Apéndice B: Visión general de los temas de Python

## **B** Visión general de los temas de Python



El objetivo de este apéndice es doble: proporcionar un rápido repaso de algunos temas de Python que no se introdujeron a fondo en los capítulos e introducir temas que te ayudarán a escribir mejores programas en Python.

### **if \_\_name\_\_ == '\_\_main\_\_'**

A lo largo del libro, hemos utilizado el siguiente bloque de código, donde func() es una función que hemos definido en el programa:

if \_\_name\_\_ == '\_\_main\_\_':  
# Do something  
func()

Este bloque de código garantiza que las sentencias dentro del bloque sólo se ejecuten cuando el programa se ejecute solo.

Cuando se ejecuta un programa, la variable especial \_\_name\_\_ se establece automáticamente en \_\_main\_\_, por lo que la condición if se evalúa como True y se llama a la función func(). Sin embargo, \_\_name\_\_ se establece de forma diferente cuando importas el programa en otro programa (ver "[Reutilizar código](app02.html#app02lev1sec07)" en [la página 235](app02.html#page_235)).

He aquí una demostración rápida. Considera el siguiente programa, que llamaremos *factorial.py*:

# Find the factorial of a number  
def fact(n):  
p = 1  
for i in range(1, n+1):  
p = p\*i  
return p  
  
➊ print(\_\_name\_\_)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
n = int(input('Enter an integer to find the factorial of: '))  
f = fact(n)  
print('Factorial of {0}: {1}'.format(n, f))

El programa define una función, fact(), que calcula el factorial del número entero que se le pasa. Cuando lo ejecutas, imprime \_\_main\_\_, que corresponde a la sentencia print en ➊, porque \_\_name\_\_ se establece automáticamente en \_\_main\_\_. A continuación, pide que se introduzca un número entero, calcula el factorial y lo imprime:

\_\_main\_\_  
Enter an integer to find the factorial of: 5  
Factorial of 5: 120

Ahora, supongamos que necesitas calcular el factorial en otro programa. En lugar de volver a escribir la función, decides reutilizar esta función importándola:

from factorial import fact  
if \_\_name\_\_ == '\_\_main\_\_':  
print('Factorial of 5: {0}'.format(fact(5)))

Ten en cuenta que ambos programas deben estar en el mismo directorio. Cuando ejecutes este programa, obtendrás la siguiente salida:

factorial  
Factorial of 5: 120

Cuando tu programa es importado por otro programa, el valor de la variable \_\_main\_\_ se establece en el nombre de archivo de ese programa, sin la extensión. En este caso, el valor de \_\_name\_\_ es factorial en lugar de \_\_main\_\_. Como la condición \_\_name\_\_ == '\_\_main\_\_' ahora se evalúa como False, el programa ya no solicita la entrada del usuario. ¡Elimina la condición para ver por ti mismo lo que ocurre!

En resumen, es una buena práctica utilizar if \_\_name\_\_ == '\_\_main\_\_' en tus programas para que las sentencias que quieres que se ejecuten cuando tu programa se ejecute como independiente *tampoco* se ejecuten cuando tu programa se importe a otro programa.

### **Comprensión de listas**

Supongamos que tenemos una lista de enteros y queremos crear una nueva lista que contenga los cuadrados de los elementos de la lista original. Esta es una forma de hacerlo que ya te resultará familiar:

>>> x = [1, 2, 3, 4]  
>>> x\_square = []  
➊ >>> for n in x:  
➋ x\_square.append(n\*\*2)  
>>> x\_square  
[1, 4, 9, 16]

Aquí utilizamos un patrón de código que hemos utilizado en varios programas a lo largo del libro. Creamos una lista vacía, x\_square, y luego la vamos añadiendo sucesivamente a medida que calculamos el cuadrado. Podemos hacerlo de forma más eficiente utilizando *comprensiones de lista*:

➌ >>> x\_square = [n\*\*2 for n in x]  
>>> x\_square  
[1, 4, 9, 16]

La sentencia ➌ se denomina comprensión de *lista* en Python. Consiste en una expresión -aquí, n\*\*2- seguida de un bucle for, for n in x. Observa que básicamente nos permite combinar las dos expresiones ➊ y ➋ en una sola para crear una nueva lista en una sola expresión.

