los archivos de texto de estas obras, o copia el texto en bruto desde tu navegador a un archivo de texto en tu ordenador.

Puedes utilizar el método count() para averiguar cuántas veces aparece una palabra o frase en una cadena. Por ejemplo, el siguiente código cuenta el número de veces que aparece 'row' en una cadena:

>>> line = "Row, row, row your boat"  
>>> line.count('row')  
2  
>>> line.lower().count('row')  
3

Observa que la conversión de la cadena a minúsculas mediante lower() capta todas las apariciones de la palabra que buscas, independientemente de cómo esté formateada.

Escribe un programa que lea los archivos que has encontrado en el Proyecto Gutenberg y determine cuántas veces aparece la palabra 'the' en cada texto. Esto será una aproximación porque también contará palabras como 'then' y 'there'. Prueba a contar 'the ', con un espacio en la cadena, y verás cuánto más baja es tu cuenta.

## Almacenamiento de datos

Muchos de tus programas pedirán a los usuarios que introduzcan cierto tipo de información. Podrías permitir a los usuarios almacenar preferencias en un juego o proporcionar datos para una visualización. Sea cual sea el enfoque de tu programa, almacenarás la información que los usuarios proporcionen en estructuras de datos como listas y diccionarios. Cuando los usuarios cierren un programa, casi siempre querrás guardar la información que introdujeron. Una forma sencilla de hacerlo consiste en almacenar tus datos utilizando el módulo json.

El módulo json te permite convertir estructuras de datos sencillas de Python en cadenas con formato JSON, y luego cargar los datos desde ese archivo la próxima vez que se ejecute el programa. También puedes utilizar json para compartir datos entre distintos programas Python. Aún mejor, el formato de datos JSON no es específico de Python, por lo que puedes compartir los datos que almacenes en formato JSON con personas que trabajen en muchos otros lenguajes de programación. Es un formato útil y portátil, y es fácil de aprender.

## Nota

El formato *JSON* *(JavaScript Object Notation)* se desarrolló originalmente para JavaScript. Sin embargo, desde entonces se ha convertido en un formato común utilizado por muchos lenguajes, incluido Python.

### Utilización de json.dumps() y json.loads()

Vamos a escribir un breve programa que almacene un conjunto de números y otro programa que vuelva a leer estos números en la memoria. El primer programa utilizará json.dumps() para almacenar el conjunto de números, y el segundo programa utilizará json.loads().

La función json.dumps() toma un argumento: un dato que debe convertirse al formato JSON. La función devuelve una cadena, que podemos escribir en un archivo de datos:

**number\_writer.py**

from pathlib import Path  
import json  
  
numbers = [2, 3, 5, 7, 11, 13]  
  
❶ path = Path('numbers.json')  
❷ contents = json.dumps(numbers)  
path.write\_text(contents)

Primero importamos el módulo json, y luego creamos una lista de números con los que trabajar. A continuación, elegimos un nombre de archivo en el que almacenar la lista de números ❶. Es habitual utilizar la extensión de archivo *.json* para indicar que los datos del archivo se almacenan en formato JSON. A continuación, utilizamos la función json.dumps() ❷ para generar una cadena que contenga la representación JSON de los datos con los que estamos trabajando. Una vez que tenemos esta cadena, la escribimos en el archivo utilizando el mismo método write\_text() que utilizamos antes.

Este programa no tiene salida, pero abramos el archivo *numbers.json* y echémosle un vistazo. Los datos se almacenan en un formato parecido al de Python:

[2, 3, 5, 7, 11, 13]

Ahora escribiremos un programa independiente que utilice json.loads() para volver a leer la lista en la memoria:

**number\_reader.py**

from pathlib import Path  
import json  
  
❶ path = Path('numbers.json')  
❷ contents = path.read\_text()  
❸ numbers = json.loads(contents)  
  
print(numbers)

Nos aseguramos de leer del mismo archivo que escribimos en ❶. Como el archivo de datos es sólo un archivo de texto con un formato específico, podemos leerlo con el método read\_text() ❷. A continuación, pasamos el contenido del archivo a json.loads() ❸. Esta función recibe una cadena con formato JSON y devuelve un objeto Python (en este caso, una lista), que asignamos a numbers. Por último, imprimimos la lista de números recuperada y vemos que es la misma lista creada en *number\_writer.py*:

[2, 3, 5, 7, 11, 13]

Ésta es una forma sencilla de compartir datos entre dos programas.

### Guardar y leer datos generados por el usuario

Guardar datos con json es útil cuando trabajas con datos generados por el usuario, porque si no almacenas la información de tu usuario de alguna manera, la perderás cuando el programa deje de ejecutarse. Veamos un ejemplo en el que pedimos al usuario su nombre la primera vez que ejecuta un programa y luego recordamos su nombre cuando vuelva a ejecutar el programa.

