

## CONCEPTOS BÁSICOS DE PYTHON



El lenguaje de programación Python tiene una amplia gama de construcciones sintácticas, funciones de biblioteca estándar y características de entorno de desarrollo interactivo. Afortunadamente, puedes ignorar la mayor parte de eso; solo necesitas aprender lo suficiente para escribir algunos pequeños programas útiles.

Sin embargo, deberá aprender algunos conceptos básicos de programación antes de poder hacer cualquier cosa. Como un mago en formación, puede pensar que estos conceptos parecen arcanos y tediosos, pero con algo de conocimiento y práctica, podrá controlar su computadora como una varita mágica y realizar hazañas increíbles.

Este capítulo tiene algunos ejemplos que lo animan a escribir en el *shell interactivo*, también llamado *REPL* (bucle de lectura-evaluación-impresión), que le permite ejecutar *instrucciones* de Python una a la vez y le muestra los resultados al instante. El uso del shell interactivo es excelente para aprender lo que hacen las instrucciones básicas de Python, así que pruébelo mientras sigue la lectura. Recordará las cosas que haga mucho mejor que las cosas que solo lea.

### Introducción de expresiones en el shell interactivo

Puede ejecutar el shell interactivo iniciando el editor Mu, que debería haber descargado al leer las instrucciones de configuración en el Prefacio. En Windows, abra el menú Inicio, escriba "Mu" y abra la aplicación Mu. En macOS, abra su carpeta Aplicaciones y haga doble clic en **Mu**. Haga clic en el botón **Nuevo** y guarde un archivo vacío como *blank.py*. Cuando ejecute este archivo en blanco haciendo clic en el botón **Ejecutar** o presionando F5, se abrirá el shell interactivo, que se abrirá como un nuevo panel que se abre en la parte inferior de la ventana del editor Mu. Debería ver un mensaje `>>>` en el shell interactivo.

Ingresa `2 + 2` en el indicador para que Python realice algunos cálculos matemáticos simples. La ventana Mu ahora debería verse así:

```
>>> 2 + 2
4
>>>
```

En Python,  $2 + 2$  se denomina *expresión*, que es el tipo de instrucción de programación más básico del lenguaje. Las expresiones constan de *valores* (como  $2$ ) y *operadores* (como  $+$ ), y siempre pueden *evaluarse* (es decir, reducirse) a un único valor. Eso significa que puedes usar expresiones en cualquier parte del código Python en el que también puedas usar un valor.

En el ejemplo anterior,  $2 + 2$  se evalúa hasta obtener un único valor,  $4$ . Un único valor sin operadores también se considera una expresión, aunque se evalúa solo como sí mismo, como se muestra aquí:

```
>>> 2
2
```

## ¡LOS ERRORES ESTÁN BIEN!

Los programas se bloquearán si contienen código que la computadora no puede entender, lo que hará que Python muestre un mensaje de error. Sin embargo, un mensaje de error no dañará su computadora, así que no tema cometer errores. Un *bloqueo* simplemente significa que el programa dejó de ejecutarse inesperadamente.

Si desea obtener más información sobre un error, puede buscar el texto exacto del mensaje de error en línea para obtener más información. También puede consultar los recursos en <https://nostarch.com/automatestuff2/> para ver una lista de mensajes de error comunes de Python y sus significados.

También puedes usar muchos otros operadores en expresiones de Python. Por ejemplo, [la Tabla 1-1](#) enumera todos los operadores matemáticos de Python.

**Tabla 1-1:** Operadores matemáticos de mayor a menor precedencia

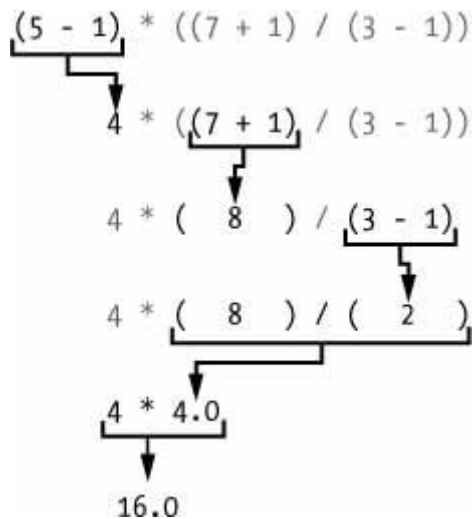
Operador Operación		Ejemplo Se evalúa como . . .	
**	Exponente	$2 ** 3$	8
%	Módulo/resto	$22 \% 8$	6
//	División de enteros/cociente de base	$22 // 8$	2
/	División	$22 / 8$	2,75
*	Multipliación	$3 * 5$	15

Operador	Operación	Ejemplo	Se evalúa como . . .
-	Sustracción	5 - 2	3
+	Suma	2 + 2	4

El *orden de operaciones* (también llamado *precedencia*) de los operadores matemáticos de Python es similar al de las matemáticas. El operador **\*\*** se evalúa primero; los operadores **\***, **/**, **//** y **%** se evalúan a continuación, de izquierda a derecha; y los operadores **+** y **-** se evalúan al final (también de izquierda a derecha). Puede utilizar paréntesis para anular la precedencia habitual si lo necesita. Los espacios en blanco entre los operadores y los valores no importan para Python (excepto la sangría al principio de la línea), pero un solo espacio es una convención. Ingrese las siguientes expresiones en el shell interactivo:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2 + 2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16,0
```

En cada caso, el programador debe ingresar la expresión, pero Python se encarga de la parte difícil de evaluarla hasta obtener un único valor. Python seguirá evaluando partes de la expresión hasta que se convierta en un único valor, como se muestra aquí:



Estas reglas para combinar operadores y valores para formar expresiones son una parte fundamental de Python como lenguaje de programación, al igual que las reglas gramaticales que nos ayudan a comunicarnos. A continuación, se muestra un ejemplo:

**Esta es una oración en inglés gramaticalmente correcta.**

**Esta oración gramaticalmente no es correcta en inglés.**

La segunda línea es difícil de analizar porque no sigue las reglas del inglés. De manera similar, si ingresa una instrucción incorrecta de Python, Python no podrá entenderla y mostrará un mensaje de error `SyntaxError`, como se muestra aquí:

```
>>> 5 +
      Archivo "<stdin>", línea 1
      5 +
      ^
SyntaxError: sintaxis no válida

>>> 42 + 5 + * 2
      Archivo "<stdin>", línea 1
      42 + 5 + * 2
      ^
SyntaxError: sintaxis no válida
```

Siempre puedes probar si una instrucción funciona ingresándola en el shell interactivo. No te preocupes por dañar la computadora: lo peor que podría pasar es que Python responda con un mensaje de error. Los desarrolladores de software profesionales reciben mensajes de error mientras escriben código todo el tiempo.

### **Los tipos de datos enteros, de punto flotante y de cadena**

Recuerde que las expresiones son simplemente valores combinados con operadores y siempre se evalúan hasta obtener un único valor. Un *tipo de datos* es

una categoría de valores y cada valor pertenece exactamente a un tipo de datos. Los tipos de datos más comunes en Python se enumeran en [la Tabla 1-2](#) . Los valores -2 y 30 , por ejemplo, se dice que son valores *enteros* . El tipo de datos entero (o *int* ) indica valores que son números enteros. Los números con un punto decimal, como 3.14 , se denominan *números de punto flotante* (o *floats* ). Tenga en cuenta que, aunque el valor 42 es un entero, el valor 42.0 sería un número de punto flotante.

**Tabla 1-2:** Tipos de datos comunes

Tipo de datos	Ejemplos
Números enteros	-2 , -1 , 0 , 1 , 2 , 3 , 4 , 5
Números de punto flotante	-1,25 , -1,0 , -0,5 , 0,0 , 0,5 , 1,0 , 1,25
Instrumentos de cuerda	'a' , 'aa' , 'aaa' , '¡Hola!' , '11 gatos'

Los programas Python también pueden tener valores de texto llamados *cadenas* o *strs* (pronunciado “stirs”). Siempre rodee su cadena entre comillas simples ( ' ) (como en 'Hola' o '¡Adiós mundo cruel!' ) para que Python sepa dónde comienza y termina la cadena. Incluso puede tener una cadena sin caracteres, " , llamada *cadena en blanco* o *cadena vacía* . Las cadenas se explican con mayor detalle en [el Capítulo 4](#) .

Si alguna vez ve el mensaje de error `SyntaxError: EOL` mientras escanea un literal de cadena , probablemente olvidó el carácter de comilla simple final al final de la cadena, como en este ejemplo:

```
>>> '¡Hola, mundo!
SyntaxError: EOL al escanear literal de cadena
```

### Concatenación y replicación de cadenas

El significado de un operador puede cambiar según los tipos de datos de los valores que se encuentran junto a él. Por ejemplo, + es el operador de suma cuando opera con dos números enteros o valores de punto flotante. Sin embargo, cuando + se utiliza con dos valores de cadena, une las cadenas como operador *de concatenación de cadenas* . Ingrese lo siguiente en el shell interactivo:

```
>>> 'Alicia' + 'Bob'
'AliciaBob'
```

La expresión se evalúa como un único valor de cadena nuevo que combina el texto de las dos cadenas. Sin embargo, si intenta utilizar el operador + en una cadena y un valor entero, Python no sabrá cómo manejar esto y mostrará un mensaje de error.

```
>>> 'Alice' + 42
```

Traceback (última llamada reciente):

```
Archivo "<pyshell#0>", línea 1, en <módulo>
```

```
'Alice' + 42
```

TypeError: solo se puede concatenar str (no "int") con str

El mensaje de error can only concatenate str (not "int") to str significa que Python pensó que estabas intentando concatenar un entero a la cadena 'Alice' . Tu código tendrá que convertir explícitamente el entero a una cadena porque Python no puede hacer esto automáticamente. (La conversión de tipos de datos se explicará en “ [Dissección de su programa](#) ” en [la página 13](#) cuando hablemos de las funciones str() , int() y float() ).

El operador \* multiplica dos valores enteros o de punto flotante. Pero cuando se utiliza en un valor de cadena y en un valor entero, se convierte en el operador *de replicación de cadena* . Introduzca una cadena multiplicada por un número en el shell interactivo para ver esto en acción.

```
>>> 'Alicia' * 5
```

```
'AliciaAliciaAliciaAlicia'
```

La expresión se evalúa como un único valor de cadena que repite la cadena original una cantidad de veces igual al valor entero. La replicación de cadenas es un truco útil, pero no se utiliza con tanta frecuencia como la concatenación de cadenas.

El operador \* se puede utilizar con solo dos valores numéricos (para multiplicación) o con un valor de cadena y un valor entero (para replicación de cadenas). De lo contrario, Python solo mostrará un mensaje de error como el siguiente:

```
>>> 'Alice' * 'Bob'
```

Traceback (última llamada reciente):

```
Archivo "<pyshell#32>", línea 1, en <módulo>
```

```
'Alice' * 'Bob'
```

TypeError: no se puede multiplicar la secuencia por un valor que no sea int del tipo 'str'

```
>>> 'Alice' * 5.0
```

Traceback (última llamada reciente):

```
Archivo "<pyshell#33>", línea 1, en <módulo>
```

```
'Alice' * 5.0
```

TypeError: no se puede multiplicar la secuencia por un valor que no sea int del tipo 'float'

Tiene sentido que Python no entienda estas expresiones: no se pueden multiplicar dos palabras y es difícil replicar una cadena arbitraria un número fraccionario de veces.

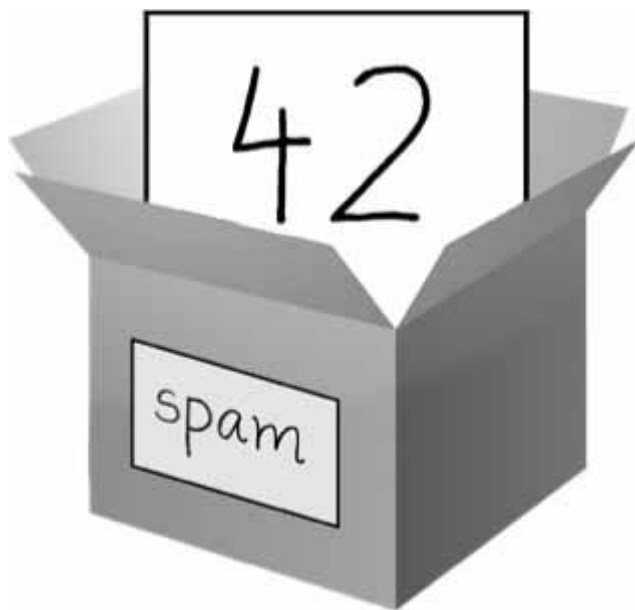
## Almacenamiento de valores en variables

Una *variable* es como una caja en la memoria de la computadora donde se puede almacenar un único valor. Si desea utilizar el resultado de una expresión evaluada más adelante en su programa, puede guardarlo dentro de una variable.

### Declaraciones de asignación

Almacenará valores en variables con una *declaración de asignación* . Una declaración de asignación consta de un nombre de variable, un signo igual (llamado *operador de asignación* ) y el valor que se almacenará. Si ingresa la declaración de asignación `spam = 42` , entonces una variable llamada `spam` tendrá el valor entero 42 almacenado en ella.

Piense en una variable como un cuadro etiquetado en el que se coloca un valor, como en [la Figura 1-1](#) .



*Figura 1-1: `spam = 42` es como decirle al programa: “La variable `spam` ahora tiene el valor entero 42”.*

Por ejemplo, ingrese lo siguiente en el shell interactivo:

```
❶ >>> spam = 40
    >>> spam
    40
    >>> huevos = 2
❷ >>> spam + huevos
```

```
42
>>> spam + huevos + spam
82
❸ >>> spam = spam + 2
>>> spam
42
```

Una variable se *inicializa* (o se crea) la primera vez que se almacena un valor en ella ❶ . Después de eso, puedes usarla en expresiones con otras variables y valores ❷ . Cuando se le asigna un nuevo valor a una variable ❸ , se olvida el valor anterior, por lo que spam se evaluó como 42 en lugar de 40 al final del ejemplo. Esto se llama *sobrescribir* la variable. Introduce el siguiente código en el shell interactivo para intentar sobrescribir una cadena:

```
>>> spam = 'Hola'
>>> spam
'Hola'
>>> spam = 'Adiós'
>>> spam
'Adiós'
```

Al igual que el cuadro de [la Figura 1-2](#) , la variable spam en este ejemplo almacena 'Hola' hasta que reemplace la cadena con 'Adiós' .



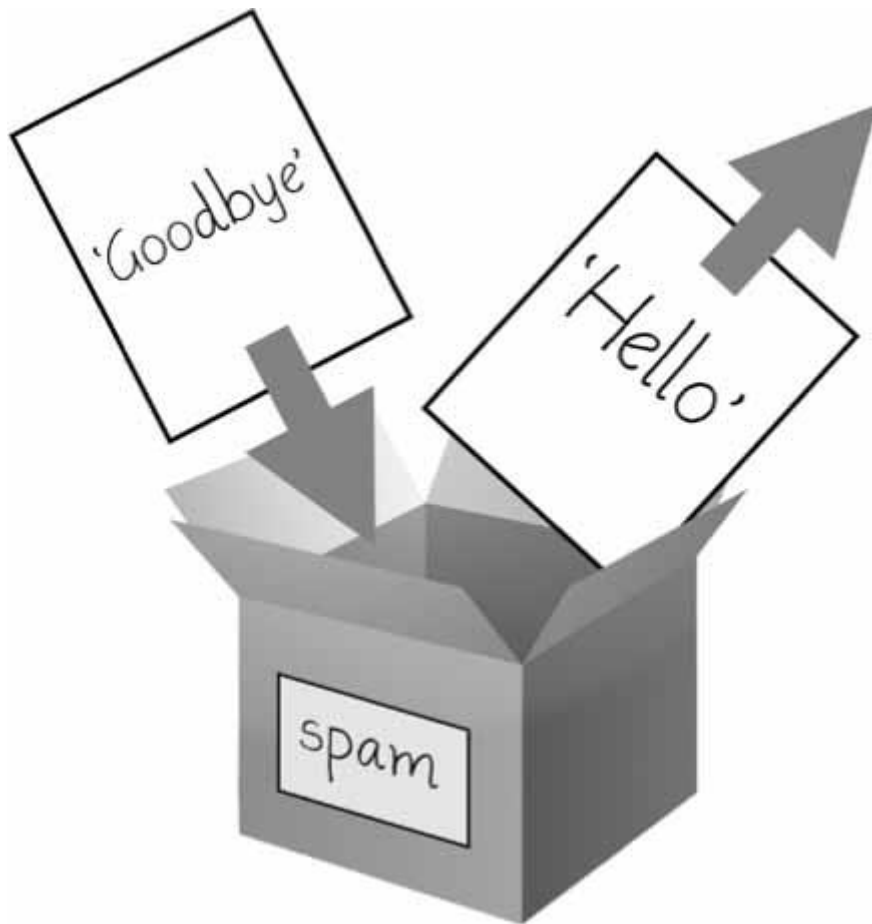


Figura 1-2: Cuando se asigna un nuevo valor a una variable, el anterior se olvida.

### **Nombres de variables**

Un buen nombre de variable describe los datos que contiene. Imagina que te mudas a una nueva casa y etiquetas todas tus cajas de mudanza como *Stuff* . ¡Nunca encontrarías nada! La mayoría de los ejemplos de este libro (y la documentación de Python) utilizan nombres de variables genéricos como `spam` , `eggs` y `bacon` , que provienen del sketch “Spam” de Monty Python. Pero en tus programas, un nombre descriptivo ayudará a que tu código sea más legible.

Aunque puede nombrar sus variables prácticamente de cualquier forma, Python tiene algunas restricciones de nombres. [La Tabla 1-3](#) tiene ejemplos de nombres de variables legales. Puede nombrar una variable de cualquier forma siempre que cumpla con las siguientes tres reglas:

- Sólo puede ser una palabra sin espacios.
- Solo puede utilizar letras, números y el carácter de subrayado ( `_` ).
- No puede comenzar con un número.

**Tabla 1-3:** Nombres de variables válidos y no válidos

Nombres de variables válidos	Nombres de variables no válidos
saldo actual	saldo actual (no se permiten guiones)
Saldo actual	Saldo actual (no se permiten espacios)
cuenta4	4cuenta (no puede comenzar con un número)
_42	42 (no puede comenzar con un número)
SUMA_TOTAL	TOTAL_\$UM ( no se permiten caracteres especiales como \$ )
Hola	'hola' ( no se permiten caracteres especiales como ' )

Los nombres de las variables distinguen entre mayúsculas y minúsculas, lo que significa que spam , SPAM , Spam y sPaM son cuatro variables diferentes. Aunque Spam es una variable válida que puedes usar en un programa, es una convención de Python comenzar las variables con una letra minúscula.

Este libro utiliza *el formato camelcase* para los nombres de las variables en lugar de guiones bajos; es decir, las variables se ven así en lugar de como se ven . Algunos programadores experimentados pueden señalar que el estilo de código oficial de Python, PEP 8, dice que se deben utilizar guiones bajos. Yo prefiero el formato camelcase sin remordimientos y señalo la sección “Una consistencia tonta es el duende de las mentes pequeñas” en el propio PEP 8:

La coherencia con la guía de estilo es importante, pero lo más importante es saber cuándo ser incoherente: a veces la guía de estilo simplemente no se aplica. En caso de duda, utilice su mejor criterio.

## Tu primer programa

Si bien el shell interactivo es útil para ejecutar instrucciones de Python de a una por vez, para escribir programas Python completos, deberá escribir las instrucciones en el editor de archivos. El *editor de archivos* es similar a los editores de texto como el Bloc de notas o TextMate, pero tiene algunas funciones específicas para ingresar código fuente. Para abrir un nuevo archivo en Mu, haga clic en el botón **Nuevo** en la fila superior.

La ventana que aparece debe contener un cursor esperando su entrada, pero es diferente del shell interactivo, que ejecuta instrucciones de Python. Tan pronto como presione ENTER , el editor de archivos le permitirá escribir muchas instrucciones, guardar el archivo y ejecutar el programa. A continuación, le indicamos cómo diferenciarlos:


- La ventana del shell interactivo siempre será la que tenga el mensaje >>> .
- La ventana del editor de archivos no tendrá el mensaje >>> .

¡Ahora es el momento de crear tu primer programa! Cuando se abra la ventana del editor de archivos, introduce lo siguiente:

```
❶ # Este programa dice hola y pregunta mi nombre.