Como otro ejemplo, considera uno de los programas que escribimos en "[Dibujar la trayectoria](ch02.html#ch02lev3sec05)" en [la página 51](ch02.html#page_51) para dibujar la trayectoria de un cuerpo en movimiento de proyectil. En estos programas, tenemos el siguiente bloque de código para calcular las *coordenadas* *x* e *y* del cuerpo en cada instante de tiempo:

# Find time intervals  
intervals = frange(0, t\_flight, 0.001)  
# List of x and y coordinates  
x = []  
y = []  
for t in intervals:  
x.append(u\*math.cos(theta)\*t)  
y.append(u\*math.sin(theta)\*t - 0.5\*g\*t\*t)

Utilizando la comprensión de listas, puedes reescribir el bloque de código como sigue:

# Find time intervals  
intervals = frange(0, t\_flight, 0.001)  
# List of x and y coordinates  
  
x = [u\*math.cos(theta)\*t for t in intervals]  
y = [u\*math.sin(theta)\*t - 0.5\*g\*t\*t for t in intervals]

Ahora el código es más compacto, ya que no tenías que crear las listas vacías, escribir un bucle for y añadir a las listas. La comprensión de listas te permite hacer esto en una sola sentencia.

También puedes añadir condicionales a una comprensión de lista para elegir selectivamente qué elementos de la lista se evalúan en la expresión. Considera, una vez más, el primer ejemplo:

>>> x = [1, 2, 3, 4]  
>>> x\_square = [n\*\*2 for n in x if n%2 == 0]  
>>> x\_square  
[4, 16]

En esta comprensión de lista, utilizamos la condición if para decirle explícitamente a Python que evalúe la expresión n\*\*2 sólo en los elementos pares de la lista x.

### **Estructura de datos del diccionario**

La primera vez que utilizamos un diccionario Python fue en el [Capítulo 4](ch04.html#ch04), cuando implementamos el método subs() en SymPy. Exploremos los diccionarios de Python con más detalle. Considera un diccionario sencillo:

>>> d = {'key1': 5, 'key2': 20}

Este código crea un diccionario con dos claves -'key1' y 'key2'- con los valores 5 y 20, respectivamente. Sólo las cadenas, los números y las tuplas pueden ser claves en un diccionario Python. Estos tipos de datos se denominan tipos de datos *inmutables* -una vez creados, no pueden modificarse-, por lo que una lista no puede ser una clave porque podemos añadir y eliminar elementos de una lista.

Ya sabemos que para recuperar el valor correspondiente a 'key1' en el diccionario, tenemos que especificarlo como d['key1']. Éste es uno de los casos de uso más habituales de un diccionario. Un caso de uso relacionado es comprobar si el diccionario contiene una determinada clave, 'x'. Podemos comprobarlo de la siguiente manera:

>>> d = {'key1': 5, 'key2': 20}  
>>> 'x' in d  
False

Una vez creado un diccionario, podemos añadirle un nuevo par clave-valor, de forma similar a como podemos añadir elementos a una lista. He aquí un ejemplo:

>>> d = {'key1': 5, 'key2': 20}  
>>> if 'x' in d:  
print(d['x'])  
else:  
d['x'] = 1  
  
>>> d  
{'key1': 5, 'x': 1, 'key2': 20}

Este fragmento de código comprueba si la clave 'x' ya existe en el diccionario, d. Si es así, imprime el valor que le corresponde; si no, añade la clave al diccionario con 1 como valor correspondiente. De forma similar al comportamiento de Python con los conjuntos, Python no puede garantizar un orden concreto de los pares clave-valor en un diccionario. Los pares clave-valor pueden estar en cualquier orden, independientemente del orden de inserción.

Además de especificar la clave como índice del diccionario, también podemos utilizar el método get() para recuperar el valor correspondiente a la clave:

>>> d.get('x')  
1

Si especificas una clave inexistente al método get(), se devuelve None. En cambio, si lo haces utilizando el estilo índice de recuperación, obtendrás un error.