Empecemos por almacenar el nombre del usuario:

**remember\_me.py**

from pathlib import Path  
import json  
  
❶ username = input("What is your name? ")  
  
❷ path = Path('username.json')  
contents = json.dumps(username)  
path.write\_text(contents)  
  
❸ print(f"We'll remember you when you come back, {username}!")

Primero pedimos un nombre de usuario para almacenarlo ❶. A continuación, escribimos los datos que acabamos de recoger en un archivo llamado *username.json* ❷. Luego imprimimos un mensaje informando al usuario de que hemos almacenado su información ❸:

What is your name? Eric  
We'll remember you when you come back, Eric!

Ahora escribamos un nuevo programa que salude a un usuario cuyo nombre ya ha sido almacenado:

**greet\_user.py**

from pathlib import Path  
import json  
  
❶ path = Path('username.json')  
contents = path.read\_text()  
❷ username = json.loads(contents)  
  
print(f"Welcome back, {username}!")

Leemos el contenido del fichero de datos ❶ y luego utilizamos json.loads() para asignar los datos recuperados a la variable username ❷. Como hemos recuperado el nombre de usuario, podemos darle la bienvenida con un saludo personalizado:

Welcome back, Eric!

Necesitamos combinar estos dos programas en un único archivo. Cuando alguien ejecute *remember\_me.py*, queremos recuperar su nombre de usuario de la memoria si es posible; si no, le pediremos un nombre de usuario y lo almacenaremos en *username.json* para la próxima vez. Podríamos escribir aquí un bloque try-except para responder adecuadamente si *username.json* no existe, pero en su lugar utilizaremos un práctico método del módulo pathlib:

**remember\_me.py**

from pathlib import Path  
import json  
  
path = Path('username.json')  
❶ if path.exists():  
 contents = path.read\_text()  
 username = json.loads(contents)  
 print(f"Welcome back, {username}!")  
❷ else:  
 username = input("What is your name? ")  
 contents = json.dumps(username)  
 path.write\_text(contents)  
 print(f"We'll remember you when you come back, {username}!")

Hay muchos métodos útiles que puedes utilizar con los objetos Path. El método exists() devuelve True si existe un archivo o carpeta y False si no existe. Aquí utilizamos path.exists() para averiguar si ya se ha almacenado un nombre de usuario ❶. Si *username.json* existe, cargamos el nombre de usuario e imprimimos un saludo personalizado al usuario.

Si el archivo *username.json* no existe ❷, pedimos un nombre de usuario y almacenamos el valor que introduzca el usuario. También imprimimos el mensaje familiar de que nos acordaremos de ellos cuando vuelvan.

Sea cual sea el bloque que se ejecute, el resultado es un nombre de usuario y un saludo apropiado. Si es la primera vez que se ejecuta el programa, ésta es la salida:

What is your name? Eric  
We'll remember you when you come back, Eric!

Si no:

Welcome back, Eric!

Esta es la salida que ves si el programa ya se ha ejecutado al menos una vez. Aunque los datos de esta sección son sólo una cadena, el programa funcionaría igual de bien con cualquier dato que pueda convertirse a una cadena con formato JSON.

### Refactorización de

A menudo, llegarás a un punto en el que tu código funcionará, pero reconocerás que podrías mejorarlo dividiéndolo en una serie de funciones que tengan tareas específicas. Este proceso se llama *refactoring*. La refactorización hace que tu código sea más limpio, más fácil de entender y más fácil de ampliar.

Podemos refactorizar *remember\_me.py* trasladando la mayor parte de su lógica a una o más funciones. El objetivo de *remember\_me.py* es saludar al usuario, así que vamos a trasladar todo nuestro código existente a una función llamada greet\_user():

**remember\_me.py**

from pathlib import Path  
import json  
  
def greet\_user():  
❶ """Greet the user by name."""  
 path = Path('username.json')  
 if path.exists():  
 contents = path.read\_text()  
 username = json.loads(contents)  
 print(f"Welcome back, {username}!")  
 else:  
 username = input("What is your name? ")  
 contents = json.dumps(username)  
 path.write\_text(contents)  
 print(f"We'll remember you when you come back, {username}!")  
  
greet\_user()

Como ahora utilizamos una función, reescribimos los comentarios como un docstring que refleje cómo funciona actualmente el programa ❶. Este archivo es un poco más limpio, pero la función greet\_user() hace algo más que saludar al usuario: también recupera un nombre de usuario almacenado si existe y solicita un nuevo nombre de usuario si no existe.