❷ print('¡Hola, mundo!')
   print('¿Cuál es tu nombre?') # pregunta por su nombre
❸ myName = input()
❹ print('Es un gusto conocerte, ' + myName)
❺ print('La longitud de tu nombre es:')
   print(len(myName))
❻ print('¿Cuál es tu edad?') # pregunta por su edad
   myAge = input()
   print('Cumplirás ' + str(int(myAge) + 1) + ' en un año.')
```

Una vez que haya ingresado su código fuente, guárdelo para no tener que volver a escribirlo cada vez que inicie Mu. Haga clic en el botón **Guardar** , ingrese *hello.py* en el campo Nombre de archivo y luego haga clic en **Guardar** .

Debes guardar tus programas de vez en cuando mientras los escribes. De esa manera, si la computadora falla o sales de Mu accidentalmente, no perderás el código. Como atajo, puedes presionar CTRL -S en Windows y Linux o -S en macOS para guardar tu archivo.

Una vez que hayas guardado, ejecutemos nuestro programa. Pulsa la tecla **F5** . Tu programa debería ejecutarse en la ventana del shell interactivo. Recuerda que debes pulsar **F5** desde la ventana del editor de archivos, no desde la ventana del shell interactivo. Introduce tu nombre cuando el programa te lo pida. La salida del programa en el shell interactivo debería ser algo así:

```
Python 3.7.0b4 (v3.7.0b4:eb96c37699, 2 de mayo de 2018, 19:02:22) [MSC v.1913
de 64 bits
```

```
(AMD64)] en win32
```

```
Escribe "copyright", "credits" o "license()" para obtener más información.
```

```
>>> ===== REINICIAR
```

```
=====
```

```
>>>
```

```
¡Hola, mundo!
```

```
¿Cómo te llamas?
```

```
Al
```

```
Es un placer conocerte, Al
La longitud de tu nombre es:
2
¿Cuál es tu edad?
4
Cumplirás 5 en un año.
>>>
```

Cuando ya no quedan líneas de código para ejecutar, el programa Python *finaliza* ; es decir, deja de ejecutarse. (También se puede decir que el programa Python *sale* ).

Puede cerrar el editor de archivos haciendo clic en la X que se encuentra en la parte superior de la ventana. Para volver a cargar un programa guardado, seleccione **Archivo ▶ Abrir...** en el menú. Hágalo ahora y, en la ventana que aparece, elija **hello.py** y haga clic en el botón **Abrir** . El programa **hello.py** que guardó anteriormente debería abrirse en la ventana del editor de archivos.

Puedes ver la ejecución de un programa usando la herramienta de visualización Python Tutor en <http://pythontutor.com/> . Puedes ver la ejecución de este programa en particular en <https://autbor.com/hello.py/> . Haz clic en el botón de avance para avanzar por cada paso de la ejecución del programa. Podrás ver cómo cambian los valores de las variables y el resultado.

## Disecionando su programa

Con su nuevo programa abierto en el editor de archivos, hagamos un recorrido rápido por las instrucciones de Python que utiliza observando lo que hace cada línea de código.

### Comentarios

La siguiente línea se llama *comentario* .

❶ # Este programa dice hola y me pregunta mi nombre.

Python ignora los comentarios y puedes usarlos para escribir notas o recordarte lo que el código está intentando hacer. Cualquier texto del resto de la línea que sigue a un signo de almohadilla ( # ) es parte de un comentario.

A veces, los programadores colocan un # delante de una línea de código para eliminarla temporalmente mientras prueban un programa. Esto se llama *comentar* código y puede ser útil cuando intentas averiguar por qué un programa no funciona. Puedes eliminar el # más tarde cuando estés listo para volver a colocar la línea.

Python también ignora la línea en blanco después del comentario. Puedes agregar tantas líneas en blanco a tu programa como quieras. Esto puede hacer que tu código sea más fácil de leer, como los párrafos de un libro.

### ***La función print()***

La función print() muestra el valor de la cadena dentro de sus paréntesis en la pantalla.

```
❷ print('¡Hola, mundo!')  
   print('¿Cuál es tu nombre?') # pregunta por su nombre
```

La línea print('¡Hola, mundo!') significa “Imprime el texto en la cadena '¡Hola, mundo!' ”. Cuando Python ejecuta esta línea, dices que Python está *llamando* a la función print() y que el valor de la cadena se está *pasando* a la función. Un valor que se pasa a una llamada de función es un *argumento* . Observa que Las comillas no se imprimen en la pantalla. Simplemente marcan dónde comienza y termina la cadena; no son parte del valor de la cadena.

### **NOTA**

*También puedes usar esta función para poner una línea en blanco en la pantalla; simplemente llama a print() sin nada entre los paréntesis.*

Cuando escribes el nombre de una función, los paréntesis de apertura y cierre al final lo identifican como el nombre de una función. Por eso, en este libro verás print() en lugar de print . [El capítulo 3](#) describe las funciones con más detalle.

### ***La función input()***

La función input() espera a que el usuario escriba algún texto en el teclado y presione ENTER .

```
❸ miNombre = entrada()
```

Esta llamada de función evalúa una cadena igual al texto del usuario, y la línea de código asigna la variable myName a este valor de cadena.

Puede pensar en la llamada a la función input() como una expresión que evalúa cualquier cadena que el usuario haya ingresado. Si el usuario ingresó 'Al' , entonces la expresión evaluaría myName = 'Al' .

Si llama a input() y ve un mensaje de error, como NameError: name 'Al' is notdefined , el problema es que está ejecutando el código con Python 2 en lugar de Python 3.

### ***Impresión del nombre del usuario***

La siguiente llamada a `print()` en realidad contiene la expresión 'Es un placer conocerte' + `myName` entre paréntesis.

```
❷ print('Es un placer conocerte, ' + miNombre)
```

Recuerde que las expresiones siempre pueden evaluarse como un único valor.

Si 'Al' es el valor almacenado en `myName` en la línea ❸, entonces esta expresión evalúa como 'It's good to meet you, Al'. Este único valor de cadena se pasa luego a `print()`, que lo imprime en la pantalla.

### **La función `len()`**

Puede pasar a la función `len()` un valor de cadena (o una variable que contenga una cadena), y la función evalúa el valor entero de la cantidad de caracteres en esa cadena.

```
❸ print('La longitud de tu nombre es:')  
    print(len(myName))
```

Ingresa lo siguiente en el shell interactivo para probar esto:

```
>>> len('hola')  
5  
>>> len('Mi monstruo muy enérgico acaba de devorar nachos.')  
46  
>>> len('')  
0
```

Al igual que en estos ejemplos, `len(myName)` se evalúa como un entero. Luego se pasa a `print()` para que se muestre en la pantalla. La función `print()` le permite pasarle valores enteros o valores de cadena, pero observe el error que aparece cuando escribe lo siguiente en el shell interactivo:

```
>>> print('Tengo ' + 29 + ' años.')  
Traceback (última llamada reciente):  
  Archivo "<pyshell#6>", línea 1, en <módulo>  
    print('Tengo ' + 29 + ' años.')  
TypeError: solo se puede concatenar str (no "int") con str
```

La función `print()` no es la que provoca ese error, sino la expresión que intentaste pasar a `print()`. Obtienes el mismo mensaje de error si escribes la expresión en el shell interactivo por sí sola.

```
>>> 'Tengo ' + 29 + ' años.'  
Traceback (última llamada más reciente):  
  Archivo "<pyshell#7>", línea 1, en <módulo>
```

```
'Tengo ' + 29 + ' años.'
```

TypeError: solo se puede concatenar str (no "int") con str

Python da un error porque el operador + solo se puede usar para sumar dos números enteros o concatenar dos cadenas. No se puede sumar un número entero a una cadena, porque esto no es gramatical en Python. Puedes solucionar esto usando una versión de cadena del número entero, como se explica en la siguiente sección.

### ***Las funciones str(), int() y float()***

Si desea concatenar un entero como 29 con una cadena para pasar a print() , deberá obtener el valor '29' , que es la forma de cadena de 29 . A la función str() se le puede pasar un valor entero y se evaluará como una versión de valor de cadena del entero, de la siguiente manera:

```
>>> str(29)
'29'
>>> print('Tengo ' + str(29) + ' años.')
Tengo 29 años.
```

Como str(29) se evalúa como '29' , la expresión 'Tengo ' + str(29) + ' años.' se evalúa como 'Tengo ' + '29' + ' años.' , que a su vez se evalúa como 'Tengo 29 años.' . Este es el valor que se pasa a la función print() .

Las funciones str() , int() y float() se evaluarán como cadena, entero y punto flotante del valor que pases, respectivamente. Intenta convertir algunos valores en el shell interactivo con estas funciones y observa lo que sucede.

```
>>> str(0)
'0'
>>> str(-3,14)
'-3,14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1,25)
1
>>> int(1,99)
1
>>> float('3,14')
3,14
>>> float(10)
10,0
```

Los ejemplos anteriores llaman a las funciones `str()` , `int()` y `float()` y les pasan valores de otros tipos de datos para obtener una cadena, un entero o una forma de punto flotante de esos valores.

La función `str()` es útil cuando tienes un entero o un flotante que quieres concatenar a una cadena. La función `int()` también es útil si tienes un número como valor de cadena que quieres usar en matemáticas. Por ejemplo, la función `input()` siempre devuelve una cadena, incluso si el usuario ingresa un número. Introduce **`spam = input()`** en el shell interactivo e introduce **101** cuando espere tu texto.

```
>>> spam = entrada()
101
>>> spam
'101'
```

El valor almacenado dentro de `spam` no es el entero 101 sino la cadena '101' . Si desea realizar operaciones matemáticas utilizando el valor en `spam` , utilice la función `int()` para obtener la forma entera de `spam` y luego guárdela como el nuevo valor en `spam` .

```
>>> spam = int(spam)
>>> spam
101
```

Ahora deberías poder tratar la variable `spam` como un entero en lugar de una cadena.

```
>>> correo no deseado * 10 / 5
202.0
```

Tenga en cuenta que si pasa un valor a `int()` que no puede evaluar como un entero, Python mostrará un mensaje de error.

```
>>> int('99.99')
Traceback (última llamada más reciente):
  Archivo "<pyshell#18>", línea 1, en <módulo>
    int('99.99')
ValueError: literal inválido para int() con base 10: '99.99'
```

```
>>> int('twelve')
Traceback (última llamada más reciente):
  Archivo "<pyshell#19>", línea 1, en <módulo>
    int('twelve')
ValueError: literal inválido para int() con base 10: 'twelve'
```



La función `int()` también es útil si necesita redondear hacia abajo un número de punto flotante.

```
>>> entero(7.7)
7
>>> entero(7.7) + 1
8
```

Utilizaste las funciones `int()` y `str()` en las últimas tres líneas de tu programa para obtener un valor del tipo de datos apropiado para el código.

```
❹ print('¿Cuál es tu edad?') # pregunta por su edad
    myAge = input()
    print('Tendrás ' + str(int(myAge) + 1) + ' en un año.')
```

## EQUIVALENCIA DE TEXTO Y NÚMERO

Aunque el valor de cadena de un número se considera un valor completamente diferente de la versión entera o de punto flotante, un entero puede ser igual a un punto flotante.

```
>>> 42 == '42'
Falso
>>> 42 == 42.0
Verdadero
>>> 42.0 == 0042.000
Verdadero
```

Python hace esta distinción porque las cadenas son texto, mientras que los números enteros y los flotantes son números.

La variable `myAge` contiene el valor devuelto por `input()`. Debido a que la función `input()` siempre devuelve una cadena (incluso si el usuario escribió un número), puedes usar el código `int(myAge)` para devolver un valor entero de la cadena en `myAge`. Luego, este valor entero se suma a 1 en la expresión `int(myAge) + 1`.

El resultado de esta suma se pasa a la función `str()`: `str(int(myAge) + 1)`. El valor de cadena devuelto se concatena con las cadenas 'You will be ' y ' in a year.' para evaluarlo como un valor de cadena grande. Esta cadena grande finalmente se pasa a `print()` para que se muestre en la pantalla.

Digamos que el usuario ingresa la cadena '4' para `myAge`. La cadena '4' se convierte en un entero, por lo que puede agregarle uno. El resultado es 5. La función `str()` convierte el resultado nuevamente en una cadena, por lo que puede

concatenarlo con la segunda cadena, 'in a year.' , para crear el mensaje final. Estos pasos de evaluación se verían así:

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str(    4 + 1    ) + ' in a year.')
print('You will be ' + str(        5        ) + ' in a year.')
print('You will be ' +          '5'          + ' in a year.')
print('You will be 5'                        + ' in a year.')
print('You will be 5 in a year.')
```

## CONTROL DE FLUJO



Entonces, ya conoces los conceptos básicos de las instrucciones individuales y que un programa es solo una serie de instrucciones. Pero la verdadera fortaleza de la programación no es simplemente ejecutar una instrucción tras otra como una lista de tareas del fin de semana. En función de cómo se evalúan las expresiones, un programa puede decidir omitir instrucciones, repetirlas o elegir una de varias instrucciones para ejecutar. De hecho, casi nunca quieres que tus programas comiencen desde la primera línea de código y simplemente ejecuten cada línea, directamente hasta el final. *Las declaraciones de control de flujo* pueden decidir qué instrucciones de Python ejecutar bajo qué condiciones.

Estas instrucciones de control de flujo corresponden directamente a los símbolos de un diagrama de flujo, por lo que proporcionaré versiones de diagrama de flujo del código analizado en este capítulo. [La Figura 2-1](#) muestra un diagrama de flujo que muestra qué hacer si llueve. Siga la ruta que trazan las flechas desde el inicio hasta el final.

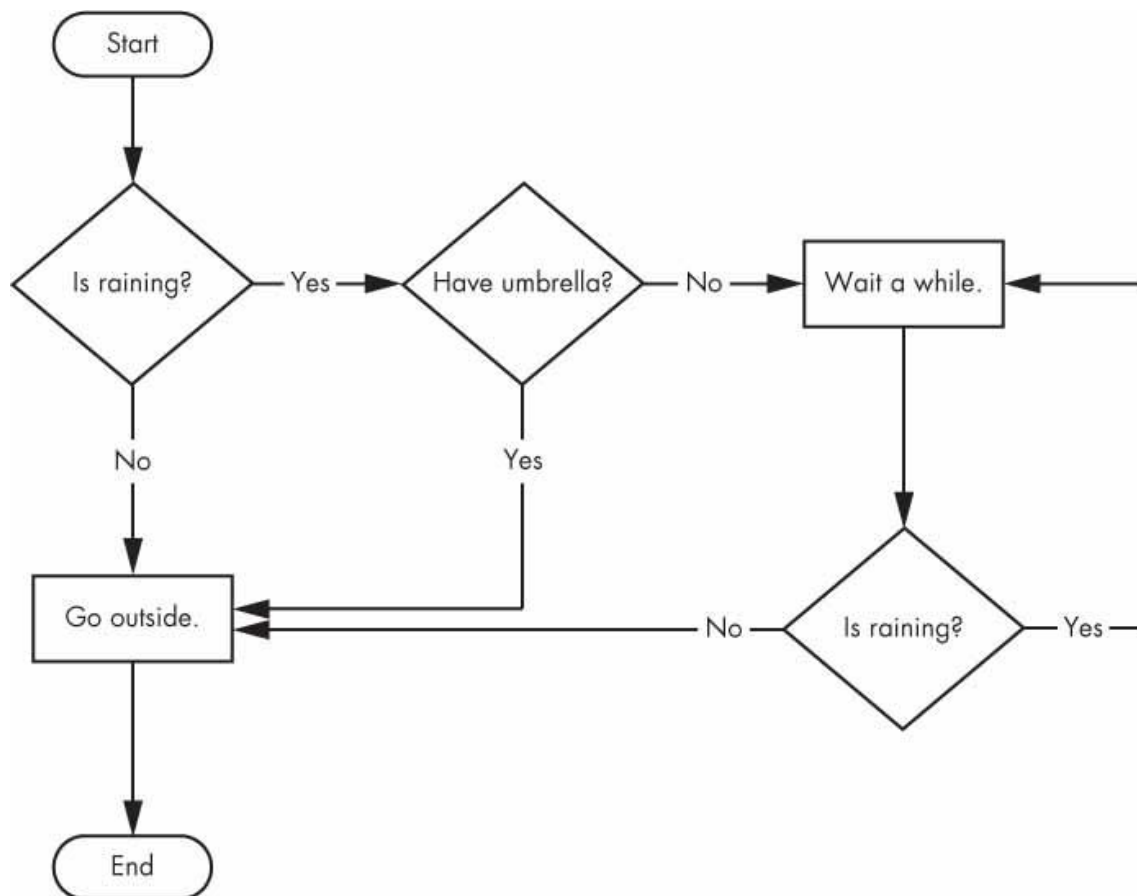


Figura 2-1: Un diagrama de flujo que le indica qué hacer si está lloviendo

En un diagrama de flujo, suele haber más de una forma de ir desde el principio hasta el final. Lo mismo ocurre con las líneas de código de un programa informático. Los diagramas de flujo representan estos puntos de ramificación con rombos, mientras que los demás pasos se representan con rectángulos. Los pasos de inicio y fin se representan con rectángulos redondeados.

Pero antes de aprender sobre las sentencias de control de flujo, primero debe aprender a representar esas opciones *de sí y no*, y debe comprender cómo escribir esos puntos de ramificación como código Python. Para ello, exploremos los valores booleanos, los operadores de comparación y los operadores booleanos.

### Valores booleanos

Mientras que los tipos de datos enteros, de punto flotante y de cadena tienen una cantidad ilimitada de valores posibles, el tipo de datos *booleano tiene solo dos valores*: True y False . (Boolean se escribe con mayúscula porque el tipo de datos recibe su nombre del matemático George Boole). Cuando se ingresan como código Python, los valores booleanos True y False no tienen las comillas que se colocan alrededor de las cadenas y siempre comienzan con una *T o F* mayúscula , con el resto de la palabra en minúscula. Ingrese lo siguiente en el shell interactivo.

(Algunas de estas instrucciones son intencionalmente incorrectas y harán que aparezcan mensajes de error).

❶ `>>> spam = True`

`>>> spam`

True

❷ `>>> true`

Traceback (última llamada reciente):

Archivo "<pyshell#2>", línea 1, en <módulo>

true

NameError: el nombre 'true' no está definido

❸ `>>> True = 2 + 2`

SyntaxError: no se puede asignar a la palabra clave

Al igual que cualquier otro valor, los valores booleanos se utilizan en expresiones y se pueden almacenar en variables ❶. Si no utilizas las mayúsculas y minúsculas adecuadas ❷ o intentas utilizar True y False para los nombres de las variables ❸, Python te mostrará un mensaje de error.

## Operadores de comparación

Los *operadores de comparación*, también llamados *operadores relacionales*, comparan dos valores y los evalúan hasta obtener un único valor booleano. [La Tabla 2-1](#) enumera los operadores de comparación.

**Tabla 2-1:** Operadores de comparación

Operador Significado	
<code>==</code>	Igual a
<code>!=</code>	No es igual a
<code>&lt;</code>	Menos que
<code>&gt;</code>	Más que
<code>&lt;=</code>	Menor o igual a
<code>&gt;=</code>	Mayor o igual a

Estos operadores se evalúan como Verdadero o Falso según los valores que les asigne. Probemos algunos operadores ahora, comenzando con `==` y `!=`.

`>>> 42 == 42`

Verdadero

`>>> 42 == 99`

Falso

```
>>> 2 != 3
```

Verdadero

```
>>> 2 != 2
```

Falso

Como es de esperar, == (igual a) se evalúa como Verdadero cuando los valores de ambos lados son iguales, y != (no igual a) se evalúa como Verdadero cuando los dos valores son diferentes. Los operadores == y != pueden funcionar con valores de cualquier tipo de datos.

```
>>> 'hola' == 'hola'
```

Verdadero

```
>>> 'hola' == 'Hola'
```

Falso

```
>>> 'perro' != 'gato'
```

Verdadero

```
>>> Verdadero == Verdadero
```

Verdadero

```
>>> Verdadero != Falso
```

Verdadero

```
>>> 42 == 42.0
```

Verdadero

```
❶ >>> 42 == '42'
```

Falso

Tenga en cuenta que un valor entero o de punto flotante siempre será distinto de un valor de cadena. La expresión `42 == '42'` ❶ se evalúa como Falso porque Python considera que el entero 42 es distinto de la cadena '42'.

Los operadores <, >, <= y >=, por otro lado, funcionan correctamente sólo con valores enteros y de punto flotante.

```
>>> 42 < 100
```

Verdadero

```
>>> 42 > 100
```

Falso

```
>>> 42 < 42
```

Falso

```
>>> recuentoDeHuevos = 42
```

```
❶ >>> recuentoDeHuevos <= 42
```

Verdadero

```
>>> miEdad = 29
```

```
❷ >>> miEdad >= 10
```

```
Verdadero
```

## LA DIFERENCIA ENTRE LOS OPERADORES == Y =

Es posible que hayas notado que el operador == (igual a) tiene dos signos iguales, mientras que el operador = (asignación) tiene solo un signo igual. Es fácil confundir estos dos operadores entre sí. Solo recuerda estos puntos:

- El operador == (igual a) pregunta si dos valores son iguales entre sí.
- El operador = (asignación) coloca el valor de la derecha en la variable de la izquierda.

Para ayudar a recordar cuál es cuál, observe que el operador == (igual a) consta de dos caracteres, al igual que el operador != (no igual a) consta de dos caracteres.

A menudo, utilizará operadores de comparación para comparar el valor de una variable con algún otro valor, como en los ejemplos `eggCount <= 42` ❶ y `myAge >= 10` ❷. (Después de todo, en lugar de ingresar 'dog' != 'cat' en su código, podría haber ingresado `True`). Verá más ejemplos de esto más adelante cuando aprenda sobre las declaraciones de control de flujo.

## Operadores booleanos

Los tres operadores booleanos ( `and` , `or` , and `not` ) se utilizan para comparar valores booleanos. Al igual que los operadores de comparación, evalúan estas expresiones hasta obtener un valor booleano. Exploreemos estos operadores en detalle, comenzando con el operador `and` .

### Operadores booleanos binarios

Los operadores `and` y `or` siempre toman dos valores booleanos (o expresiones), por lo que se consideran operadores *binarios* . El operador `and` evalúa una expresión como `True` si *ambos* valores booleanos son `True` ; de lo contrario, evalúa como `False` . Ingrese algunas expresiones usando `and` en el shell interactivo para verlo en acción.

```
>>> Verdadero y Verdadero
```

```
Verdadero
```

```
>>> Verdadero y Falso
```

```
Falso
```

Una *tabla de verdad* muestra todos los resultados posibles de un operador booleano. [La Tabla 2-2](#) es la tabla de verdad para el operador `and` .