El método get() también te permite establecer un valor por defecto para claves inexistentes:

>>> d.get('y', 0)  
0

No hay ninguna clave 'y' en el diccionario d, por lo que se devuelve 0. Sin embargo, si hay una clave, se devuelve el valor en su lugar:

>>> d['y'] = 1  
>>> d.get('y', 0)  
1

Los métodos keys() y values() devuelven cada uno una estructura de datos en forma de lista con todas las claves y valores, respectivamente, de un diccionario:

>>> d.keys()  
dict\_keys(['key1', 'x', 'key2', 'y'])  
>>> d.values()  
dict\_values([5, 1, 20, 1])

Para iterar sobre los pares clave y valor de un diccionario, utiliza el método items():

>>> d.items()  
dict\_items([('key1', 5), ('x', 1), ('key2', 20), ('y', 1)])

Este método devuelve una *vista* de tuplas, y cada tupla es un par clave-valor. Podemos utilizar el siguiente fragmento de código para imprimirlas bien:

>>> for k, v in d.items():  
print(k, v)  
  
key1 5  
x 1  
key2 20  
y 1

Las vistas son más eficientes en memoria que las listas, y no te permiten añadir o eliminar elementos.

### **Múltiples valores de retorno**

En los programas que hemos escrito hasta ahora, la mayoría de las funciones devuelven un único valor, pero a veces las funciones devuelven varios valores. Vimos un ejemplo de una función de este tipo en "[Medir la dispersión](ch03.html#ch03lev1sec04)" en [la página 71](ch03.html#page_71), donde en el programa para hallar el rango, devolvimos tres números de la función find\_range(). Aquí tienes otro ejemplo del enfoque que adoptamos allí:

import math  
def components(u, theta):  
x = u\*math.cos(theta)  
y = u\*math.sin(theta)  
return x, y

La función components() acepta como parámetros una velocidad, u, y un ángulo, theta, en radianes, y calcula las componentes x y y y las devuelve. Para devolver los componentes calculados, simplemente enumeramos las etiquetas Python correspondientes en la sentencia return separadas por una coma. Esto crea y devuelve una tupla formada por los elementos x y y. En el código de llamada, recibimos los valores múltiples:

if \_\_name\_\_ == '\_\_main\_\_':  
theta = math.radians(45)  
x, y = components(theta)

Como la función components() devuelve una tupla, podemos recuperar los valores devueltos utilizando índices de tupla:

c = components(theta)  
x = c[0]  
y = c[1]

Esto tiene ventajas porque no tenemos que conocer todos los valores diferentes que se devuelven. Por ejemplo, no tienes que escribir x,y,z = myfunc1() cuando la función devuelva tres valores o a,x,y,z = myfunc1() cuando la función devuelva cuatro valores, y así sucesivamente.

En cualquiera de los casos anteriores, el código que llama a la función components() debe saber cuál de los valores devueltos corresponde a cada componente de la velocidad, ya que no hay forma de saberlo a partir de los propios valores.

Un enfoque fácil de utilizar es devolver en su lugar un objeto diccionario, como vimos en el caso de la función solve() de SymPy cuando se utiliza con el argumento de la palabra clave dict=True. He aquí cómo podemos reescribir la función de componentes anterior para que devuelva un diccionario:

import math  
  
def components(theta):  
x = math.cos(theta)  
y = math.sin(theta)  
  
return {'x': x, 'y': y}

Aquí, devolvemos un diccionario con las claves 'x' y 'y' que hacen referencia a los componentes x y y y a sus correspondientes valores numéricos. Con esta nueva definición de la función, no tenemos que preocuparnos del orden de los valores devueltos. Sólo utilizamos la clave 'x' para recuperar el componente x y la clave 'y' para recuperar el componente y:

if \_\_name\_\_ == '\_\_main\_\_':  
theta = math.radians(45)  
c = components(theta)  
y = c['y']  
x = c['x']  
print(x, y)

Este enfoque elimina la necesidad de utilizar índices para referirnos a un valor devuelto concreto. El código siguiente reescribe el programa para encontrar el rango (ver "[Medir la dispersión](ch03.html#ch03lev1sec04)" en [la página 71](ch03.html#page_71)) de forma que los resultados se devuelvan como un diccionario en lugar de una tupla:

'''  
Find the range using a dictionary to return values  
'''  
def find\_range(numbers):  
lowest = min(numbers)  
highest = max(numbers)  
# Find the range  
r = highest-lowest  
return {'lowest':lowest, 'highest':highest, 'range':r}  
  
if \_\_name\_\_ == '\_\_main\_\_':  
donations = [100, 60, 70, 900, 100, 200, 500, 500, 503, 600, 1000, 1200]  
result = find\_range(donations)  
➊ print('Lowest: {0} Highest: {1} Range: {2}'.  
format(result['lowest'], result['highest'], result['range']))

La función find\_range() devuelve ahora un diccionario con las claves lowest, highest, y range y con el número más bajo, el número más alto y el rango como valores correspondientes. En ➊, simplemente utilizamos la clave correspondiente para recuperar el valor correspondiente.