Vamos a refactorizar greet\_user() para que no realice tantas tareas diferentes. Empezaremos moviendo el código para recuperar un nombre de usuario almacenado a una función separada:

from pathlib import Path  
import json  
  
def get\_stored\_username(path):  
❶ """Get stored username if available."""  
 if path.exists():  
 contents = path.read\_text()  
 username = json.loads(contents)  
 return username  
 else:  
❷ return None  
  
def greet\_user():  
 """Greet the user by name."""  
 path = Path('username.json')  
 username = get\_stored\_username(path)  
❸ if username:  
 print(f"Welcome back, {username}!")  
 else:  
 username = input("What is your name? ")  
 contents = json.dumps(username)  
 path.write\_text(contents)  
 print(f"We'll remember you when you come back, {username}!")  
  
greet\_user()

La nueva función get\_stored\_username() ❶ tiene un propósito claro, como se indica en el docstring. Esta función recupera un nombre de usuario almacenado y lo devuelve si lo encuentra. Si la ruta que se pasa a get\_stored\_username() no existe, la función devuelve None ❷. Esto es una buena práctica: una función debe devolver el valor que esperas o debe devolver None. Esto nos permite realizar una sencilla prueba con el valor de retorno de la función. Imprimimos un mensaje de bienvenida al usuario si el intento de recuperar un nombre de usuario tiene éxito ❸, y si no lo tiene, le pedimos un nuevo nombre de usuario.

Deberíamos eliminar un bloque más de código de greet\_user(). Si el nombre de usuario no existe, deberíamos mover el código que solicita un nuevo nombre de usuario a una función dedicada a ese fin:

from pathlib import Path  
import json  
  
def get\_stored\_username(path):  
 """Get stored username if available."""  
 --snip--  
  
def get\_new\_username(path):  
 """Prompt for a new username."""  
 username = input("What is your name? ")  
 contents = json.dumps(username)  
 path.write\_text(contents)  
 return username  
  
def greet\_user():  
 """Greet the user by name."""  
 path = Path('username.json')  
❶ username = get\_stored\_username(path)  
 if username:  
 print(f"Welcome back, {username}!")  
 else:  
❷ username = get\_new\_username(path)  
 print(f"We'll remember you when you come back, {username}!")  
  
greet\_user()

Cada función de esta versión final de *remember\_me.py* tiene un único y claro propósito. Llamamos a greet\_user(), y esa función imprime un mensaje apropiado: o bien da la bienvenida a un usuario existente o bien saluda a un nuevo usuario. Para ello, llama a get\_stored\_username() ❶, que sólo se encarga de recuperar un nombre de usuario almacenado, si existe. Por último, si es necesario, greet\_user() llama a get\_new\_username()❷, que sólo se encarga de obtener un nuevo nombre de usuario y almacenarlo. Esta compartimentación del trabajo es una parte esencial para escribir un código claro que sea fácil de mantener y ampliar.

## Pruébalo tú mismo

10-11. Número favorito: Escribe un programa que pregunte por el número favorito del usuario. Utiliza json.dumps() para almacenar este número en un archivo. Escribe otro programa que lea este valor e imprima el mensaje "¡Sé cuál es tu número favorito! Es \_\_\_\_\_".

10-12. Número Favorito Recordado: Combina los dos programas que escribiste en el Ejercicio 10-11 en un solo archivo. Si el número ya está almacenado, informa del número favorito al usuario. Si no, pregunta al usuario por su número favorito y guárdalo en un archivo. Ejecuta el programa dos veces para comprobar que funciona.

10-13. Diccionario del usuario: El ejemplo de *remember\_me.py* sólo almacena un dato, el nombre de usuario. Amplía este ejemplo pidiendo dos datos más sobre el usuario, y almacena toda la información que recopiles en un diccionario. Escribe este diccionario en un archivo utilizando json.dumps(), y vuelve a leerlo utilizando json.loads(). Imprime un resumen que muestre exactamente lo que tu programa recuerda sobre el usuario.

10-14. Verificar usuario: El listado final de *remember\_me.py* asume que el usuario ya ha introducido su nombre de usuario o que el programa se está ejecutando por primera vez. Deberíamos modificarlo en caso de que el usuario actual no sea la persona que utilizó el programa por última vez.

Antes de imprimir un mensaje de bienvenida en greet\_user(), pregunta al usuario si éste es el nombre de usuario correcto. Si no lo es, llama a get\_new\_username() para obtener el nombre de usuario correcto.

## Resumen

En este capítulo has aprendido a trabajar con archivos. Aprendiste a leer todo el contenido de un archivo, y luego a trabajar con el contenido línea a línea si es necesario. Aprendiste a escribir todo el texto que quieras en un archivo. También leíste sobre excepciones y sobre cómo manejar las excepciones que probablemente verás en tus programas. Por último, aprendiste a almacenar estructuras de datos Python para poder guardar la información que te proporcionen tus usuarios, evitando que tengan que empezar de nuevo cada vez que ejecuten un programa.

En el Capítulo 11, aprenderás formas eficaces de probar tu código. Esto te ayudará a confiar en que el código que desarrollas es correcto, y te ayudará a identificar los errores que se introduzcan a medida que sigas construyendo sobre los programas que has escrito.