**Tabla 2-2:** Tabla de verdad del operador `and`

Expresión	Se evalúa como . . .
Cierto y cierto	Verdadero
Verdadero y falso	FALSO
Falso y verdadero	FALSO
Falso y falso	FALSO

Por otra parte, el operador `or` evalúa una expresión como `True` si *cualquiera* de los dos valores booleanos es `True` . Si ambos son `False` , se evalúa como `False` .

>>> **Falso o Verdadero**

Verdadero

>>> **Falso o Falso**

Falso

Puede ver cada resultado posible del operador `or` en su tabla de verdad, que se muestra en [la Tabla 2-3](#) .

**Tabla 2-3:** Tabla de verdad del operador `or`

Expresión	Se evalúa como . . .
Verdadero o Verdadero	Verdadero
Verdadero o falso	Verdadero
Falso o verdadero	Verdadero
Falso o falso	FALSO

### ***El operador `no`***

A diferencia de `and` y `or` , el operador `not` opera solo sobre un valor booleano (o expresión). Esto lo convierte en un operador *unario* . El operador `not` simplemente evalúa el valor booleano opuesto.

>>> **no es verdadero**

Falso

❶ >>> **no no no no es verdadero Verdadero**

Verdadero

De forma muy similar al uso de negaciones dobles en el habla y la escritura, puedes anidar operadores `not` ❶ , aunque nunca hay ninguna razón para hacerlo en programas reales. [La Tabla 2-4](#) muestra la tabla de verdad para `not` .

**Tabla 2-4:** Tabla de verdad del operador `not`

---

**Expresión   Se evalúa como ...**

---

No es cierto FALSO

No es falso Verdadero

---

**Combinación de operadores booleanos y de comparación**

Dado que los operadores de comparación evalúan valores booleanos, puedes usarlos en expresiones con operadores booleanos.

Recuerde que los operadores `and`, `or`, and `not` se denominan operadores booleanos porque siempre operan sobre los valores booleanos `True` y `False`. Si bien expresiones como `4 < 5` no son valores booleanos, son expresiones que se evalúan en sentido descendente para obtener valores booleanos. Intente introducir algunas expresiones booleanas que utilicen operadores de comparación en el shell interactivo.

```
>>> (4 < 5) y (5 < 6)
```

Verdadero

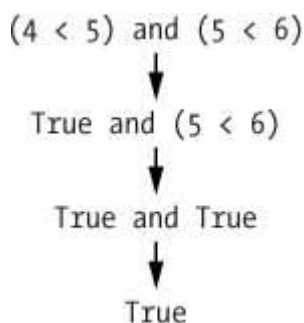
```
>>> (4 < 5) y (9 < 6)
```

Falso

```
>>> (1 == 2) o (2 == 2)
```

Verdadero

La computadora evaluará primero la expresión de la izquierda y luego evaluará la expresión de la derecha. Cuando conozca el valor booleano de cada una, luego evaluará toda la expresión hasta obtener un valor booleano. Puedes pensar en el proceso de evaluación de la computadora para `(4 < 5) y (5 < 6)` de la siguiente manera:



También puedes utilizar varios operadores booleanos en una expresión, junto con los operadores de comparación:

```
>>> 2 + 2 == 4 y no 2 + 2 == 5 y 2 * 2 == 2 + 2
```

Verdadero



Los operadores booleanos tienen un orden de operaciones, al igual que los operadores matemáticos. Después de evaluar los operadores matemáticos y de comparación, Python evalúa primero los operadores not , luego los operadores and y, por último, los operadores or .

## Elementos del control de flujo

Las sentencias de control de flujo suelen comenzar con una parte llamada *condición* y siempre van seguidas de un bloque de código llamado *cláusula* . Antes de aprender sobre las sentencias de control de flujo específicas de Python, explicaré qué son una condición y un bloque.

### Condiciones

Las expresiones booleanas que has visto hasta ahora podrían considerarse condiciones, que son lo mismo que expresiones; *condición* es solo un nombre más específico en el contexto de las sentencias de control de flujo. Las condiciones siempre se evalúan hasta un valor booleano, True o False . Una sentencia de control de flujo decide qué hacer en función de si su condición es True o False , y casi todas las sentencias de control de flujo utilizan una condición.

### Bloques de código

Las líneas de código Python se pueden agrupar en *bloques* . Puedes saber cuándo comienza y termina un bloque a partir de la sangría de las líneas de código. Existen tres reglas para los bloques.

- Los bloqueos comienzan cuando la sangría aumenta.
- Los bloques pueden contener otros bloques.
- Los bloques finalizan cuando la sangría disminuye a cero o a la sangría de un bloque contenedor.

Los bloques son más fáciles de entender mirando algún código sangrado, así que busquemos los bloques en parte de un pequeño programa de juego, que se muestra aquí:

```
nombre = 'Mary'
contraseña = 'swordfish'
si nombre == 'Mary':
    ❶ print('Hola, Mary')
    si contraseña == 'swordfish':
        ❷ print('Acceso concedido.')
```

de lo contrario:

```
❸ print('Contraseña incorrecta.')
```

Puedes ver la ejecución de este programa en <https://autbor.com/blocks/> . El primer bloque de código ❶ comienza en la línea `print('Hola, Mary')` y contiene todas las líneas posteriores. Dentro de este bloque hay otro bloque ❷ , que tiene solo una línea: `print('Acceso concedido.')` . El tercer bloque ❸ también tiene una línea: `print('Contraseña incorrecta.')` .

## Ejecución del programa

En el programa *hello.py* del capítulo anterior , Python comenzó a ejecutar instrucciones en la parte superior del programa y las fue bajando una tras otra. La *ejecución del programa* (o simplemente, *ejecución* ) es un término que se utiliza para referirse a la instrucción actual que se está ejecutando. Si imprime el código fuente en papel y coloca el dedo en cada línea a medida que se ejecuta, puede pensar en su dedo como la ejecución del programa.

Sin embargo, no todos los programas se ejecutan simplemente yendo hacia abajo. Si usas el dedo para recorrer un programa con instrucciones de control de flujo, probablemente te encontrarás saltando por el código fuente en función de las condiciones y probablemente te saltes cláusulas enteras.

## Declaraciones de control de flujo

Ahora, exploremos la parte más importante del control de flujo: las sentencias en sí. Las sentencias representan los diamantes que vio en el diagrama de flujo de [la Figura 2-1](#) y son las decisiones reales que tomarán sus programas.

### Declaraciones if

El tipo más común de sentencia de control de flujo es la sentencia `if` . La cláusula de una sentencia `if` (es decir, el bloque que sigue a la sentencia `if` ) se ejecutará si la condición de la sentencia es `True` . La cláusula se omite si la condición es `False` .

En términos sencillos, una declaración `if` podría leerse como “Si esta condición es verdadera, ejecute el código en la cláusula”. En Python, una declaración `if` consta de lo siguiente:

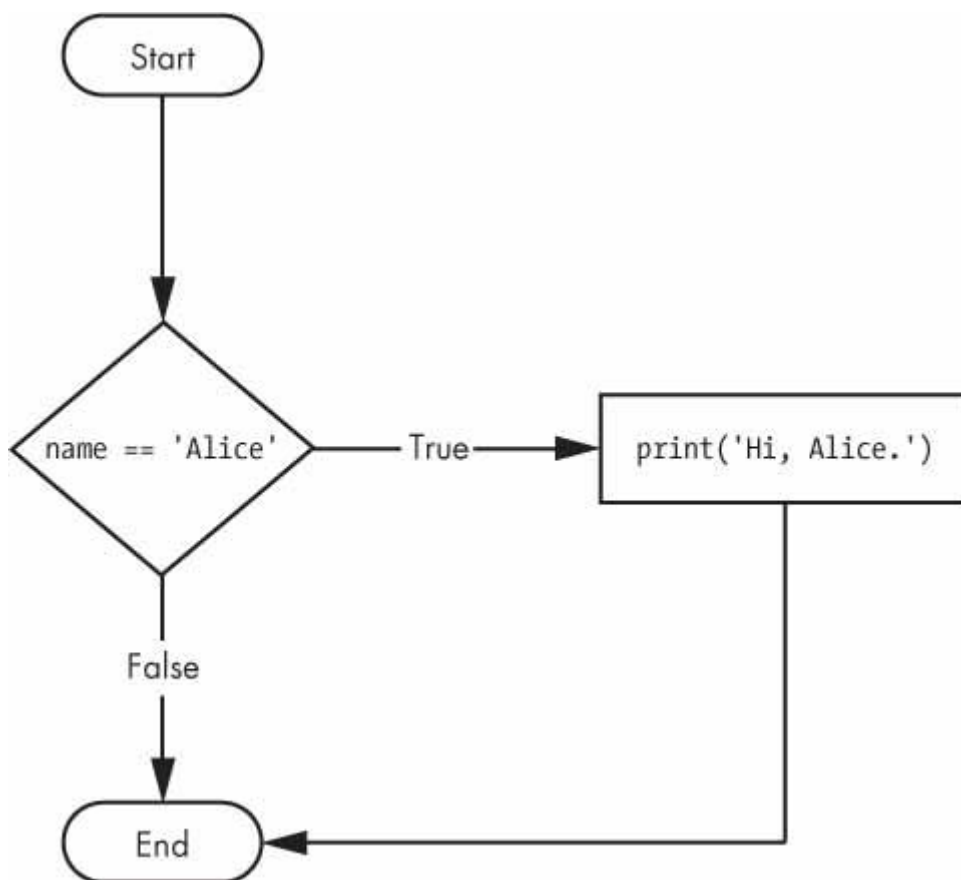
- La palabra clave `if`
- Una condición (es decir, una expresión que evalúa como Verdadero o Falso )
- Dos puntos

- A partir de la siguiente línea, un bloque de código sangrado (llamado cláusula if )

Por ejemplo, supongamos que tienes un código que verifica si el nombre de alguien es Alice. (Supongamos que al nombre se le asignó un valor anteriormente).

```
si nombre == 'Alice':  
    print('Hola, Alice.')
```

Todas las sentencias de control de flujo terminan con dos puntos y van seguidas de un nuevo bloque de código (la cláusula). La cláusula de esta sentencia if es el bloque con `print('Hi, Alice.')` . [La figura 2-2](#) muestra cómo se vería un diagrama de flujo de este código.



*Figura 2-2: Diagrama de flujo para una declaración if*

### **Declaraciones else**

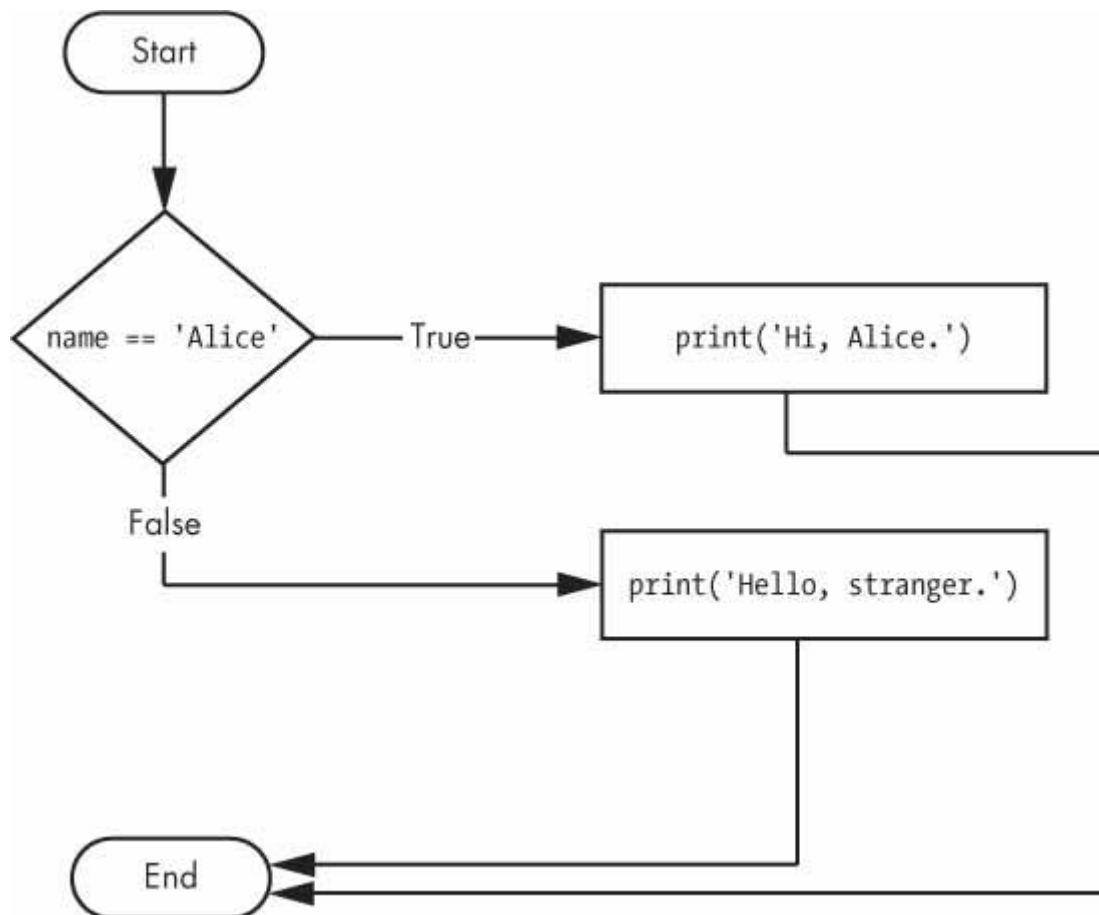
Una cláusula if puede ir seguida opcionalmente de una sentencia else . La cláusula else se ejecuta solo cuando la condición de la sentencia if es False . En lenguaje sencillo, una sentencia else podría leerse como “Si esta condición es verdadera, ejecuta este código. O bien, ejecuta ese código”. Una sentencia else no tiene una condición y, en el código, una sentencia else siempre consta de lo siguiente:

- La palabra clave else
- Dos puntos
- A partir de la siguiente línea, un bloque de código sangrado (llamado cláusula else )

Volviendo al ejemplo de Alice, veamos un código que utiliza una declaración else para ofrecer un saludo diferente si el nombre de la persona no es Alice.

```
si nombre == 'Alice':  
    print('Hola, Alice.')  
de lo contrario:  
    print('Hola, extraño.')
```

[La figura 2-3](#) muestra cómo se vería un diagrama de flujo de este código.



*Figura 2-3: Diagrama de flujo para una declaración else*

### **Declaraciones elif**

Si bien solo se ejecutará una de las cláusulas if o else , es posible que tenga un caso en el que desee que se ejecute una de las *muchas cláusulas posibles*. La

*declaración elif* es una declaración “else if” que siempre sigue a una declaración if u otra declaración elif . Proporciona otra condición que se verifica solo si todas las condiciones anteriores fueron False . En el código, una declaración elif siempre consta de lo siguiente:

- La palabra clave elif
- Una condición (es decir, una expresión que evalúa como Verdadero o Falso )
- Dos puntos
- A partir de la siguiente línea, un bloque de código sangrado (llamado cláusula elif )

Agreguemos un elif al verificador de nombres para ver esta declaración en acción.

```
si nombre == 'Alice':  
    print('Hola, Alice.')  
elif edad < 12:  
    print('No eres Alice, niña.')
```

Esta vez, verifica la edad de la persona y el programa le dirá algo diferente si es menor de 12 años. Puedes ver el diagrama de flujo para esto en [la Figura 2-4](#) .

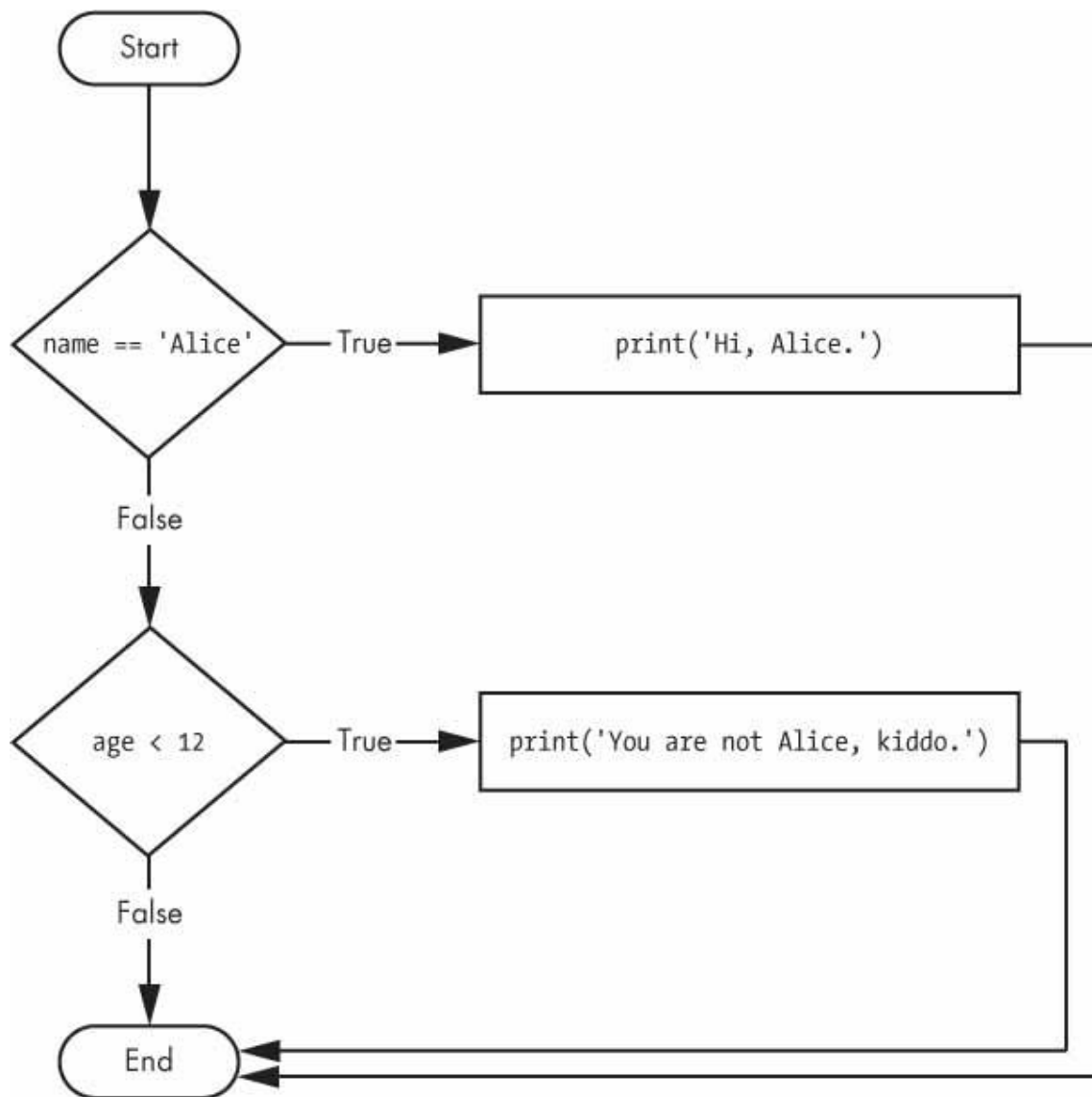


Figura 2-4: Diagrama de flujo para una declaración elif

La cláusula elif se ejecuta si `age < 12` es True y `name == 'Alice'` es False . Sin embargo, si ambas condiciones son False , entonces ambas cláusulas se omiten. No se garantiza que se ejecute al menos una de las cláusulas. Cuando hay una cadena de declaraciones elif , solo se ejecutará una o ninguna de las cláusulas. Una vez que se determina que una de las condiciones de las declaraciones es True , el resto de las cláusulas elif se omiten automáticamente. Por ejemplo, abra una nueva ventana del editor de archivos e ingrese el siguiente código, guardándolo como *vampire.py* :

```
nombre = 'Carol'
edad = 3000
si nombre == 'Alice':
    print('Hola, Alice.')
elif edad < 12:
```

```
    print('No eres Alice, pequeña.')  
elif edad > 2000:  
    print('A diferencia de ti, Alice no es una vampiresa inmortal y no muerta.')  
elif edad > 100:  
    print('Tú no eres Alice, abuelita.')
```

Puedes ver la ejecución de este programa en <https://autbor.com/vampire/> . Aquí, he añadido dos declaraciones elif más para que el verificador de nombres salude a una persona con diferentes respuestas según su edad . [La Figura 2-5](#) muestra el diagrama de flujo para esto.