Si sólo nos interesara el rango de un grupo de números y no nos importaran los números más bajo y más alto, sólo utilizaríamos result['range'] y no nos preocuparíamos de qué otros valores se devuelven.

### **Manejo de excepciones**

En el [Capítulo 1](ch01.html#ch01), aprendimos que intentar convertir una cadena como '1.1' en un número entero utilizando la función int() produce una excepción ValueError. Pero con un bloque try...except, podemos imprimir un mensaje de error fácil de usar:

>>> try:  
int('1.1')  
except ValueError:  
print('Failed to convert 1.1 to an integer')  
  
Failed to convert 1.1 to an integer

Cuando cualquier sentencia del bloque try lanza una excepción, el tipo de excepción lanzada se compara con el especificado por la sentencia except. Si hay coincidencia, el programa se reanuda en el bloque except. Si la excepción no coincide, la ejecución del programa se detiene y muestra la excepción. He aquí un ejemplo:

>>> try:  
print(1/0)  
except ValueError:  
print('Division unsuccessful')  
  
Traceback (most recent call last):  
File "<pyshell#66>", line 2, in <module>  
print(1/0)  
ZeroDivisionError: division by zero

Este bloque de código intenta una división por 0, lo que provoca una excepción ZeroDivisionError. Aunque la división se realiza en un bloque try...except, el tipo de excepción se especifica incorrectamente, y la excepción no se maneja correctamente. La forma correcta de gestionar esta excepción es especificar ZeroDivisionError como tipo de excepción.

#### ***Especificar varios tipos de excepción***

También puedes especificar varios tipos de excepción. Considera la función reciprocal(), que devuelve el recíproco del número que se le pasa:

def reciprocal(n):  
try:  
print(1/n)  
except (ZeroDivisionError, TypeError):  
print('You entered an invalid number')

Definimos la función reciprocal(), que imprime el recíproco de la entrada del usuario. Sabemos que si se llama a la función con 0, provocará una excepción ZeroDivisionError. En cambio, si pasas una cadena, provocará una excepción TypeError. La función considera ambos casos como entrada no válida y especifica tanto ZeroDivisionError como TypeError en la sentencia except como una tupla.

Intentemos llamar a la función con una entrada válida, es decir, un número distinto de cero:

>>> reciprocal(5)  
0.2

A continuación, llamamos a la función con 0 como argumento:

>>> reciprocal(0)  
Enter an integer: 0  
You entered an invalid number

El argumento 0 provoca la excepción ZeroDivisionError, que está en la tupla de tipos de excepción especificada en la sentencia except, por lo que el código imprime un mensaje de error.

Ahora, introduzcamos una cadena:

>>> reciprocal('1')

En este caso, hemos introducido un número no válido, lo que provoca la excepción TypeError. Esta excepción también está en la tupla de excepciones especificadas, por lo que el código imprime un mensaje de error. Si quieres dar un mensaje de error más específico, podemos especificar varias sentencias except como se indica a continuación:

def reciprocal(n):  
try:  
print(1/n)  
except TypeError:  
print('You must specify a number')  
except ZeroDivisionError:  
print('Division by 0 is invalid')  
  
>>> reciprocal(0)  
Division by 0 is invalid  
>>> reciprocal('1')  
You must specify a number

Además de TypeError, ValueError, y ZeroDivisionError, existen otros tipos de excepción incorporados. La documentación de Python en [*https://docs.python.org/3.4/library/exceptions.html#bltin-exceptions*](https://docs.python.org/3.4/library/exceptions.html#bltin-exceptions) enumera las excepciones incorporadas para Python 3.4.

#### ***El bloque else***

El bloque else se utiliza para especificar qué sentencias ejecutar cuando no hay ninguna excepción. Considera un ejemplo del programa que escribimos para dibujar la trayectoria de un proyectil (ver "[Dibujar la trayectoria](ch02.html#ch02lev3sec05)" en [la página 51](ch02.html#page_51)):

if \_\_name\_\_ == '\_\_main\_\_':  
try:  
u = float(input('Enter the initial velocity (m/s): '))  
theta = float(input('Enter the angle of projection (degrees): '))  
except ValueError:  
print('You entered an invalid input')  
➊ else:  
draw\_trajectory(u, theta)  
plt.show()

Si la entrada para u o theta no se pudiera convertir a un número de coma flotante, no tendría sentido que el programa llamara a las funciones draw\_trajectory() y plt.show(). En su lugar, especificamos estas dos sentencias en el bloque else en ➊. Utilizar try...except...else te permitirá gestionar distintos tipos de errores durante el tiempo de ejecución y tomar las medidas adecuadas cuando haya un error o cuando no lo haya:

1. Si hay una excepción y existe una sentencia except correspondiente al tipo de excepción planteada, la ejecución se transfiere al bloque except correspondiente.

2. Si no hay excepción, la ejecución se transfiere al bloque else.

### **Leer archivos en Python**

Abrir un archivo es el primer paso para leer datos de él. Empecemos con un ejemplo rápido. Considera un archivo que consiste en una colección de números con un número por línea:

100  
60  
70  
900  
100  
200  
500  
500  
503  
600  
1000  
1200

Queremos escribir una función que lea el archivo y devuelva una lista de esos números:

def read\_data(path):  
numbers = []  
➊ f = open(path)  
➋ for line in f:  
numbers.append(float(line))  
f.close()  
return numbers

En primer lugar, definimos la función read\_data() y creamos una lista vacía para almacenar todos los números. En ➊, utilizamos la función open() para abrir el archivo cuya ubicación se ha especificado mediante el argumento ruta. Un ejemplo de la ruta sería */home/nombredeusuario/mydata*.txt en Linux, *C:\mydata.txt* en Microsoft Windows, o */Usuarios/nombredeusuario/mydata.txt* en OS X. La función open() devuelve un objeto archivo, al que utilizamos la etiqueta f para referirnos. Podemos repasar cada línea del archivo utilizando un bucle for en ➋. Como cada línea se devuelve como una cadena, la convertimos en un número y lo añadimos a la lista numbers. El bucle deja de ejecutarse una vez que se han leído todas las líneas, y cerramos el archivo utilizando el método close(). Por último, devolvemos la lista numbers.

Esto es similar a cómo leímos los números de un archivo en [el Capítulo 3](ch03.html#ch03), aunque no tuvimos que cerrar el archivo explícitamente porque allí utilizamos un enfoque diferente. Utilizando el método del Capítulo [3](ch03.html#ch03), reescribiríamos la función anterior de la siguiente manera:

def read\_data(path):  
numbers = []  
➊ with open(path) as f:  
for line in f:  
numbers.append(float(line))  
➋ return numbers

La declaración clave aquí está en ➊. Es similar a escribir f = open(path), pero sólo parcialmente. Además de abrir el archivo y asignar el objeto archivo devuelto por open() a f, también establece un nuevo *contexto* con todas las sentencias de ese bloque -en este caso, todas las sentencias anteriores a la sentencia return. Cuando se han ejecutado todas las sentencias del cuerpo, el archivo se cierra automáticamente. Es decir, cuando la ejecución llega a la sentencia ➋, el archivo se cierra sin necesidad de llamar explícitamente al método close(). Este método también significa que si se produce alguna excepción mientras se trabaja con el archivo, éste se cerrará antes de que salga el programa. Este es el método preferido para trabajar con archivos.

#### ***Leer todas las líneas a la vez***

En lugar de leer las líneas una a una para construir una lista, podemos utilizar el método readlines() para leer todas las líneas a la vez en una lista. El resultado es una función más compacta:

def read\_data(path):  
with open(path) as f:  
➊ lines = f.readlines()  
numbers = [float(n) for n in lines]  
return numbers

Leemos todas las líneas del archivo en una lista utilizando el método readlines() en ➊. A continuación, convertimos cada uno de los elementos de la lista en un número de coma flotante utilizando la función float() y la comprensión de listas. Por último, devolvemos la lista numbers.