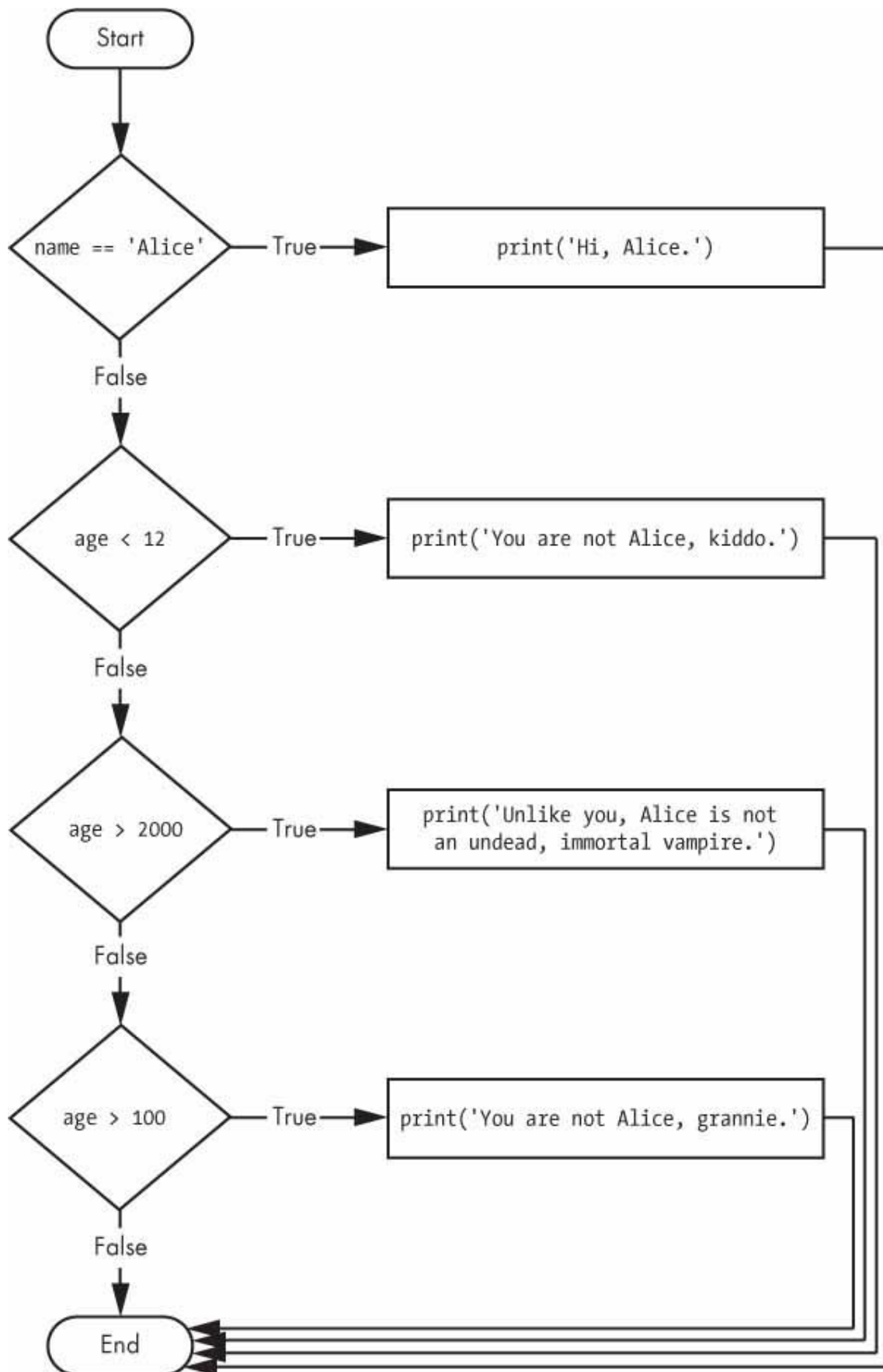




Figura 2-5: Diagrama de flujo para múltiples declaraciones elif en el programa *vampire.py*

Sin embargo, el orden de las sentencias elif sí importa. Reorganizémoslas para introducir un error. Recuerde que el resto de las cláusulas elif se omiten automáticamente una vez que se encuentra una condición True , por lo que si intercambia algunas de las cláusulas en *vampire.py* , se encontrará con un problema. Cambie el código para que se parezca al siguiente y guárdelo como *vampire2.py* :

```
nombre = 'Carol'
edad = 3000
si nombre == 'Alice':
    print('Hola, Alice.')
elif edad < 12:
    print('No eres Alice, niña.')
❶ elif edad > 100:
    print('No eres Alice, abuelita.')
elif edad > 2000:
    print('A diferencia de ti, Alice no es una vampiresa inmortal no muerta.')
```

Puedes ver la ejecución de este programa en <https://autbor.com/vampire2/> . Digamos que la variable age contiene el valor 3000 antes de que se ejecute este código. Podrías esperar que el código imprima la cadena 'A diferencia de ti, Alice no es una vampiresa inmortal no muerta.' . Sin embargo, debido a que la condición age > 100 es True (después de todo, 3000 es mayor que 100) ❶ , se imprime la cadena 'No eres Alice, abuelita.' y el resto de las declaraciones elif se omiten automáticamente. Recuerda que, como máximo, solo se ejecutará una de las cláusulas y, para las declaraciones elif , ¡el orden importa!

La figura 2-6 muestra el diagrama de flujo del código anterior. Observe cómo se intercambian los rombos para edad > 100 y edad > 2000 .

Opcionalmente, puede tener una sentencia else después de la última sentencia elif . En ese caso, se *garantiza* que se ejecutará al menos una (y solo una) de las cláusulas. Si las condiciones en cada sentencia if y elif son False , entonces se ejecuta la cláusula else . Por ejemplo, recreemos el programa Alice para usar las cláusulas if , elif y else .

```
nombre = 'Carol'
edad = 3000
si nombre == 'Alice':
    print('Hola, Alice.')
elif edad < 12:
```

```
print('No eres Alice, niña.')  
de lo contrario:  
    print('No eres Alice ni una niña.')
```

Puedes ver la ejecución de este programa en <https://autbor.com/littlekid/> . La [Figura 2-7](#) muestra el diagrama de flujo de este nuevo código, que guardaremos como *littleKid.py* .

En términos sencillos, este tipo de estructura de control de flujo sería “Si la primera condición es verdadera, haga esto. De lo contrario, si la segunda condición es verdadera, haga eso. De lo contrario, haga otra cosa”. Cuando use las declaraciones `if` , `elif` y `else` juntas, recuerde estas reglas sobre cómo ordenarlas para evitar errores como el de [la Figura 2-6](#) . Primero, siempre hay exactamente una declaración `if` . Cualquier declaración `if` es verdadera. Las instrucciones `elif` que necesita deben seguir a la instrucción `if` . En segundo lugar, si desea asegurarse de que se ejecute al menos una cláusula, cierre la estructura con una instrucción `else` .

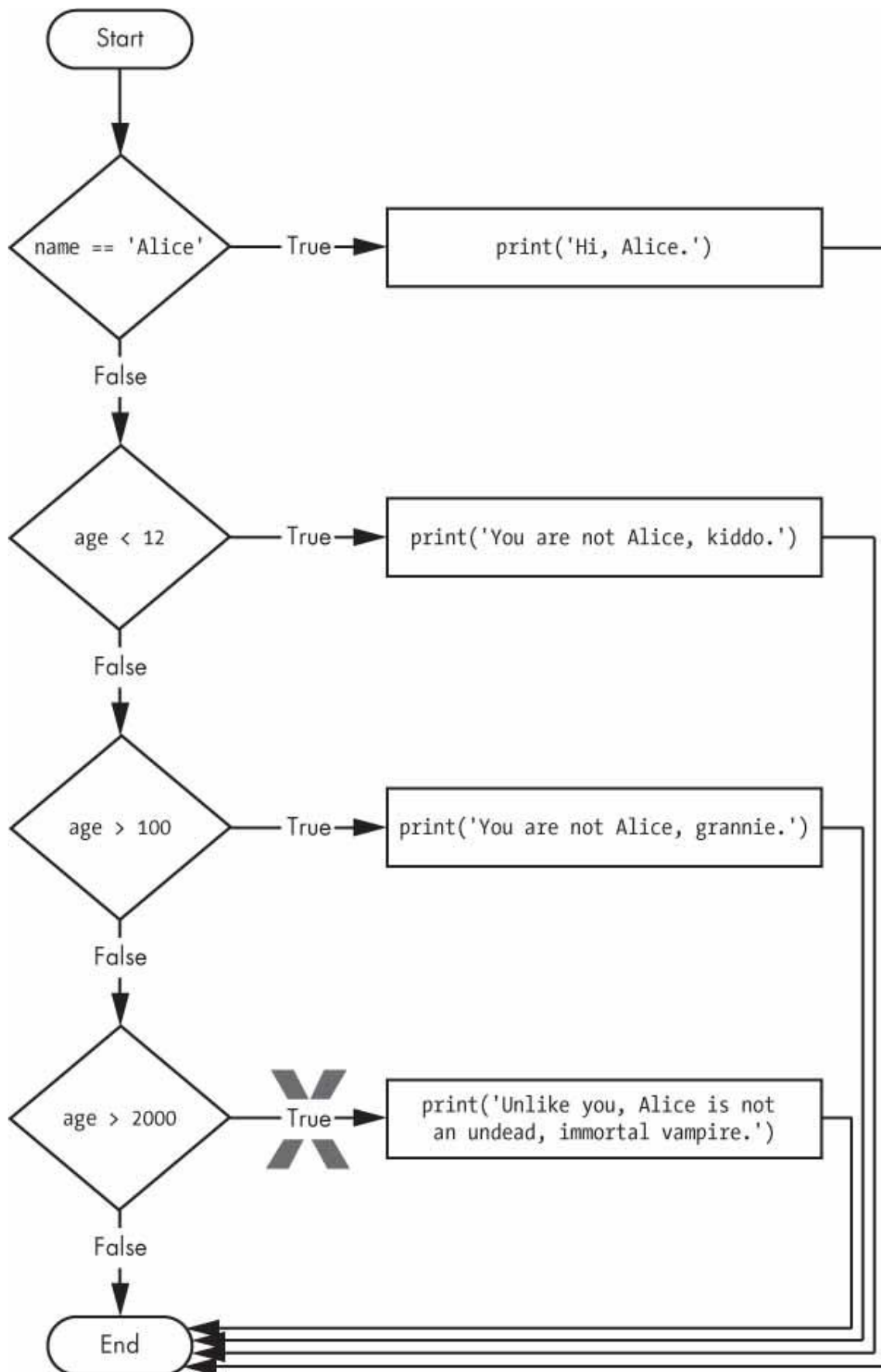


Figura 2-6: Diagrama de flujo del programa vampire2.py . Lógicamente, la ruta X nunca se ejecutará, porque si age fuese mayor que 2000 , ya habría sido mayor que 100 .

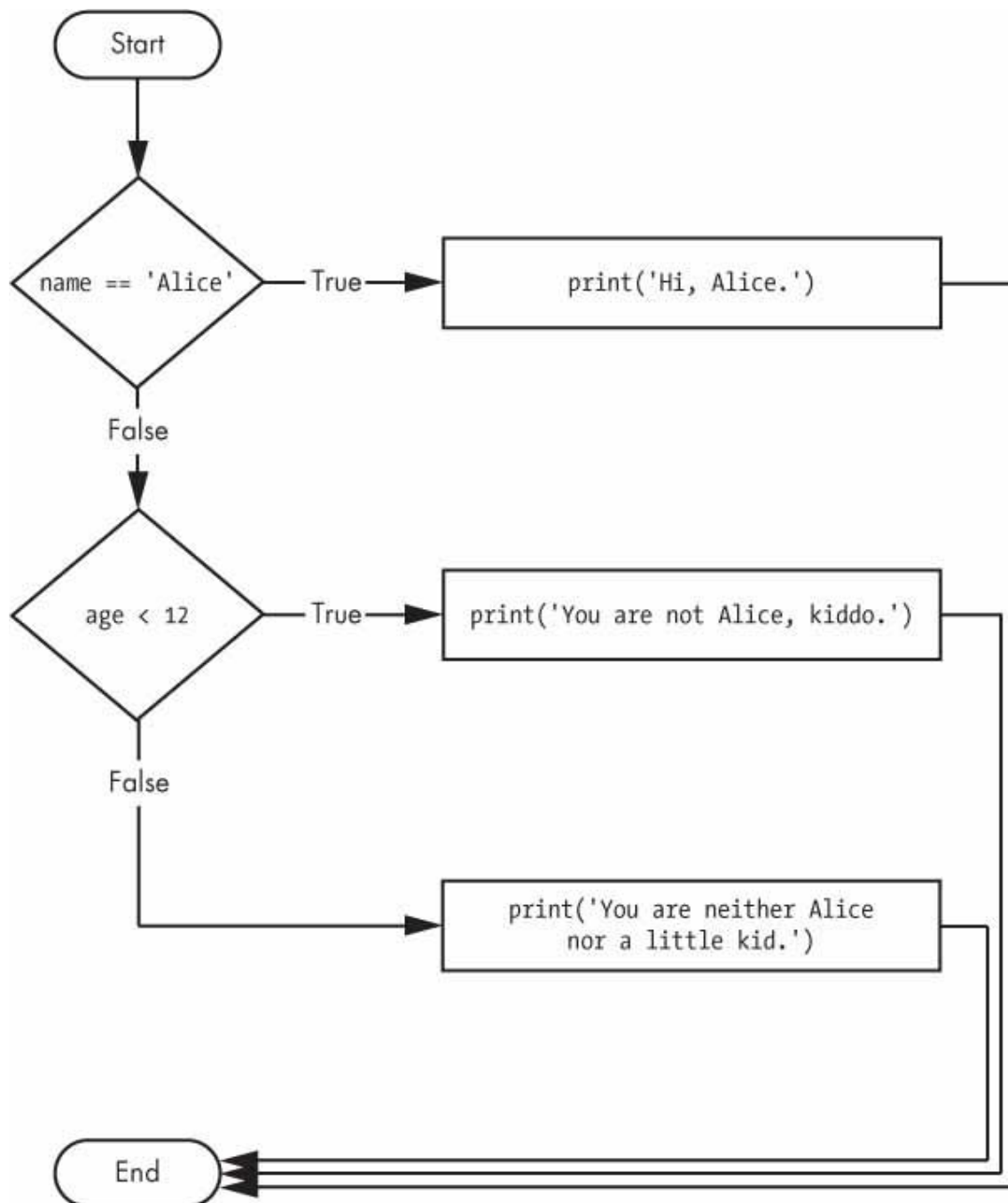


Figura 2-7: Diagrama de flujo del programa littleKid.py anterior

### **Sentencias de bucle while**

Puedes hacer que un bloque de código se ejecute una y otra vez usando una sentencia while . El código en una cláusula while se ejecutará siempre que la condición de la sentencia while sea True . En código, una sentencia while siempre consta de lo siguiente:

- La palabra clave `while`
- Una condición (es decir, una expresión que evalúa como Verdadero o Falso )
- Dos puntos
- A partir de la siguiente línea, un bloque de código sangrado (llamado cláusula `while` )

Puedes ver que una sentencia `while` se parece a una sentencia `if` . La diferencia está en cómo se comportan. Al final de una cláusula `if` , la ejecución del programa continúa después de la sentencia `if` . Pero al final de una cláusula `while` , la ejecución del programa vuelve al inicio de la sentencia `while` . La cláusula `while` se suele llamar *bucle while* o simplemente *bucle* .

Veamos una sentencia `if` y un bucle `while` que utilizan la misma condición y realizan las mismas acciones en función de esa condición. Aquí está el código con una sentencia `if` :

```
spam = 0
si spam < 5:
    print('Hola, mundo.')
    spam = spam + 1
```

Aquí está el código con una declaración `while` :

```
spam = 0
mientras spam < 5:
    print('Hola, mundo.')
    spam = spam + 1
```

Estas instrucciones son similares: tanto `if` como `while` comprueban el valor de `spam` y, si es menor que 5, imprimen un mensaje. Pero cuando se ejecutan estos dos fragmentos de código, sucede algo muy diferente para cada uno. Para la instrucción `if` , el resultado es simplemente "Hola, mundo" . Pero para la instrucción `while` , es "Hola, mundo". repetido cinco veces. Observe los diagramas de flujo de estos dos fragmentos de código, [Figuras 2-8 y 2-9](#) , para ver por qué sucede esto.

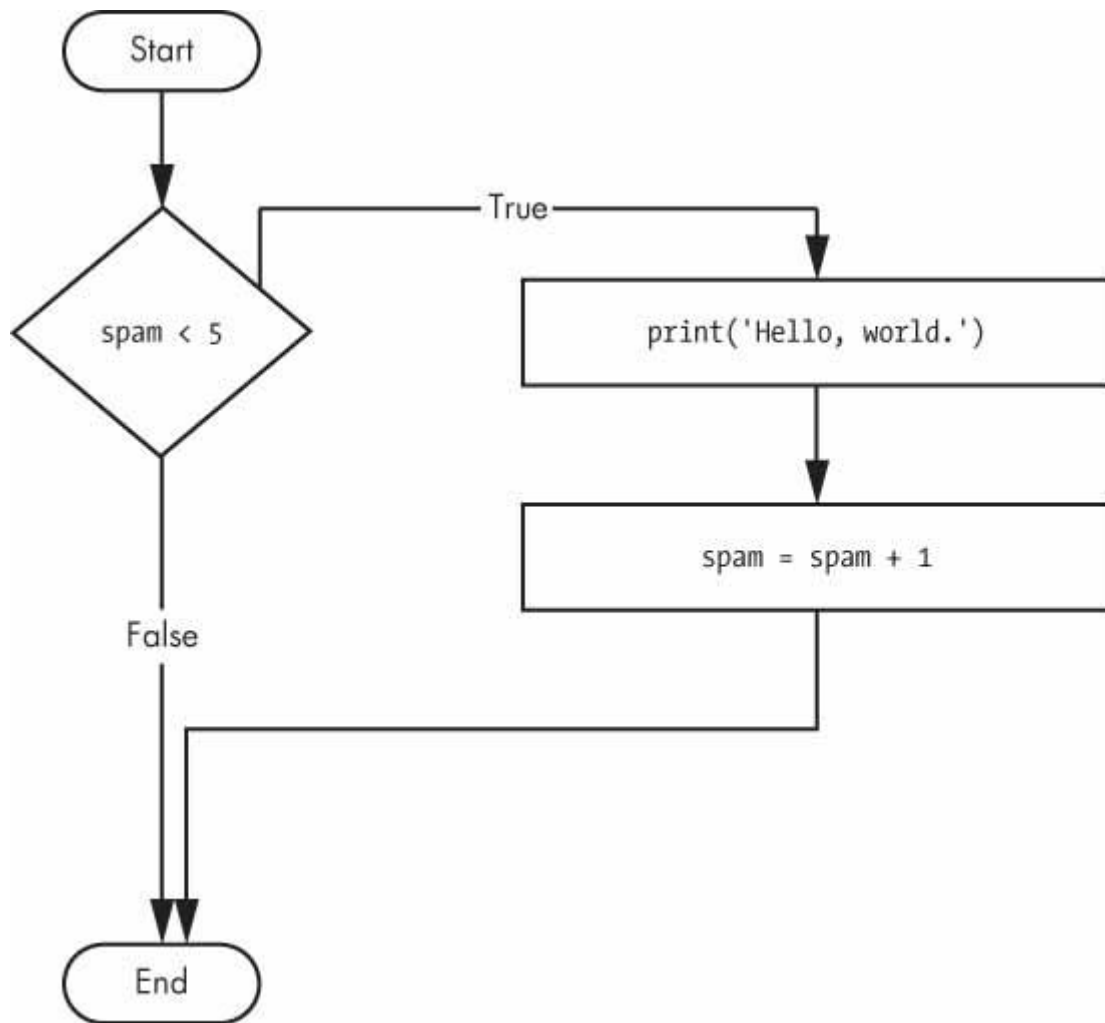


Figura 2-8: Diagrama de flujo para el código de la declaración if

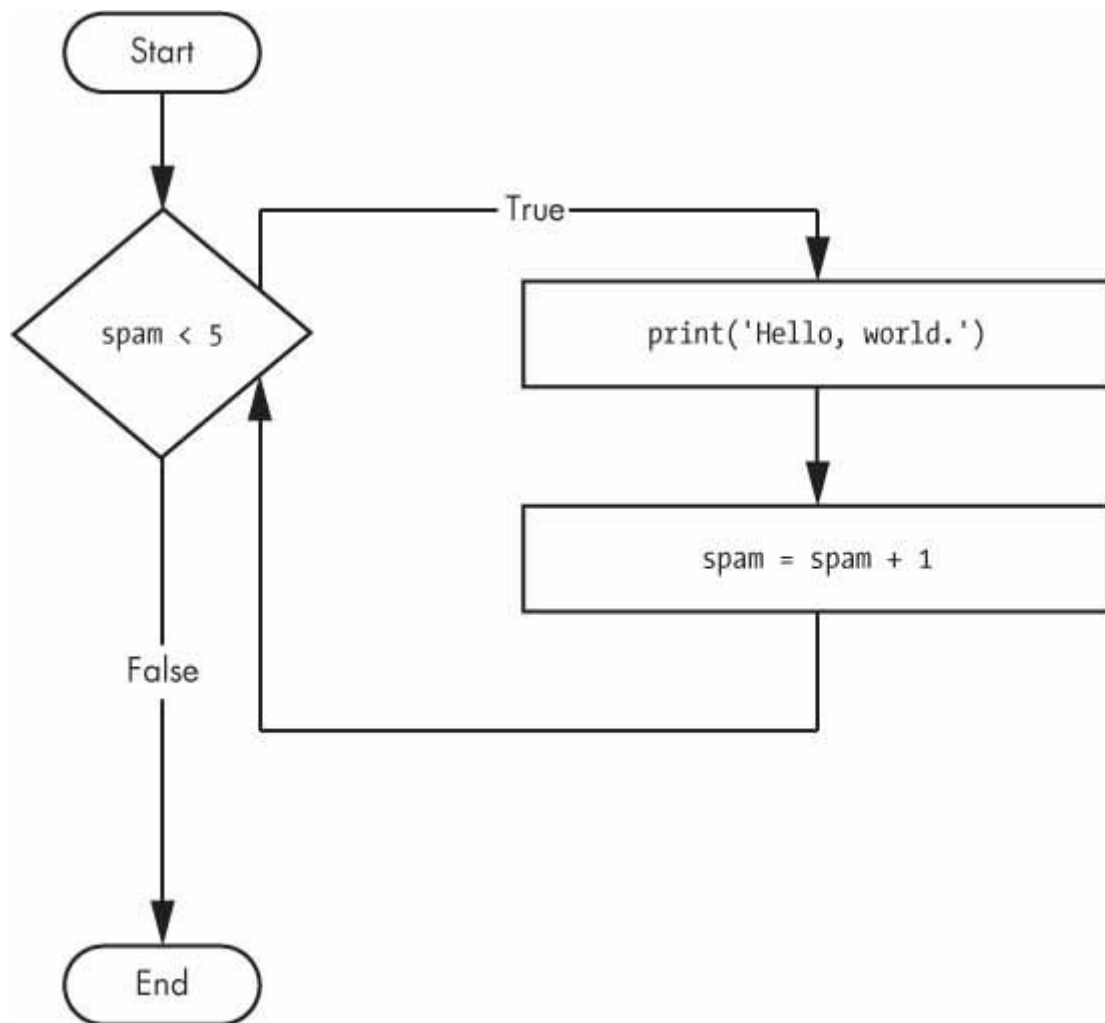


Figura 2-9: Diagrama de flujo para el código de la declaración `while`

El código con la declaración `if` verifica la condición e imprime `Hello, world.` solo una vez si esa condición es verdadera. El código con el bucle `while`, por otro lado, lo imprimirá cinco veces. El bucle se detiene después de cinco impresiones porque el entero en `spam` aumenta en uno al final de cada iteración del bucle, lo que significa que el bucle se ejecutará cinco veces antes de que `spam < 5` sea falso.