#### ***Especificar el nombre del archivo como entrada***

La función read\_data() toma como argumento la ruta del archivo. Si tu programa te permite especificar el nombre del archivo como entrada, esta función debería funcionar para cualquier archivo, siempre que éste contenga datos que esperemos leer. Aquí tienes un ejemplo:

if \_\_name\_\_=='\_\_main\_\_':  
data\_file = input('Enter the path of the file: ')  
data = read\_data(data\_file)  
print(data)

Una vez que hayas añadido este código al final de la función read\_data() y la ejecutes, te pedirá que introduzcas la ruta al archivo. A continuación, imprimirá los números que lea del archivo:

Enter the path of the file /home/amit/work/mydata.txt  
[100.0,60.0,70.0,900.0,100.0,200.0,500.0,500.0,503.0,600.0,1000.0,1200.0]

#### ***Tratamiento de errores en la lectura de archivos***

Hay un par de cosas que pueden ir mal al leer archivos: (1) el archivo no se puede leer, o (2) los datos del archivo no están en el formato esperado. Aquí tienes un ejemplo de lo que ocurre cuando no se puede leer un archivo:

Enter the path of the file: /home/amit/work/mydata2.txt  
Traceback (most recent call last):  
File "read\_file.py", line 11, in <module>  
data = read\_data(data\_file)  
File "read\_file.py", line 4, in read\_data  
with open(path) as f:  
FileNotFoundError: [Errno 2] No such file or directory: '/home/amit/work/  
mydata2.txt'

Como he introducido una ruta de archivo que no existe, se lanza la excepción FileNotFoundError cuando intentamos abrir el archivo. Podemos hacer que el programa muestre un mensaje de error fácil de usar modificando nuestra función read\_data() de la siguiente manera:

def read\_data(path):  
numbers = []  
try:  
with open(path) as f:  
for line in f:  
numbers.append(float(line))  
except FileNotFoundError:  
print('File not found')  
return numbers

Ahora, cuando especifiques una ruta de archivo inexistente, recibirás en su lugar un mensaje de error:

Enter the path of the file: /home/amit/work/mydata2.txt  
File not found

La segunda fuente de errores puede ser que los datos del archivo no sean los que tu programa espera leer. Por ejemplo, considera un archivo que tenga lo siguiente:

10  
20  
3o  
1/5  
5.6

La tercera línea de este archivo no es convertible a un número de coma flotante porque contiene la letra o en lugar del número 0, y la cuarta línea consiste en 1/5, una fracción en forma de cadena, que float() no puede manejar.

Si proporcionas este archivo de datos al programa anterior, producirá el siguiente error:

Enter the path of the file: bad\_data.txt  
Traceback (most recent call last):  
File "read\_file.py", line 13, in <module>  
data = read\_data(data\_file)  
File "read\_file.py", line 6, in read\_data  
numbers.append(float(line))  
ValueError: could not convert string to float: '3o\n'

La tercera línea del archivo es 3o, no el número 30, así que cuando intentamos convertirlo en un número de coma flotante, el resultado es ValueError. Hay dos enfoques que puedes adoptar cuando hay datos de este tipo en un archivo. La primera es informar del error y salir del programa. La función read\_data() modificada quedaría como sigue:

def read\_data(path):  
numbers = []  
try:  
with open(path) as f:  
for line in f:  
➊ try:  
➋ n = float(line)  
except ValueError:  
print('Bad data: {0}'.format(line))  
➌ break  
➍ numbers.append(n)  
except FileNotFoundError:  
print('File not found')  
return numbers

Insertamos otro bloque try...except en la función que comienza en ➊, y convertimos la línea en un número de coma flotante en ➋. Si el programa lanza la excepción ValueError, imprimimos un mensaje de error con la línea en cuestión y salimos del bucle for utilizando break en ➌. A continuación, el programa deja de leer el archivo. La lista devuelta, numbers, contiene todos los datos que se leyeron correctamente antes de encontrar los datos erróneos. Si no hay error, añadimos el número de coma flotante a la lista numbers en ➍.