En el bucle `while`, la condición siempre se comprueba al comienzo de cada *iteración* (es decir, cada vez que se ejecuta el bucle). Si la condición es `True`, se ejecuta la cláusula `y`, a continuación, se vuelve a comprobar la condición. La primera vez que se descubre que la condición es `False`, se omite la cláusula `while`.

### Un bucle `while` molesto

A continuación, se incluye un pequeño programa de ejemplo que le pedirá que escriba, literalmente, su nombre. Seleccione **Archivo** ► **Nuevo** para abrir una

nueva ventana del editor de archivos, ingrese el siguiente código y guarde el archivo como *yourName.py* :

```
❶ nombre = ""
❷ while nombre != 'tu nombre':
    print('Por favor escribe tu nombre.')
    ❸ nombre = input()
❹ print('¡Gracias!')
```

Puede ver la ejecución de este programa en <https://autbor.com/yourname/> .

Primero, el programa establece la variable de nombre ❶ en una cadena vacía. Esto es así que la condición `nombre != 'tu nombre'` se evaluará como Verdadero y la ejecución del programa ingresará a la cláusula ❷ del bucle while .

El código dentro de esta cláusula solicita al usuario que escriba su nombre, que se asigna a la variable de nombre ❸ . Dado que esta es la última línea del bloque, la ejecución retrocede al inicio del bucle while y vuelve a evaluar la condición. Si el valor en nombre no es *igual* a la cadena 'your name' , entonces la condición es True y la ejecución ingresa nuevamente a la cláusula while .

Pero una vez que el usuario escribe yourname , la condición del bucle while será `'yourname' != 'yourname'` , que se evalúa como False . La condición ahora es False , y en lugar de que la ejecución del programa vuelva a ingresar a la cláusula del bucle while , Python la omite y continúa ejecutando el resto del programa ❹ . [La Figura 2-10](#) muestra un diagrama de flujo para el programa *yourName.py* .



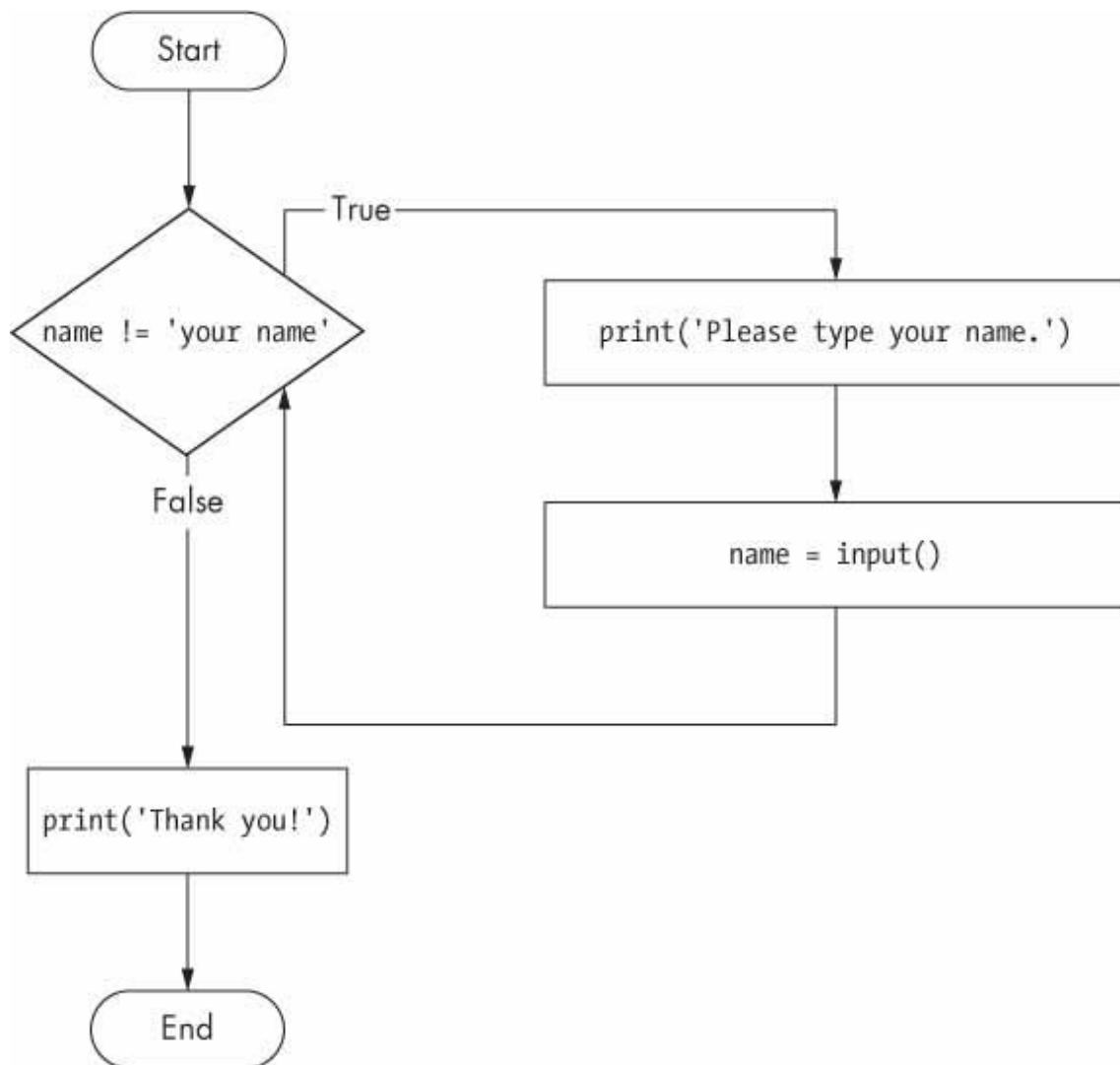


Figura 2-10: Diagrama de flujo del programa yourName.py

Ahora, veamos *yourName.py* en acción. Presione **F5** para ejecutarlo e ingrese algo que no sea su nombre varias veces antes de darle al programa lo que quiere.

Por favor escribe tu nombre.

**Al**

Por favor escribe tu nombre.

**Albert**

Por favor escribe tu nombre.

**%#@#%\*(^&!!!**

Por favor escribe tu nombre.

**tu nombre**

¡Gracias!

Si nunca ingresa su nombre, entonces la condición del bucle `while` nunca será `False` y el programa seguirá preguntando por siempre. Aquí, la llamada `input()` permite al usuario ingresar la cadena correcta para que el

programa continúe. En otros programas, la condición podría nunca cambiar y eso puede ser un problema. Veamos cómo puede salir de un bucle while .

### ***Declaraciones de ruptura***

Existe un atajo para lograr que la ejecución del programa salga antes de tiempo de la cláusula del bucle while . Si la ejecución llega a una sentencia break , sale inmediatamente de la cláusula del bucle while . En el código, una sentencia break simplemente contiene la palabra clave break .

Bastante simple, ¿verdad? Aquí hay un programa que hace lo mismo que el programa anterior, pero utiliza una sentencia break para escapar del bucle. Ingrese el siguiente código y guarde el archivo como *yourName2.py* :

```
❶ while True:
    print('Por favor escriba su nombre.')
    ❷ nombre = input()
    ❸ if nombre == 'su nombre':
        ❹ break
❺ print('¡Gracias!')
```

Puedes ver la ejecución de este programa en <https://author.com/yourname2/> . La primera línea ❶ crea un *bucle infinito* ; es un bucle while cuya condición siempre es True . (La expresión True , después de todo, siempre se evalúa hacia abajo hasta el valor True .) Después de que la ejecución del programa entre en este bucle, saldrá del bucle solo cuando se ejecute una sentencia break . (Un bucle infinito que *nunca* sale es un error de programación común).

Al igual que antes, este programa le pide al usuario que ingrese su nombre ❷ . Ahora, sin embargo, mientras la ejecución aún está dentro del bucle while , una sentencia if verifica ❸ si name es igual a 'your name' . Si esta condición es True , se ejecuta la sentencia break ❹ y la ejecución sale del bucle para imprimir ('¡Gracias!') ❺ . De lo contrario, se omite la cláusula de la sentencia if que contiene la sentencia break , lo que coloca la ejecución al final del bucle while . En este punto, la ejecución del programa salta de nuevo al inicio de la sentencia while ❶ para volver a verificar la condición. Dado que esta condición es simplemente el valor booleano True , la ejecución ingresa al bucle para solicitarle al usuario que escriba nuevamente su nombre . Vea [la Figura 2-11](#) para ver el diagrama de flujo de este programa.

Ejecute *yourName2.py* e ingrese el mismo texto que ingresó en *yourName.py* . El programa reescrito debería responder de la misma manera que el original.

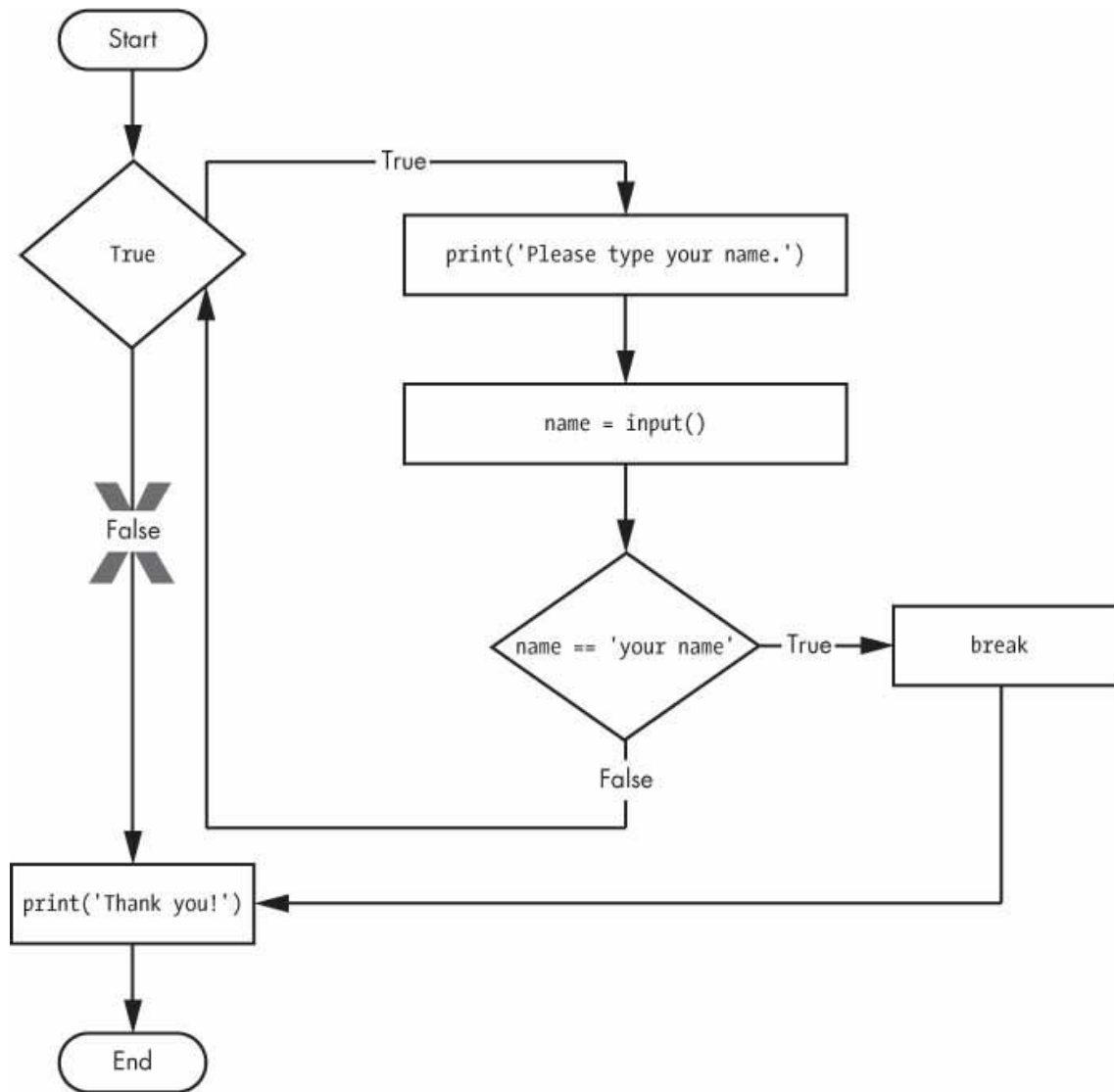


Figura 2-11: Diagrama de flujo del programa `yourName2.py` con un bucle infinito. Tenga en cuenta que la ruta X nunca se ejecutará lógicamente, porque la condición del bucle siempre es `True`.

### **continuar Declaraciones**

Al igual que las sentencias `break`, las sentencias `continue` se utilizan dentro de los bucles. Cuando la ejecución del programa llega a una sentencia `continue`, la ejecución del programa salta inmediatamente al inicio del bucle y reevalúa la condición del bucle. (Esto también es lo que sucede cuando la ejecución llega al final del bucle).

Utilicemos `continue` para escribir un programa que solicite un nombre y una contraseña. Ingrese el siguiente código en una nueva ventana del editor de archivos y guarde el programa como `swordfish.py`.

### **¿ATRAPADO EN UN BUCLE INFINITO?**

Si alguna vez ejecuta un programa que tiene un error que hace que se quede atascado en un bucle infinito, presione CTRL -C o seleccione **Shell ▶ Reiniciar Shell** en el menú de IDLE. Esto enviará un error de KeyboardInterrupt a su programa y hará que se detenga inmediatamente. Intente detener un programa creando un bucle infinito simple en el editor de archivos y guarde el programa como *infiniteLoop.py* .

mientras sea verdadero:

```
print('¡Hola, mundo!')
```

Cuando ejecute este programa, imprimirá Hello, world! en la pantalla para siempre porque la condición de la instrucción while siempre es True . CTRL -C también es útil si simplemente desea finalizar su programa inmediatamente, incluso si no está atrapado en un bucle infinito.

mientras True:

```
print('¿Quién eres?')
```

```
nombre = input()
```

❶ si nombre != 'Joe':

❷ continúe

```
print('Hola, Joe. ¿Cuál es la contraseña? (Es un pez.)')
```

❸ contraseña = input()

```
si contraseña == 'pez espada':
```

❹ romper

❺ print('Acceso concedido.')

Si el usuario ingresa cualquier nombre que no sea Joe ❶ , la instrucción continue ❷ hace que la ejecución del programa salte de nuevo al inicio del bucle. Cuando el programa reevalúa la condición, la ejecución siempre ingresará al bucle, ya que la condición es simplemente el valor True . Una vez que el usuario pasa esa instrucción if , se le solicita una contraseña ❸ . Si la contraseña ingresada es swordfish , entonces se ejecuta la instrucción break ❹ y la ejecución salta fuera del bucle while para imprimir Access grant ❺ . De lo contrario, la ejecución continúa hasta el final del bucle while , donde luego salta de nuevo al inicio del bucle. Consulte [la Figura 2-12](#) para ver el diagrama de flujo de este programa.

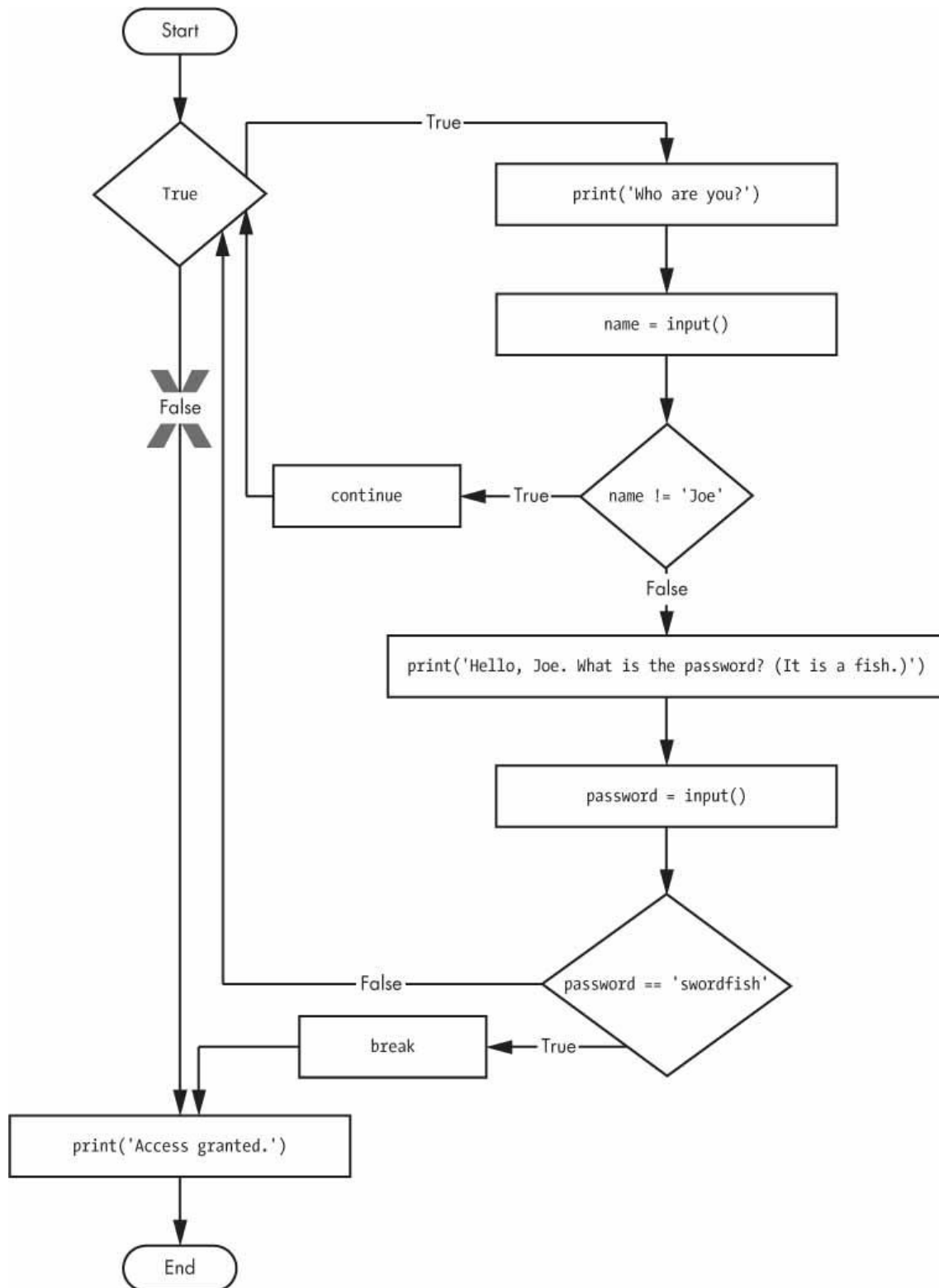


Figura 2-12: Diagrama de flujo para `swordfish.py`. Lógicamente, la ruta X nunca se ejecutará, porque la condición del bucle siempre es **True**.

### VALORES DE “VERDAD” Y “FALSEDAZ”

Las condiciones considerarán que algunos valores de otros tipos de datos son equivalentes a **True** y **False**. Cuando se utilizan en condiciones, `0`, `0.0` y `''` (la

cadena vacía) se consideran False , mientras que todos los demás valores se consideran True . Por ejemplo, observe el siguiente programa:

```
nombre = ""
❶ while not nombre:
    print('Ingresa tu nombre:')
    nombre = input()
print('¿Cuántos invitados tendrás?')
numOfGuests = int(input())
❷ if numOfGuests:
    ❸ print('Asegúrate de tener suficiente espacio para todos tus invitados.')
print('Listo')
```

Puede ver la ejecución de este programa en <https://autbor.com/howmanyguests/> . Si el usuario ingresa una cadena en blanco para name , entonces la condición de la declaración while será True ❶ , y el programa continúa solicitando un nombre. Si el valor para numOfGuests no es 0 ❷ , entonces la condición se considera True y el programa imprimirá un recordatorio para el usuario ❸ .

Podrías haber ingresado not name != "" en lugar de not name , y numOfGuests != 0 en lugar de numOfGuests , pero usar los valores truey y falsey puede hacer que tu código sea más fácil de leer.

Ejecuta este programa y dale alguna información. Hasta que digas que eres Joe, el programa no debería pedirte una contraseña y, una vez que ingreses la contraseña correcta, debería cerrarse.

¿Quién eres?

**Estoy bien, gracias. ¿Quién eres?**

¿Quién eres?

**Joe**

Hola, Joe. ¿Cuál es la contraseña? (Es un pez).

**Mary**

¿Quién eres?

**Joe**

Hola, Joe. ¿Cuál es la contraseña? (Es un pez).

**swordfish**

Acceso concedido.

Puedes ver la ejecución de este programa en <https://autbor.com/hellojoe/> .

**Bucles for y la función range()**

El bucle while continúa ejecutándose mientras su condición sea verdadera (de ahí su nombre), pero ¿qué sucede si desea ejecutar un bloque de código solo una cierta cantidad de veces? Puede hacerlo con una declaración de bucle for y la función range() .

En el código, una declaración for se parece a algo como for i in range(5): e incluye lo siguiente:

- La palabra clave for
- Un nombre de variable
- La palabra clave in
- Una llamada al método range() con hasta tres números enteros pasados
- Dos puntos
- A partir de la siguiente línea, un bloque de código sangrado (llamado cláusula for )

Creemos un nuevo programa llamado *fiveTimes.py* para ayudarle a ver un bucle for en acción.

```
print('Mi nombre es')
para i en rango(5):
    print('Jimmy Five Times (' + str(i) + '))
```

Puede ver la ejecución de este programa en <https://autbor.com/fivetimesfor/> . El código en la cláusula del bucle for se ejecuta cinco veces. La primera vez que se ejecuta, la variable i se establece en 0 . La llamada print() en la cláusula imprimirá Jimmy Five Times (0) . Después de que Python finaliza una iteración a través de todo el código dentro de la cláusula del bucle for , la ejecución vuelve al principio del bucle y la declaración for incrementa i en uno. Es por eso que range(5) da como resultado cinco iteraciones a través de la cláusula, con i establecida en 0 , luego 1 , luego 2 , luego 3 y luego 4 . La variable i irá hasta, pero no incluirá, el entero pasado a range() . [La Figura 2-13](#) muestra un diagrama de flujo para el programa *fiveTimes.py* .

Cuando ejecute este programa, debería imprimir Jimmy cinco veces seguido del valor de i cinco veces antes de salir del bucle for .

```
Mi nombre es
Jimmy Cinco Veces (0)
Jimmy Cinco Veces (1)
Jimmy Cinco Veces (2)
```

Jimmy Cinco Veces (3)

Jimmy Cinco Veces (4)

## NOTA

*También puedes usar las sentencias `break` y `continue` dentro de los bucles `for`. La sentencia `continue` continuará hasta el siguiente valor del contador del bucle `for`, como si la ejecución del programa hubiera llegado al final del bucle y hubiera regresado al inicio. De hecho, puedes usar las sentencias `continue` y `break` solo dentro de los bucles `while` y `for`. Si intentas usar estas sentencias en otro lugar, Python te dará un error.*

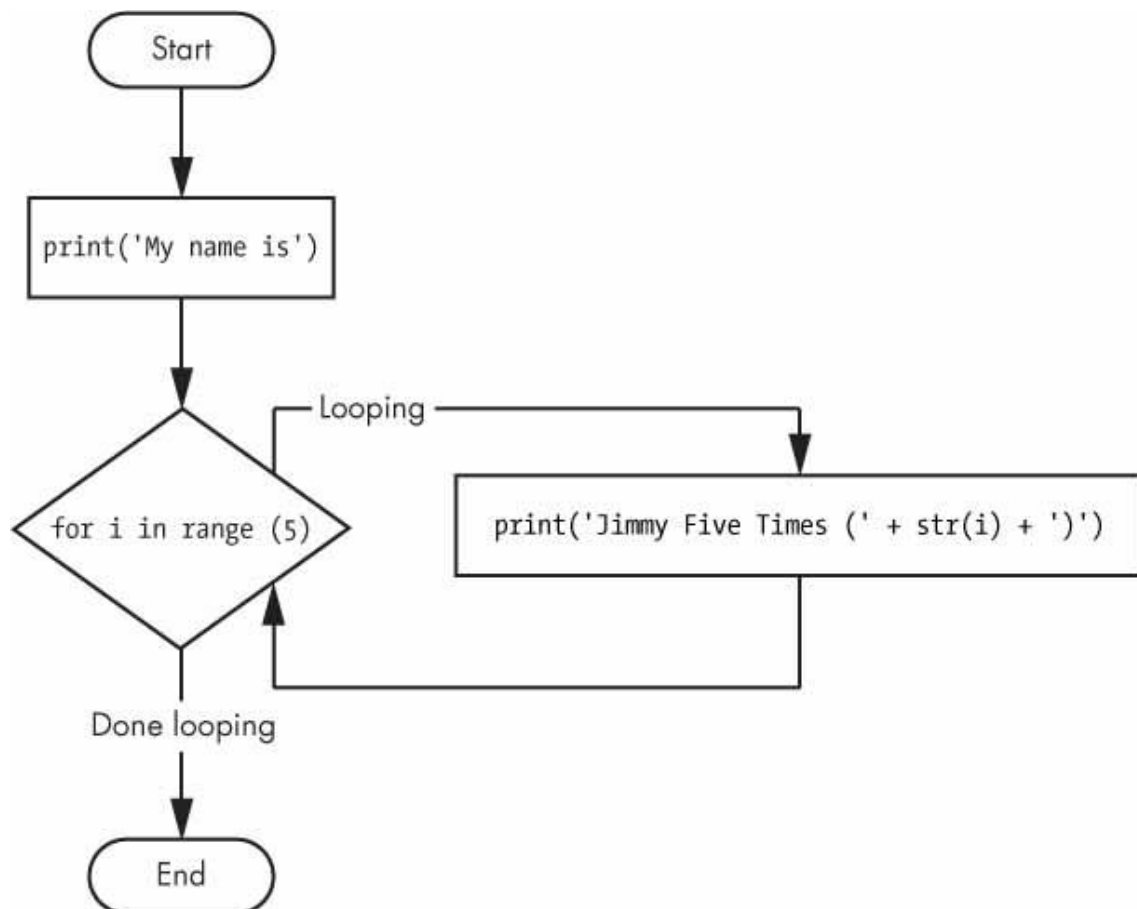


Figura 2-13: Diagrama de flujo de fiveTimes.py

Como otro ejemplo de bucle `for`, considere esta historia sobre el matemático Carl Friedrich Gauss. Cuando Gauss era un niño, un maestro quería darle a la clase algo de trabajo que no debía hacer. El maestro les dijo que sumaran todos los números del 0 al 100. El joven Gauss ideó un truco inteligente para averiguar la respuesta en unos segundos, pero usted puede escribir un programa Python con un bucle `for` para que haga este cálculo por usted.

❶ `total = 0`

❷ para `num` en `rango(101)`:



③ `total = total + num`

④ `print(total)`

El resultado debe ser 5050. Cuando el programa se inicia por primera vez, la variable `total` se establece en 0 ①. A continuación, el bucle `for` ② ejecuta `total = total + num` ③ 100 veces. Cuando el bucle haya terminado sus 100 iteraciones, todos los números enteros de 0 a 100 se habrán sumado a `total`. En este punto, `total` se imprime en la pantalla ④. Incluso en las computadoras más lentas, este programa tarda menos de un segundo en completarse.

(El joven Gauss descubrió una forma de resolver el problema en segundos. Hay 50 pares de números que suman 101: 1 + 100, 2 + 99, 3 + 98, y así sucesivamente, hasta 50 + 51. Como  $50 \times 101$  es 5050, la suma de todos los números del 0 al 100 es 5050. ¡Niño listo!)

### Un bucle `while` equivalente

En realidad, puedes usar un bucle `while` para hacer lo mismo que un bucle `for`; los bucles `for` son simplemente más concisos. Reescribamos *fiveTimes.py* para usar un bucle `while` equivalente a un bucle `for`.

```
print('Mi nombre es')
i = 0
mientras i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

Puede ver la ejecución de este programa en <https://autbor.com/fivetimeswhile/>. Si ejecuta este programa, el resultado debería ser el mismo que el del programa *fiveTimes.py*, que utiliza un bucle `for`.

### Los argumentos de inicio, detención y avance gradual de `range()`

Algunas funciones se pueden llamar con varios argumentos separados por una coma, y `range()` es una de ellas. Esto le permite cambiar el entero que se pasa a `range()` para que siga cualquier secuencia de enteros, incluso comenzando con un número distinto de cero.

```
para i en rango(12, 16):
    print(i)
```

El primer argumento será donde comienza la variable del bucle `for`, y el segundo argumento será hasta, pero sin incluir, el número en el que se detendrá.

12

13

14

15

La función `range()` también se puede llamar con tres argumentos. Los dos primeros argumentos serán los valores de inicio y fin, y el tercero será el *argumento de paso* . El paso es la cantidad en la que se incrementa la variable después de cada iteración.

para `i` en el rango `(0, 10, 2)`:

```
print(i)
```

Entonces, al llamar a `range(0, 10, 2)` se contará de cero a ocho en intervalos de dos.

0

2

4

6

8

La función `range()` es flexible en la secuencia de números que produce para los bucles `for` . *Por ejemplo* (nunca me disculpo por mis juegos de palabras), puedes incluso usar un número negativo para el argumento `step` para que el bucle `for` cuente hacia atrás en lugar de hacia arriba.

para `i` en rango `(5, -1, -1)`:

```
print(i)
```

Este bucle `for` tendría el siguiente resultado:

5

4

3

2

1

0

Al ejecutar un bucle `for` para imprimir `i` con rango `(5, -1, -1)` se debería imprimir desde cinco hasta cero.

## Importación de módulos

Todos los programas Python pueden llamar a un conjunto básico de funciones llamadas *funciones integradas* , incluidas las funciones `print()` , `input()` y `len()` que ya has visto. Python también viene con un conjunto de módulos llamados *biblioteca estándar* . Cada módulo es un programa Python que contiene un grupo relacionado de funciones que se pueden incorporar en tus programas.

Por ejemplo, el módulo `math` tiene funciones relacionadas con las matemáticas, el módulo `random` tiene funciones relacionadas con los números aleatorios, etc.

Antes de poder utilizar las funciones de un módulo, debe importar el módulo con una declaración de importación . En el código, una declaración de importación consta de lo siguiente:

- La palabra clave de importación
- El nombre del módulo
- Opcionalmente, más nombres de módulos, siempre que estén separados por comas.

Una vez que importes un módulo, podrás usar todas las funciones interesantes de ese módulo. Probémoslo con el módulo aleatorio , que nos dará acceso a la función `random.randint()` .

Ingrese este código en el editor de archivos y guárdelo como *printRandom.py* :

```
importar aleatorio
para i en rango(5):
    print(random.randint(1, 10))
```

## NO SOBRESCRIBIR LOS NOMBRES DE LOS MÓDULOS

Cuando guardes tus scripts de Python, ten cuidado de no darles un nombre que sea usado por uno de los módulos de Python, como *random.py* , *sys.py* , *os.py* o *math.py* . Si nombras accidentalmente uno de tus programas, digamos, *random.py* , y usas una declaración `import random` en otro programa, tu programa importará tu archivo *random.py* en lugar del módulo `random` de Python . Esto puede llevar a errores como `AttributeError: module 'random' has no attribute 'randint'` , ya que tu *random.py* no tiene las funciones que tiene el módulo `random` real . Tampoco uses los nombres de ninguna función incorporada de Python, como `print()` o `input()` .

Este tipo de problemas no son habituales, pero pueden ser difíciles de resolver. A medida que adquiera más experiencia en programación, conocerá mejor los nombres estándar que utilizan los módulos y las funciones de Python, y se encontrará con estos problemas con menos frecuencia.

Cuando ejecute este programa, el resultado será similar a esto:

```
4
1
8
```

Puede ver la ejecución de este programa en <https://autbor.com/printrandom/> . La llamada a la función `random.randint()` evalúa un valor entero aleatorio entre los dos enteros que le pasa. Dado que `randint()` está en el módulo `random` , primero debe escribir **random.** delante del nombre de la función para indicarle a Python que busque esta función dentro del módulo `random` .

A continuación se muestra un ejemplo de una declaración de importación que importa cuatro módulos diferentes:

```
importar aleatorio, sys, os, matemáticas
```

Ahora podemos utilizar cualquiera de las funciones de estos cuatro módulos. Aprenderemos más sobre ellas más adelante en el libro.

### ***Declaraciones de importación***

Una forma alternativa de la declaración de importación se compone de la palabra clave `from` , seguida del nombre del módulo, la palabra clave `import` y un asterisco; por ejemplo, `from random import *` .

Con esta forma de declaración de importación , las llamadas a funciones en `random` no necesitarán el prefijo `random` . Sin embargo, el uso del nombre completo hace que el código sea más legible, por lo que es mejor utilizar la forma de importación `random` de la declaración.

### **Cómo finalizar un programa antes de tiempo con la función `sys.exit()`**

El último concepto de control de flujo que se tratará es cómo finalizar el programa. Los programas siempre finalizan si la ejecución del programa llega al final de las instrucciones. Sin embargo, puede hacer que el programa finalice o salga antes de la última instrucción llamando a la función `sys.exit()` . Dado que esta función se encuentra en el módulo `sys` , debe importar `sys` antes de que su programa pueda utilizarla.

Abra una ventana del editor de archivos e ingrese el siguiente código, guardándolo como `exitExample.py` :

```
import sys

while True:
    print('Escribe exit para salir.')
    response = input()
    if response == 'exit':
```

```
sys.exit()
print('Escribiste ' + response + '')
```

Ejecute este programa en modo IDLE. Este programa tiene un bucle infinito sin ninguna instrucción break dentro. La única forma en que este programa terminará es si la ejecución llega a la llamada `sys.exit()` . Cuando `response` es igual a `exit` , se ejecuta la línea que contiene la llamada `sys.exit()` . Dado que la variable `response` está configurada por la función `input()` , el usuario debe ingresar `exit` para detener el programa.

### Un programa corto: Adivina el número

Los ejemplos que te he mostrado hasta ahora son útiles para introducir conceptos básicos, pero ahora veamos cómo todo lo que has aprendido se combina en un programa más completo. En esta sección, te mostraré un juego simple de “adivina el número”. Cuando ejecutes este programa, el resultado será algo como esto:

Estoy pensando en un número entre 1 y 20.

Adivina.

**10**

Tu suposición es demasiado baja.

Adivina.

**15**

Tu suposición es demasiado baja.

Adivina.

**17**

Tu suposición es demasiado alta.

Adivina.

**16**

¡Buen trabajo! ¡Adivinaste mi número en 4 intentos!

Ingresa el siguiente código fuente en el editor de archivos y guarde el archivo como *guessTheNumber.py* :

```
# Este es un juego de adivinar el número.
import random
secretNumber = random.randint(1, 20)
print('Estoy pensando en un número entre 1 y 20.')

# Pídele al jugador que adivine 6 veces.
for guessesTaken in range(1, 7):
    print('Adivina.')
    guess = int(input())
```

```
if guess < secretNumber:
    print('Tu suposición es demasiado baja.')
elif guess > secretNumber:
    print('Tu suposición es demasiado alta.')
else:
    break # ¡Esta condición es la suposición correcta!
```

```
if guess == secretNumber:
    print('¡Buen trabajo! ¡Adivinaste mi número en ' + str(guessesTaken) + '
guesses!')
else:
    print('No. El número en el que estaba pensando era ' + str(secretNumber))
```

Puedes ver la ejecución de este programa en <https://autbor.com/guessthenumber/> . Veamos este código línea por línea, comenzando desde arriba.

```
# Este es un juego de adivinar el número.
import random
secretNumber = random.randint(1, 20)
```

En primer lugar, un comentario en la parte superior del código explica lo que hace el programa. A continuación, el programa importa el módulo aleatorio para poder utilizar la función `random.randint()` para generar un número que el usuario deberá adivinar. El valor de retorno, un entero aleatorio entre 1 y 20, se almacena en la variable `secretNumber` .

```
print('Estoy pensando en un número entre 1 y 20.')
```

```
# Pídale al jugador que adivine 6 veces.
for guessesTaken in range(1, 7):
    print('Adivina.')
    guess = int(input())
```

El programa le dice al jugador que ha encontrado un número secreto y le dará seis oportunidades para adivinarlo. El código que permite al jugador ingresar una suposición y verifica que la haya realizado está en un bucle `for` que se repetirá seis veces como máximo. Lo primero que sucede en el bucle es que el jugador ingresa una suposición. Dado que `input()` devuelve una cadena, su valor de retorno se pasa directamente a `int()` , que traduce la cadena a un valor entero. Este se almacena en una variable llamada `guess` .

```
si suposición < numeroSecreto:
    print('Su suposición es demasiado baja.')
```

```
elif suposición > numeroSecreto:  
    print('Su suposición es demasiado alta.')
```

Estas pocas líneas de código comprueban si la suposición es menor o mayor que el número secreto. En cualquier caso, se imprime una pista en la pantalla.

```
de lo contrario:  
    break # ¡Esta condición es la suposición correcta!
```

Si la suposición no es ni mayor ni menor que el número secreto, entonces debe ser igual al número secreto; en cuyo caso, desea que la ejecución del programa salga del bucle for .

```
if guess == secretNumber:  
    print('¡Buen trabajo! ¡Adivinaste mi número en ' + str(guessesTaken) + ' guesses!')  
else:  
    print('No. El número en el que estaba pensando era ' + str(secretNumber))
```

Después del bucle for , la sentencia if...else anterior comprueba si el jugador ha adivinado correctamente el número y luego imprime un mensaje apropiado en la pantalla. En ambos casos, el programa muestra una variable que contiene un valor entero ( guessesTaken y secretNumber ). Como debe concatenar estos valores enteros en cadenas, pasa estas variables a la función str() , que devuelve la forma de valor de cadena de estos números enteros. Ahora, estas cadenas se pueden concatenar con los operadores + antes de pasarlas finalmente a la llamada a la función print() .

### **Un programa corto: Piedra, papel y tijera**

Utilicemos los conceptos de programación que hemos aprendido hasta ahora para crear un juego de piedra, papel o tijera sencillo. El resultado será el siguiente:

PIEDRA, PAPEL, TIJERA

0 victorias, 0 derrotas, 0 empates

Introduce tu movimiento: (r)ock (p)aper (s)cissors o (q)uit

**p**

PAPEL contra...

PAPEL ;

Es un empate!

0 victorias, 1 derrotas, 1 empates

Introduce tu movimiento: (r)ock (p)aper (s)cissors o (q)uit

**s**

TIJERA contra...

PAPEL

¡Tú ganas!

1 victoria, 1 derrota, 1 empate

Ingrese su movimiento: (r)ock (p)aper (s)cissors o (q)uit

**q**

Escriba el siguiente código fuente en el editor de archivos y guarde el archivo como *rpsGame.py*:

```
import random, sys
```

```
print('PIEDRA, PAPEL, TIJERA')
```

```
# Estas variables mantienen un registro del número de victorias, derrotas y  
empates.
```

```
victorias = 0
```

```
derrotas = 0
```

```
empates = 0
```

```
while True: # El bucle principal del juego.
```

```
    print('%s Victorias, %s Derrotas, %s Empates' % (victorias, derrotas, empates))
```

```
    while True: # El bucle de entrada del jugador.
```

```
        print('Ingrese su movimiento: (r)ock (p)aper (s)cissors o (q)uit')
```

```
        movimientoJugador = input()
```

```
        if movimientoJugador == 'q':
```

```
            sys.exit() # Salir del programa.
```

```
        if movimientoJugador == 'r' or movimientoJugador == 'p' or movimientoJugador  
== 's':
```

```
            break # Salir del bucle de entrada del jugador.
```

```
        print('Escribe uno de r, p, s o q.')
```

```
# Muestra lo que eligió el jugador:
```

```
if playerMove == 'r':
```

```
    print('PIEDRA versus...')
```

```
elif playerMove == 'p':
```

```
    print('PAPEL versus...')
```

```
elif playerMove == 's':
```

```
    print('TIJERAS versus...')
```

```
# Muestra lo que eligió la computadora:
```

```
randomNumber = random.randint(1, 3)
```

```
if randomNumber == 1:
```

```
    computerMove = 'r'
```

```
    print('PIEDRA')
```



```

elif randomNumber == 2:
    computerMove = 'p'
    print('PAPEL')
elif randomNumber == 3:
    computerMove = 's'
    print('TIJERAS')

# Muestra y registra la victoria/derrota/empate:
if playerMove == computerMove:
    print('¡Es un empate!')
    ties = ties + 1
elif playerMove == 'r' and computerMove == 's':
    print('¡Ganaste!')
    gana = gana + 1
elif movimientoJugador == 'p' y movimientoComputadora == 'r':
    print('¡Ganaste!')
    victorias = victorias + 1
elif movimientoJugador == 's' y movimientoComputadora == 'p':
    print('¡Ganaste!')
    victorias = victorias + 1
elif movimientoJugador == 'r' y movimientoComputadora == 'p':
    print('¡Pierdes!')
    derrotas = derrotas + 1
elif movimientoJugador == 'p' y movimientoComputadora == 's':
    print('¡Pierdes!')
    derrotas = derrotas + 1
elif movimientoJugador == 's' y movimientoComputadora == 'r':
    print('¡Pierdes!')
    derrotas = derrotas + 1

```

Veamos este código línea por línea, comenzando desde arriba.

```
import random, sys
```

```
print('PIEDRA, PAPEL, TIJERA')
```

```
# Estas variables registran el número de victorias, derrotas y empates.
```

```
victorias = 0
```

```
derrotas = 0
```

```
empates = 0
```

Primero, importamos el módulo random y sys para que nuestro programa pueda llamar a las funciones random.randint() y sys.exit() . También configuramos tres variables para llevar un registro de cuántas victorias, derrotas y empates ha tenido el jugador.

mientras True: # El bucle principal del juego.

```
print('%s Victorias, %s Derrotas, %s Empates' % (victorias, derrotas, empates))
```

mientras True: # El bucle de entrada del jugador.