Ahora, cuando proporciones el archivo *datos\_malos.txt* al programa, sólo leerá las dos primeras líneas, mostrará el mensaje de error y saldrá:

Enter the path of the file: bad\_data.txt  
Bad data: 3o  
  
[10.0, 20.0]

Devolver datos parciales puede no ser deseable, así que podríamos sustituir la sentencia break en ➌ por return y no se devolvería ningún dato.

El segundo enfoque consiste en ignorar el error y continuar con el resto del archivo. Aquí tienes una función read\_data() modificada que hace esto:

def read\_data(path):  
numbers = []  
try:  
with open(path) as f:  
for line in f:  
try:  
n = float(line)  
except ValueError:  
print('Bad data: {0}'.format(line))  
➊ continue  
numbers.append(n)  
except FileNotFoundError:  
print('File not found')  
return numbers

El único cambio aquí es que en lugar de salir del bucle for, simplemente continuamos con la siguiente iteración utilizando la sentencia continue en ➊. La salida del programa es ahora la siguiente:

Bad data: 3o  
  
Bad data: 1/5  
  
[10.0, 20.0, 5.6]

La aplicación específica en la que estés leyendo el archivo determinará cuál de los enfoques anteriores quieres adoptar para tratar los datos erróneos.

### **Reutilizar el código**

A lo largo de este libro, hemos utilizado clases y funciones que formaban parte de la biblioteca estándar de Python o que estaban disponibles tras instalar paquetes de terceros, como matplotlib y SymPy. Ahora veremos un ejemplo rápido de cómo podemos importar nuestros propios programas en otros programas.

Consideremos la función find\_corr\_x\_y() que escribimos en "[Cálculo de la correlación entre dos conjuntos de datos](ch03.html#ch03lev1sec05)" en [la página 75](ch03.html#page_75). Crearemos un archivo aparte, *correlación.py*, que sólo contiene la definición de la función:

'''  
Function to calculate the linear correlation coefficient  
'''  
  
def find\_corr\_x\_y(x,y):  
# Size of each set  
n = len(x)  
  
# Find the sum of the products  
prod=[]  
for xi,yi in zip(x,y):  
prod.append(xi\*yi)  
  
sum\_prod\_x\_y = sum(prod)  
sum\_x = sum(x)  
sum\_y = sum(y)  
squared\_sum\_x = sum\_x\*\*2  
squared\_sum\_y = sum\_y\*\*2  
  
x\_square=[]  
for xi in x:  
x\_square.append(xi\*\*2)  
x\_square\_sum = sum(x\_square)  
  
y\_square=[]  
for yi in y:  
y\_square.append(yi\*\*2)  
y\_square\_sum = sum(y\_square)  
  
numerator = n\*sum\_prod\_x\_y - sum\_x\*sum\_y  
denominator\_term1 = n\*x\_square\_sum - squared\_sum\_x  
denominator\_term2 = n\*y\_square\_sum - squared\_sum\_y  
denominator = (denominator\_term1\*denominator\_term2)\*\*0.5  
  
correlation = numerator/denominator  
  
return correlation

Sin la extensión de archivo *.py*, un archivo Python se denomina módulo. Suele reservarse para archivos que definen clases y funciones que se utilizarán en otros programas. El siguiente programa importa la función find\_corr\_x\_y() del módulo de correlación que acabamos de definir:

from correlation import find\_corr\_x\_y  
if \_\_name\_\_ == '\_\_main\_\_':  
high\_school\_math = [83, 85, 84, 96, 94, 86, 87, 97, 97, 85]  
college\_admission = [85, 87, 86, 97, 96, 88, 89, 98, 98, 87]  
corr = find\_corr\_x\_y(high\_school\_math, college\_admission)  
print('Correlation coefficient: {0}'.format(corr))

Este programa encuentra la correlación entre las notas de matemáticas del instituto y las puntuaciones de admisión a la universidad de los estudiantes que consideramos en [la Tabla 3-3](ch03.html#ch3tab3) de la [página 80](ch03.html#page_80). Importamos la función find\_corr\_x\_y() del módulo de correlación, creamos las listas que representan los dos conjuntos de notas y llamamos a la función find\_corr\_x\_y() con las dos listas como argumentos. Cuando ejecutes el programa, imprimirá el coeficiente de correlación. Ten en cuenta que los dos archivos deben estar en el mismo directorio, estrictamente para simplificar las cosas.