```
print('Ingresa tu movimiento: (r)ock (p)aper (s)cissors o (q)uit')
```

```
movimientoJugador = input()
```

```
si movimientoJugador == 'q':
```

```
    sys.exit() # Salir del programa.
```

```
    si movimientoJugador == 'r' o movimientoJugador == 'p' o movimientoJugador == 's':
```

```
        break # Salir del bucle de entrada del jugador.
```

```
    print('Escribe uno de r, p, s o q.')
```

Este programa utiliza un bucle while dentro de otro bucle while . El primer bucle es el bucle principal del juego y se juega un único juego de piedra, papel o tijera en cada iteración de este bucle. El segundo bucle solicita la entrada del jugador y sigue repitiéndose hasta que el jugador haya introducido una r , p , s o q para su movimiento. La r , p y s corresponden a piedra, papel o tijera, respectivamente, mientras que la q significa que el jugador tiene la intención de salir. En ese caso, se llama a sys.exit() y el programa sale. Si el jugador ha introducido r , p o s , la ejecución sale del bucle. De lo contrario, el programa le recuerda al jugador que introduzca r , p , s o q y vuelve al inicio del bucle.

```
# Muestra lo que eligió el jugador:
```

```
if playerMove == 'r':
```

```
    print('PIEDRA versus...')
```

```
elif playerMove == 'p':
```

```
    print('PAPEL versus...')
```

```
elif playerMove == 's':
```

```
    print('TIJERAS versus...')
```

El movimiento del jugador se muestra en la pantalla.

```
# Muestra lo que eligió la computadora:
```

```
randomNumber = random.randint(1, 3)
```

```
if randomNumber == 1:
```

```
    computerMove = 'r'
```

```
    print('ROCA')
```

```
elif randomNumber == 2:
```

```
    computerMove = 'p'
```

```
print('PAPEL')
elif randomNumber == 3:
    computerMove = 's'
    print('TIJERAS')
```

A continuación, se selecciona aleatoriamente el movimiento de la computadora. Dado que `random.randint()` solo puede devolver un número aleatorio, el valor entero 1, 2 o 3 que devuelve se almacena en una variable llamada `randomNumber`. El programa almacena una cadena 'r', 'p' o 's' en `computerMove` en función del número entero en `randomNumber` y, además, muestra el movimiento de la computadora.

```
# Mostrar y registrar la victoria/derrota/empate:
if playerMove == computerMove:
    print('¡Es un empate!')
    ties = ties + 1
elif playerMove == 'r' and computerMove == 's':
    print('¡Ganaste!')
    wins = wins + 1
elif playerMove == 'p' and computerMove == 'r': print('¡Ganaste!')    wins = wins
+ 1 elif playerMove == 's' and computerMove == 'p': print(
    '¡Ganaste!'    )    wins = wins + 1 elif playerMove == 'r' and computerMove
== 'p':    print('¡Pierdes!')    lastimas = lastimas + 1 elif playerMove == 'p' and
computerMove == 's': print('¡Pierdes!') lastimas = lastimas + 1 elif playerMove == 'p'
and computerMove == 's' : print    (    '¡Pierdes!')    lastimas = lastimas +
1 elif playerMove == 's' and computerMove == 'r ... ¡perder!')    pérdidas =
pérdidas + 1
```

Por último, el programa compara las cadenas de `playerMove` y `computerMove` y muestra los resultados en la pantalla. También incrementa las variables `wins`, `lost` o `ties` según corresponda. Una vez que la ejecución llega al final, vuelve al inicio del bucle principal del programa para comenzar otro juego.

## FUNCIONES



Ya estás familiarizado con las funciones `print()`, `input()` y `len()` de los capítulos anteriores. Python proporciona varias funciones integradas como estas, pero también puedes escribir tus propias funciones. Una *función* es como un miniprograma dentro de un programa.

Para entender mejor cómo funcionan las funciones, vamos a crear una. Introduzca este programa en el editor de archivos y guárdelo como *helloFunc.py*:

```
❶ def hola():  
    ❷ print('¡Hola!')  
    print('¡Hola!')  
    print('Hola.')  
  
❸ hola()  
    hola()  
    hola()
```

Puedes ver la ejecución de este programa en <https://autbor.com/hellofunc/>. La primera línea es una sentencia `def` ❶, que define una función llamada `hello()`. El código del bloque que sigue a la sentencia `def` ❷ es el cuerpo de la función. Este código se ejecuta cuando se llama a la función, no cuando se define por primera vez.

Las líneas `hello()` después de la función ❸ son llamadas a funciones. En el código, una llamada a una función es simplemente el nombre de la función seguido de paréntesis, posiblemente con algún número de argumentos entre los paréntesis. Cuando la ejecución del programa llega a estas llamadas, saltará a la línea

superior de la función y comenzará a ejecutar el código allí. Cuando llega al final de la función, la ejecución regresa a la línea que llamó a la función y continúa avanzando a través del código como antes.

Dado que este programa llama a `hello()` tres veces, el código de la función `hello()` se ejecuta tres veces. Cuando se ejecuta este programa, el resultado se ve así:

```
¡Hola!  
¡Hola!  
Hola a todos.  
¡Hola!  
¡Hola!  
Hola a todos.  
¡Hola!  
¡Hola!  
Hola a todos.
```

Uno de los principales objetivos de las funciones es agrupar el código que se ejecuta varias veces. Sin una función definida, tendría que copiar y pegar este código cada vez, y el programa se vería así:

```
print('¡Hola!')  
print('¡Hola!!!') print(''  
Hola a todos.')
```

```
print('¡Hola!')  
print('¡Hola!!!') print(''  
Hola a todos.')
```

```
print('¡Hola!')  
print('¡Hola!!!')  
print('Hola a todos.')
```

En general, siempre debes evitar duplicar el código porque si alguna vez decides actualizarlo (si, por ejemplo, encuentras un error que necesitas corregir), tendrás que recordar cambiar el código en todos los lugares donde lo copiaste.

A medida que adquiera más experiencia en programación, con frecuencia se encontrará *eliminando* códigos duplicados, lo que significa deshacerse de códigos duplicados o copiados y pegados. La eliminación de códigos duplicados hace que sus programas sean más breves, más fáciles de leer y más fáciles de actualizar.

## Declaraciones `def` con parámetros

Cuando llamas a la función `print()` o `len()`, les pasas valores, llamados *argumentos*, escribiéndolos entre paréntesis. También puedes definir

tus propias funciones que acepten argumentos. Escribe este ejemplo en el editor de archivos y guárdalo como *helloFunc2.py* :

```
❶ def hola(nombre):  
    ❷ print('Hola, ' + nombre)  
  
❸ hola('Alice')  
    hola('Bob')
```

Al ejecutar este programa el resultado será el siguiente:

Hola, Alice.

Hola, Bob.

Puedes ver la ejecución de este programa en <https://autbor.com/hellofunc2/> . La definición de la función `hello()` en este programa tiene un parámetro llamado `name` ❶ . *Los parámetros* son variables que contienen argumentos. Cuando se llama a una función con argumentos, los argumentos se almacenan en los parámetros. La primera vez que se llama a la función `hello()` , se le pasa el argumento 'Alice' ❸ . La ejecución del programa ingresa a la función y el parámetro `name` se establece automáticamente en 'Alice' , que es lo que se imprime mediante la declaración `print()` ❷ .

Una cosa especial que se debe tener en cuenta acerca de los parámetros es que el valor almacenado en un parámetro se olvida cuando la función retorna. Por ejemplo, si agregé `print(name)` después de `hello('Bob')` en el programa anterior, el programa le arrojaría un `NameError` porque no hay una variable llamada `name` . Esta variable se destruye después de que la llamada a la función `hello('Bob')` retorna, por lo que `print(name)` haría referencia a una variable `name` que no existe.

Esto es similar a cómo se olvidan las variables de un programa cuando éste finaliza. Hablaré más sobre por qué sucede esto más adelante en el capítulo, cuando analice qué es el alcance local de una función.

*Definir, Llamar, Pasar, Argumento, Parámetro*

Los términos *definir* , *llamar* , *pasar* , *argumento* y *parámetro* pueden ser confusos. Veamos un ejemplo de código para repasar estos términos:

```
❶ def sayHello(nombre):  
    print('Hola, ' + nombre)  
❷ sayHello('Al')
```

Definir una función es crearla, al igual que una sentencia de asignación como `spam = 42` crea la variable `spam` . La sentencia `def` *define* la función `sayHello()` ❶ . La línea `sayHello('Al')` ❷ *llama* a la función ahora creada, enviando la ejecución a la parte superior del código de la función. Esta llamada de función también se conoce como *pasar* el valor de cadena 'Al' a la función. Un valor que se En una llamada a una función se pasa un *argumento* . El argumento 'Al' se asigna a una variable local denominada `nombre` . Las variables que tienen argumentos asignados son *parámetros* .

Es fácil confundir estos términos, pero mantenerlos claros garantizará que sepa exactamente lo que significa el texto de este capítulo.

### Valores de retorno y declaraciones de retorno

Cuando llamas a la función `len()` y le pasas un argumento como 'Hola' , la llamada a la función se evalúa como el valor entero 5 , que es la longitud de la cadena que le pasaste. En general, el valor que se evalúa como resultado de una llamada a la función se denomina *valor de retorno* de la función.

Al crear una función mediante la declaración `def` , puede especificar cuál debe ser el valor de retorno con una declaración de retorno . Una declaración de retorno consta de lo siguiente:

- La palabra clave `return`
- El valor o expresión que debe devolver la función

Cuando se utiliza una expresión con una declaración de retorno , el valor de retorno es el que evalúa esta expresión. Por ejemplo, el siguiente programa define una función que devuelve una cadena diferente según el número que se le pase como argumento. Ingrese este código en el editor de archivos y guárdelo como *magic8Ball.py* :

```
❶ import random

❷ def getAnswer(answerNumber):
    ❸ if answerNumber == 1:
        return 'Es seguro'
    elif answerNumber == 2:
        return 'Es decididamente así'
    elif answerNumber == 3:
        return 'Sí'
    elif answerNumber == 4:
        return 'Respuesta confusa, inténtalo de nuevo'
    elif answerNumber == 5:
```

```

        return 'Pregunta de nuevo más tarde'
    elif answerNumber == 6:
        return 'Concéntrate y pregunta de nuevo'
    elif answerNumber == 7:
        return 'Mi respuesta es no'
    elif answerNumber == 8:
        return 'La perspectiva no es tan buena'
    elif answerNumber == 9:
        return 'Muy dudoso'

```

```

❷ r = random.randint(1, 9)
❸ fortune = getAnswer(r)
❹ print(fortune)

```

Puedes ver la ejecución de este programa en <https://autbor.com/magic8ball/> . Cuando este programa se inicia, Python primero importa el módulo aleatorio ❶ . Luego se define la función getAnswer() ❷ . Debido a que la función se define (y no se llama), la ejecución omite el código que contiene. A continuación, se llama a la función random.randint() con dos argumentos: 1 y 9 ❸ . Se evalúa como un entero aleatorio entre 1 y 9 (incluidos los propios 1 y 9 ), y este valor se almacena en una variable llamada r .

La función getAnswer() se llama con r como argumento ❹ . La ejecución del programa se mueve al principio de la función getAnswer() ❺ , y el valor r se almacena en un parámetro llamado answerNumber . Luego, dependiendo del valor en answerNumber , la función devuelve uno de los muchos valores de cadena posibles. La ejecución del programa regresa a la línea en la parte inferior del programa que originalmente llamó a getAnswer() ❻ . La cadena devuelta se asigna a una variable llamada fortune , que luego se pasa a una llamada print() ❼ y se imprime en la pantalla.

Tenga en cuenta que, dado que puede pasar valores de retorno como argumento a otra llamada de función, puede acortar estas tres líneas:

```

r = aleatorio.randint(1, 9)
fortuna = obtenerRespuesta(r)
imprimir(fortuna)

```

a esta única línea equivalente:

```

imprimir(obtenerRespuesta(aleatorio.randint(1, 9)))

```



Recuerde que las expresiones se componen de valores y operadores. Se puede utilizar una llamada de función en una expresión porque la llamada evalúa su valor de retorno.

### El valor ninguno

En Python, existe un valor llamado `None`, que representa la ausencia de un valor. El valor `None` es el único valor del tipo de datos `NoneType`. (Otros lenguajes de programación pueden llamar a este valor `null`, `nil` o `undefined`). Al igual que los valores booleanos `True` y `False`, `None` debe escribirse con *N* mayúscula.

Este valor sin valor puede ser útil cuando necesitas almacenar algo que no se confunda con un valor real en una variable. Un lugar donde se usa `None` es como valor de retorno de `print()`. La función `print()` muestra texto en la pantalla, pero no necesita devolver nada de la misma manera que lo hacen `len()` o `input()`. Pero como todas las llamadas de función necesitan evaluar un valor de retorno, `print()` devuelve `None`. Para ver esto en acción, ingresa lo siguiente en el shell interactivo:

```
>>> spam = print('¡Hola!')
```

```
¡Hola!
```

```
>>> Ninguno == spam
```

```
Verdadero
```

Tras bambalinas, Python agrega `return None` al final de cualquier definición de función sin una declaración de retorno. Esto es similar a cómo un bucle `while` o `for` termina implícitamente con una declaración de continuación. Además, si usa una declaración de retorno sin un valor (es decir, solo la palabra clave `return` por sí sola), se devuelve `None`.

### Argumentos de palabras clave y la función `print()`

La mayoría de los argumentos se identifican por su posición en la llamada de función. Por ejemplo, `random.randint(1, 10)` es diferente de `random.randint(10, 1)`. La llamada de función `random.randint(1, 10)` devolverá un entero aleatorio entre 1 y 10 porque el primer argumento es el extremo inferior del rango y el segundo argumento es el extremo superior (mientras que `random.randint(10, 1)` provoca un error).

Sin embargo, en lugar de por su posición, *los argumentos de palabras clave* se identifican por la palabra clave que se coloca antes de ellos en la llamada de función. Los argumentos de palabras clave se utilizan a menudo para *parámetros opcionales*. Por ejemplo, la función `print()` tiene los parámetros opcionales `end` y `sep` para especificar lo que se debe imprimir al final de sus argumentos y entre sus argumentos (separándolos), respectivamente.

Si ejecutaste un programa con el siguiente código:

```
print('Hola')  
print('Mundo')
```

La salida se vería así:

```
Hola  
Mundo
```

Las dos cadenas de salida aparecen en líneas separadas porque la función `print()` agrega automáticamente un carácter de nueva línea al final de la cadena que se le pasa. Sin embargo, puede establecer el argumento de palabra clave `end` para cambiar el carácter de nueva línea a una cadena diferente. Por ejemplo, si el código fuera este:

```
print('Hola', fin='')  
print('Mundo')
```

La salida se vería así:

```
Hola Mundo
```

La salida se imprime en una sola línea porque ya no se imprime una nueva línea después de 'Hello' . En su lugar, se imprime la cadena en blanco. Esto es útil si necesita deshabilitar la nueva línea que se agrega al final de cada llamada a la función `print()` .

De manera similar, cuando pasas múltiples valores de cadena a `print()` , la función los separará automáticamente con un solo espacio. Introduce lo siguiente en el shell interactivo:

```
>>> print('gatos', 'perros', 'ratones')  
gatos perros ratones
```

Pero puede reemplazar la cadena de separación predeterminada al pasarle al argumento de la palabra clave `sep` una cadena diferente. Ingresa lo siguiente en el shell interactivo:

```
>>> print('gatos', 'perros', 'ratones', sep=',')  
gatos,perros,ratones
```

También puedes agregar argumentos de palabras clave a las funciones que escribas, pero primero tendrás que aprender sobre los tipos de datos de lista y diccionario en los próximos dos capítulos. Por ahora, solo debes saber que algunas funciones tienen argumentos de palabras clave opcionales que se pueden especificar cuando se llama a la función.

## La pila de llamadas

Imagina que tienes una conversación con alguien que no tiene nada que ver. Hablas de tu amiga Alice, lo que te recuerda una historia sobre tu compañero de trabajo Bob, pero primero tienes que explicar algo sobre tu prima Carol. Terminas tu historia sobre Carol y vuelves a hablar de Bob, y cuando terminas tu historia sobre Bob, vuelves a hablar de Alice. Pero entonces te acuerdas de tu hermano David, así que cuentas una historia sobre él y luego vuelves a terminar tu historia original sobre Alice. Tu conversación siguió una estructura similar *a una pila*, como en [la Figura 3-1](#). La conversación es similar a una pila porque el tema actual siempre está en la parte superior de la pila.

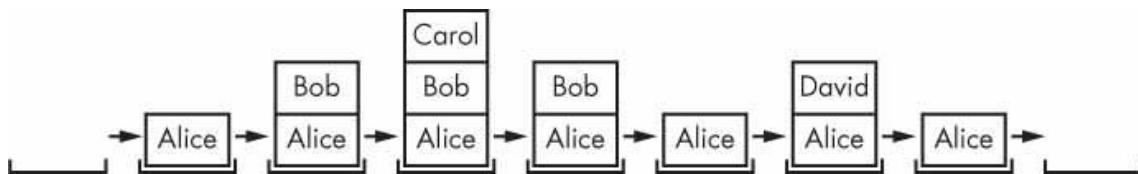


Figura 3-1: Su pila de conversaciones serpenteantes

De manera similar a nuestra conversación sinuosa, llamar a una función no envía la ejecución en un viaje de ida hacia la parte superior de la función. Python recordará qué línea de código llamó a la función para que la ejecución pueda regresar allí cuando encuentre una declaración de retorno. Si esa función original llamó a otras funciones, la ejecución regresaría primero a esas llamadas de función, antes de regresar de la llamada de función original.

Abra una ventana del editor de archivos e ingrese el siguiente código, guardándolo como `abcdCallStack.py`:

```
def a():
    print('a() comienza')
    ❶ b()
    ❷ d()
    print('a() regresa')

def b():
    print('b() comienza')
    ❸ c()
    print('b() regresa')

def c():
    ❹ print('c() comienza')
    print('c() regresa')
```

```
def d():  
    print('d() comienza')  
    print('d() regresa')
```

⑤ a()

Si ejecuta este programa, el resultado se verá así:

```
a() inicia  
b() inicia  
c() inicia  
c() retorna  
b() retorna  
d() inicia  
d() retorna  
a() retorna
```

Puedes ver la ejecución de este programa en <https://autbor.com/abcdcallstack/>.

Cuando se llama a a() ⑤, llama a b() ①, que a su vez llama a c() ③. La función c() no llama a nada; solo muestra c() comienza ④ y c() regresa antes de regresar a la línea en b() que lo llamó ③. Una vez que la ejecución regresa al código en b() que llamó a c(), regresa a la línea en a() que llamó a b() ①. La ejecución continúa a la siguiente línea en la función b() ②, que es una llamada a d(). Al igual que la función c(), la función d() tampoco llama a nada. Solo muestra d() comienza y d() regresa antes de regresar a la línea en b() que lo llamó. Dado que b() no contiene otro código, la ejecución regresa a la línea en a() que llamó a b() ②. La última línea en a() muestra los retornos de a() antes de regresar a la llamada a() original al final del programa ⑤.

La *pila de llamadas* es la forma en que Python recuerda dónde devolver la ejecución después de cada llamada de función. La pila de llamadas no se almacena en una variable en su programa; en cambio, Python la maneja detrás de escena. Cuando su programa llama a una función, Python crea un *objeto de marco* en la parte superior de la pila de llamadas. MarcoLos objetos almacenan el número de línea de la llamada a la función original para que Python pueda recordar dónde regresar. Si se realiza otra llamada a la función, Python coloca otro objeto de marco en la pila de llamadas por encima del otro.

Cuando una llamada de función retorna, Python elimina un objeto de marco de la parte superior de la pila y mueve la ejecución al número de línea almacenado en él. Tenga en cuenta que los objetos de marco siempre se agregan y eliminan de la parte superior de la pila y no de ningún otro lugar. [La Figura 3-2](#) ilustra el estado de

la pila de llamadas en *abcdCallStack.py* a medida que se llama a cada función y retorna.

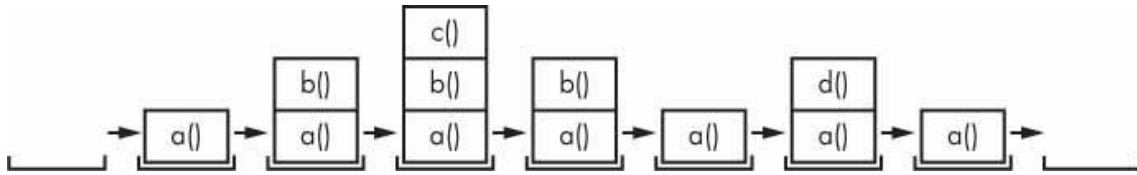


Figura 3-2: Los objetos de marco de la pila de llamadas como llamadas *abcdCallStack.py* y retornos de funciones

La parte superior de la pila de llamadas es la función en la que se encuentra actualmente la ejecución. Cuando la pila de llamadas está vacía, la ejecución está en una línea fuera de todas las funciones.

La pila de llamadas es un detalle técnico que no es estrictamente necesario conocer para escribir programas. Es suficiente entender que las llamadas a funciones regresan al número de línea desde el que fueron llamadas. Sin embargo, comprender las pilas de llamadas facilita la comprensión de los ámbitos local y global, que se describen en la siguiente sección.

#### Alcance local y global

Se dice que los parámetros y las variables que se asignan en una función llamada existen en *el ámbito local* de esa función. Se dice que las variables que se asignan fuera de todas las funciones existen en *el ámbito global*. Una variable que existe en un ámbito local se denomina *variable local*, mientras que una variable que existe en el ámbito global se denomina *variable global*. Una variable debe ser una u otra; no puede ser local y global a la vez.

Piense en un *ámbito* como un contenedor de variables. Cuando se destruye un ámbito, se olvidan todos los valores almacenados en las variables del ámbito. Solo hay un ámbito global y se crea cuando comienza su programa. Cuando su programa termina, el ámbito global se destruye y se olvidan todas sus variables. De lo contrario, la próxima vez que ejecute un programa, las variables recordarán sus valores de la última vez que lo ejecutó.

Se crea un ámbito local cada vez que se llama a una función. Todas las variables asignadas en la función existen dentro del ámbito local de la función. Cuando la función retorna, el ámbito local se destruye y estas variables se olvidan. La próxima vez que llame a la función, las variables locales no recordarán los valores almacenados en ellas desde la última vez que se llamó a la función. Las variables locales también se almacenan en objetos de marco en la pila de llamadas.

Los alcances son importantes por varias razones:

- El código en el ámbito global, fuera de todas las funciones, no puede utilizar ninguna variable local.
- Sin embargo, el código en un ámbito local puede acceder a variables globales.
- El código en el ámbito local de una función no puede utilizar variables en ningún otro ámbito local.
- Puedes usar el mismo nombre para distintas variables si se encuentran en ámbitos diferentes. Es decir, puede haber una variable local denominada spam y una variable global también denominada spam .

La razón por la que Python tiene diferentes ámbitos en lugar de simplemente convertir todo en una variable global es que cuando las variables son modificadas por el código en una llamada particular a una función, la función interactúa con el resto del programa solo a través de sus parámetros y el valor de retorno. Esto reduce la cantidad de líneas de código que pueden estar causando un error. Si su programa no contenía nada más que variables globales y tenía un error debido a que una variable se estableció en un valor incorrecto, entonces sería difícil rastrear dónde se estableció este valor incorrecto. ¡Podría haberse establecido desde cualquier parte del programa y su programa podría tener cientos o miles de líneas de largo! Pero si el error es causado por una variable local con un valor incorrecto, sabe que solo el código en esa función podría haberlo establecido incorrectamente.

Si bien está bien usar variables globales en programas pequeños, es un mal hábito depender de variables globales a medida que los programas se hacen cada vez más grandes.

*Las variables locales no se pueden utilizar en el ámbito global*

Considere este programa, que provocará un error cuando lo ejecute:

```
def spam():
    ❶ huevos = 31337
    spam()
    print(huevos)
```

Si ejecuta este programa, el resultado se verá así:

```
Traceback (última llamada más reciente):
  Archivo "C:/test1.py", línea 4, en <module>
    print(eggs)
NameError: el nombre 'eggs' no está definido
```

El error ocurre porque la variable `eggs` existe solo en el ámbito local creado cuando se llama a `spam()` ❶. Una vez que la ejecución del programa regresa de `spam`, ese ámbito local se destruye y ya no hay una variable llamada `eggs`. Entonces, cuando su programa intenta ejecutar `print(eggs)`, Python le da un error que dice que `eggs` no está definido. Esto tiene sentido si lo piensa; cuando la ejecución del programa está en el ámbito global, no existen ámbitos locales, por lo que no puede haber ninguna variable local. Es por eso que solo se pueden usar variables globales en el ámbito global.

*Los ámbitos locales no pueden utilizar variables en otros ámbitos locales*

Se crea un nuevo ámbito local cada vez que se llama a una función, incluso cuando se llama a una función desde otra función. Considere este programa:

```
def spam():
    ❶ huevos = 99
    ❷ tocino()
    ❸ print(huevos)

def tocino():
    jamón = 101
    ❹ huevos = 0

❺ spam()
```

Puede ver la ejecución de este programa en <https://autbor.com/otherlocalscopes/>. Cuando se inicia el programa, se llama a la función `spam()` ❺ y se crea un ámbito local. La variable local `eggs` ❶ se establece en 99. Luego se llama a la función `bacon()` ❷ y se crea un segundo ámbito local. Pueden existir varios ámbitos locales al mismo tiempo. En este nuevo ámbito local, la variable local `ham` se establece en 101 y también se crea una variable local `eggs` —que es diferente de la del ámbito local de `spam()`— ❹ y se establece en 0.

Cuando `bacon()` retorna, el ámbito local para esa llamada se destruye, incluida su variable `eggs`. La ejecución del programa continúa en la función `spam()` para imprimir el valor de `eggs` ❸. Dado que el ámbito local para la llamada a `spam()` aún existe, la única variable `eggs` es la variable `eggs` de la función `spam()`, que se estableció en 99. Esto es lo que imprime el programa.

El resultado es que las variables locales de una función están completamente separadas de las variables locales de otra función.

*Las variables globales se pueden leer desde un ámbito local*

Consideremos el siguiente programa:

```
def spam():  
    print(huevos)  
huevos = 42  
spam()  
print(huevos)
```

Puedes ver la ejecución de este programa en <https://autbor.com/readglobal/>.

Dado que no hay ningún parámetro llamado eggs ni ningún código que asigne un valor a eggs en la función spam() , cuando eggs se usa en spam() , Python lo considera una referencia a la variable global eggs . Es por eso que se imprime 42 cuando se ejecuta el programa anterior.

*Variables locales y globales con el mismo nombre*

Técnicamente, es perfectamente aceptable utilizar el mismo nombre de variable para una variable global y para variables locales en distintos ámbitos en Python. Pero, para simplificarte la vida, evita hacerlo. Para ver qué sucede, ingresa el siguiente código en el editor de archivos y guárdalo como *localGlobalSameName.py* :

```
def spam():  
    ❶ huevos = 'spam local'  
    print(huevos) # imprime 'spam local'  
  
def tocino():  
    ❷ huevos = 'tocino local'  
    print(huevos) # imprime 'tocino local'  
    spam()  
    print(huevos) # imprime 'tocino local'  
  
❸ huevos = 'global'  
    tocino()  
    print(huevos) # imprime 'global'
```

Al ejecutar este programa se genera el siguiente resultado:

```
tocino local  
spam local  
tocino local  
global
```



Puede ver la ejecución de este programa en <https://autbor.com/localglobalsamenname/> . En realidad, hay tres variables diferentes en este programa, pero, para mayor confusión, todas se llaman eggs . Las variables son las siguientes:

- ❶ Una variable llamada huevos que existe en un ámbito local cuando se llama a spam() .
- ❷ Una variable llamada huevos que existe en un ámbito local cuando se llama a bacon() .
- ❸ Una variable llamada huevos que existe en el ámbito global.

Dado que estas tres variables independientes tienen el mismo nombre, puede resultar confuso hacer un seguimiento de cuál se está utilizando en un momento determinado. Por este motivo, debe evitar utilizar el mismo nombre de variable en distintos ámbitos.

#### La Declaración global

Si necesita modificar una variable global desde una función, utilice la declaración global . Si tiene una línea como global eggs en la parte superior de una función, le indica a Python: “En esta función, eggs hace referencia a la variable global, por lo que no cree una variable local con este nombre”. Por ejemplo, ingrese el siguiente código en el editor de archivos y guárdelo como *globalStatement.py* :

```
def spam():  
    ❶ huevos globales  
    ❷ huevos = 'spam'  
  
huevos = spam 'global'  
(  
print(huevos)
```

Cuando ejecute este programa, la llamada print() final generará esto:

correo basura

Puede ver la ejecución de este programa en <https://autbor.com/globalstatement/> . Debido a que eggs se declara global en la parte superior de spam() ❶ , cuando eggs se establece en 'spam' ❷ , esta asignación se realiza a eggs con alcance global . No se crea ninguna variable eggs local .

Hay cuatro reglas para determinar si una variable está en un ámbito local o global:

- Si una variable se utiliza en el ámbito global (es decir, fuera de todas las funciones), entonces siempre es una variable global.
- Si hay una declaración global para esa variable en una función, es una variable global.
- De lo contrario, si la variable se utiliza en una declaración de asignación en la función, es una variable local.
- Pero si la variable no se utiliza en una declaración de asignación, es una variable global.

Para entender mejor estas reglas, aquí se incluye un programa de ejemplo. Introduzca el siguiente código en el editor de archivos y guárdelo como *sameNameLocalGlobal.py*:

```
def spam():
    ❶ global huevos
    huevos = 'spam' # este es el global

def bacon():
    ❷ huevos = 'bacon' # este es un local

def ham():
    ❸ print(huevos) # este es el global

huevos = 42 # este es el
spam() global
print(huevos)
```

En la función `spam()`, `eggs` es la variable global `eggs` porque hay una declaración global para `eggs` al comienzo de la función ❶. En `bacon()`, `eggs` es una variable local porque hay una declaración de asignación para ella en esa función ❷. En `ham()` ❸, `eggs` es la variable global porque no hay una declaración de asignación o declaración global para ella en esa función. Si ejecuta *sameNameLocalGlobal.py*, la salida se verá así:

correo basura

Puede ver la ejecución de este programa en <https://autbor.com/sameNameLocalGlobal/>. En una función, una variable siempre será global o siempre será local. El código de una función no puede usar una variable local llamada `eggs` y luego usar la variable global `eggs` más adelante en esa misma función.

## NOTA

*Si alguna vez desea modificar el valor almacenado en una variable global desde una función, debe utilizar una declaración global en esa variable.*

Si intentas usar una variable local en una función antes de asignarle un valor, como en el siguiente programa, Python te dará un error. Para verlo, ingresa lo siguiente en el editor de archivos y guárdalo como `sameNameError.py`:

```
def spam():  
    print(huevos) # ¡ERROR!  
    ❶ huevos = 'spam local'
```

```
    ❷ huevos = 'global'  
    spam()
```

Si ejecuta el programa anterior, aparecerá un mensaje de error.

Traceback (última llamada más reciente):

```
Archivo "C:/sameNameError.py", línea 6, en <module>  
    spam()
```

```
Archivo "C:/sameNameError.py", línea 2, en spam  
    print(eggs) # ¡ERROR!
```

UnboundLocalError: se hizo referencia a la variable local 'eggs' antes de la asignación

Puedes ver la ejecución de este programa

en <https://autbor.com/sameNameError/>. Este error ocurre porque Python ve que hay una declaración de asignación para eggs en la función spam() ❶ y, por lo tanto, considera que eggs es local. Pero como print(eggs) se ejecuta antes de que se le asigne algo a eggs, la variable local eggs no existe. Python *no* recurrirá a la variable global eggs ❷.

## FUNCIONA COMO “CAJAS NEGRAS”

A menudo, todo lo que necesitas saber sobre una función son sus entradas (los parámetros) y el valor de salida; no siempre tienes que preocuparte por cómo funciona realmente el código de la función. Cuando piensas en las funciones de esta manera de alto nivel, es común decir que estás tratando a una función como una "caja negra".

Esta idea es fundamental para la programación moderna. En los capítulos posteriores de este libro se mostrarán varios módulos con funciones escritas por otras personas. Si bien puede echar un vistazo al código fuente si siente curiosidad, no necesita saber cómo funcionan estas funciones para poder

usarlas. Y como se recomienda escribir funciones sin variables globales, por lo general no tiene que preocuparse por la interacción del código de la función con el resto del programa.

### Manejo de excepciones

En este momento, recibir un error o *excepción* en su programa Python significa que todo el programa se bloqueará. No desea que esto suceda en programas del mundo real. En cambio, desea que el programa detecte errores, los gestione y luego continúe ejecutándose.

Por ejemplo, considere el siguiente programa, que tiene un error de división por cero. Abra una ventana del editor de archivos e ingrese el siguiente código, guardándolo como *zeroDivide.py* :

```
def spam(divideBy):  
    return 42 / divideBy  
  
imprimir(spam(2))  
imprimir(spam(12))  
imprimir(spam(0))  
imprimir(spam(1))
```

Hemos definido una función llamada *spam* , le hemos asignado un parámetro y luego hemos impreso el valor de esa función con varios parámetros para ver qué sucede. Este es el resultado que se obtiene cuando se ejecuta el código anterior:

21.0

3.5

Traceback (última llamada más reciente):

Archivo "C:/zeroDivide.py", línea 6, en <módulo>

print(spam(0))

Archivo "C:/zeroDivide.py", línea 2, en spam

return 42 / divideBy

ZeroDivisionError: división por cero

Puede ver la ejecución de este programa en <https://autbor.com/zerodivide/> .

Un error *ZeroDivisionError* ocurre cada vez que intenta dividir un número por cero.

A partir del número de línea que se indica en el mensaje de error, sabe que la declaración de retorno en *spam()* está causando un error.

Los errores se pueden manejar con instrucciones *try* y *except* . El código que podría tener un error se coloca en una cláusula *try* . La ejecución del programa se mueve al inicio de la siguiente cláusula *except* si ocurre un error.

Puede colocar el código de división por cero anterior en una cláusula try y hacer que una cláusula except contenga código para manejar lo que sucede cuando ocurre este error.

```
def spam(divideBy):  
    try:  
        return 42 / divideBy  
    except ZeroDivisionError:  
        print('Error: argumento no válido.')
```

```
print(spam(2))  
print(spam(12))  
print(spam(0))  
print(spam(1))
```

Cuando el código de una cláusula try provoca un error, la ejecución del programa pasa inmediatamente al código de la cláusula except . Después de ejecutar ese código, la ejecución continúa de forma normal. El resultado del programa anterior es el siguiente:

```
21.0  
3.5  
Error: argumento no válido.  
Ninguno  
42.0
```

Puede ver la ejecución de este programa en <https://autbor.com/tryexceptzerodivide/> . Tenga en cuenta que también se detectarán los errores que se produzcan en las llamadas a funciones en un bloque try . Considere el siguiente programa, que en cambio tiene las llamadas spam() en el bloque try :

```
def spam(divideBy):  
    return 42 / divideBy  
  
intenta:  
    print(spam(2))  
    print(spam(12))  
    print(spam(0))  
    print(spam(1))  
excepto ZeroDivisionError:  
    print('Error: argumento no válido.')
```

Cuando se ejecuta este programa, el resultado se ve así:

21.0

3.5

Error: Argumento no válido.

Puede ver la ejecución de este programa en <https://autbor.com/spamintry/> . El motivo por el que `print(spam(1))` nunca se ejecuta es porque una vez que la ejecución salta al código en la cláusula `except` , no regresa a la cláusula `try` . En cambio, simplemente continúa avanzando en el programa de manera normal.

Un programa corto: Zigzag

Utilicemos los conceptos de programación que has aprendido hasta ahora para crear un pequeño programa de animación. Este programa creará un patrón en zigzag de ida y vuelta hasta que el usuario lo detenga presionando el botón Detener del editor Mu o presionando CTRL-C . Cuando ejecute este programa, el resultado será similar a esto:

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

Escriba el siguiente código fuente en el editor de archivos y guarde el archivo como `zigzag.py` :

```
import time, sys
indent = 0 # Cuántos espacios sangrar.
indentIncreasing = True # Si la sangría aumenta o no.
```

`try:`

```
    while True: # El bucle principal del programa.
        print(' ' * indent, end='')
        print('*****')
        time.sleep(0.1) # Pausa por 1/10 de segundo.
```

```
    if indentIncreasing:
        # Aumenta el número de espacios:
        indent = indent + 1
        if indent == 20:
```

```

        # Cambia la dirección:
        indentIncreasing = False

    else:
        # Disminuye el número de espacios:
        indent = indent - 1
        if indent == 0:
            # Cambia la dirección:
            indentIncreasing = True
except KeyboardInterrupt:
    sys.exit()

```

Veamos este código línea por línea, comenzando desde arriba.

```

tiempo de importación, sys
indent = 0 # Cuántos espacios sangrar.
indentIncreasing = True # Si la sangría es creciente o no.

```

Primero, importaremos los módulos time y sys . Nuestro programa utiliza dos variables: la variable indent registra cuántos espacios de sangría hay antes de la banda de ocho asteriscos e indentIncreasing contiene un valor booleano para determinar si la cantidad de sangría aumenta o disminuye.

```

try:
    while True: # El bucle principal del programa.
        print(' ' * indent, end="")
        print('*****')
        time.sleep(0.1) # Pausa por 1/10 de segundo.

```

A continuación, colocamos el resto del programa dentro de una sentencia try. Cuando el usuario presiona CTRL-C mientras se ejecuta un programa Python, Python genera la excepción KeyboardInterrupt . Si no hay una sentencia try - except para capturar esta excepción, el programa se bloquea con un desagradable mensaje de error. Sin embargo, para nuestro programa, queremos que maneje de manera limpia la excepción KeyboardInterrupt llamando a sys.exit() . (El código para esto está en la sentencia except al final del programa).

El bucle while True: infinite repetirá las instrucciones de nuestro programa para siempre. Esto implica usar ' ' \* indent para imprimir la cantidad correcta de espacios de sangría. No queremos imprimir automáticamente una nueva línea después de estos espacios, por lo que también pasamos end="" a la primera llamada print() . Una segunda llamada print() imprime la banda de asteriscos. La función time.sleep() aún no se ha cubierto, pero basta con decir que introduce una pausa de una décima de segundo en nuestro programa en este punto.

si indentIncreasing:

# Aumenta el número de espacios:

indent = indent + 1

si indent == 20:

indentIncreasing = False # Cambia de dirección.

A continuación, queremos ajustar la cantidad de sangría para la próxima vez que imprimamos asteriscos. Si indentIncreasing es True , entonces queremos agregar uno a indent . Pero una vez que indent alcance 20 , queremos que la sangría disminuya.

de lo contrario:

# Disminuir el número de espacios:

sangría = sangría - 1

si sangría == 0:

indentIncreasing = True # Cambiar dirección.

Mientras tanto, si indentIncreasing era False , queremos restar uno de indent . Una vez que indent llega a 0 , queremos que la sangría aumente una vez más. De cualquier manera, la ejecución del programa saltará de nuevo al inicio del bucle principal del programa para imprimir los asteriscos nuevamente.

excepto KeyboardInterrupt:

sys.exit()

Si el usuario presiona CTRL-C en cualquier momento en que la ejecución del programa se encuentre en el bloque try , se genera la excepción KeyboardInterrupt y esta instrucción except la maneja . La ejecución del programa se mueve dentro del bloque except , que ejecuta sys.exit() y cierra el programa. De esta manera, aunque el bucle principal del programa sea un bucle infinito, el usuario tiene una forma de cerrar el programa.

## LISTAS





Otro tema que deberá comprender antes de comenzar a escribir programas en serio es el tipo de datos de lista y su primo, la tupla. Las listas y las tuplas pueden contener múltiples valores, lo que facilita la escritura de programas que manejan grandes cantidades de datos. Y dado que las listas pueden contener otras listas, puede usarlas para organizar los datos en estructuras jerárquicas.

En este capítulo, analizaré los conceptos básicos de las listas. También te enseñaré sobre los métodos, que son funciones vinculadas a valores de un determinado tipo de datos. Luego, cubriré brevemente los tipos de datos de secuencia (listas, tuplas y cadenas) y mostraré cómo se comparan entre sí. En el próximo capítulo, te presentaré el tipo de datos de diccionario.

## El tipo de datos de lista

Una *lista* es un valor que contiene varios valores en una secuencia ordenada. El término *valor de lista* se refiere a la lista en sí (que es un valor que se puede almacenar en una variable o pasar a una función como cualquier otro valor), no a los valores dentro del valor de lista. Un valor de lista se ve así: ['cat', 'bat', 'rat', 'elephant'] . Así como los valores de cadena se escriben con caracteres de comillas para marcar dónde comienza y termina la cadena, una lista comienza con un corchete de apertura y termina con un corchete de cierre, [] . Los valores dentro de la lista también se denominan *items* . Los items se separan con comas (es decir, están *delimitados por comas* ). Por ejemplo, ingrese lo siguiente en el shell interactivo:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['gato', 'murciélago', 'rata', 'elefante']
['gato', 'murciélago', 'rata', 'elefante']
>>> ['hola', 3.1415, Verdadero, Ninguno, 42]
['hola', 3.1415, Verdadero, Ninguno, 42]
❶ >>> spam = ['gato', 'murciélago', 'rata', 'elefante']
>>> spam
['gato', 'murciélago', 'rata', 'elefante']
```

A la variable spam ❶ todavía se le asigna un solo valor: el valor de la lista. Pero el valor de la lista en sí contiene otros valores. El valor [] es una lista vacía que no contiene valores, similar a " , la cadena vacía.

## Obtención de valores individuales en una lista con índices

Digamos que tienes la lista ['cat', 'murciélago', 'rata', 'elefante'] almacenada en una variable llamada spam . El código Python spam[0] se evaluaría como 'cat' , y spam[1] se evaluaría como 'murciélago' , y así sucesivamente. El entero dentro

de los corchetes que sigue a la lista se llama *índice* . El primer valor de la lista está en el índice 0 , el segundo valor está en el índice 1 , el tercer valor está en el índice 2 , y así sucesivamente. [La Figura 4-1](#) muestra un valor de lista asignado a spam , junto con lo que evaluarían las expresiones de índice. Ten en cuenta que debido a que el primer índice es 0 , el último índice es uno menos que el tamaño de la lista; una lista de cuatro elementos tiene 3 como su último índice.

```
spam = ["cat", "bat", "rat", "elephant"]  
      ↑      ↑      ↑      ↑  
    spam[0] spam[1] spam[2] spam[3]
```

*Figura 4-1: Un valor de lista almacenado en la variable spam , que muestra a qué valor se refiere cada índice*

Por ejemplo, introduzca las siguientes expresiones en el shell interactivo. Comience asignando una lista a la variable spam .

```
>>> spam = ['gato', 'murciélago', 'rata', 'elefante']  
>>> spam[0]  
'gato'  
>>> spam[1]  
'murciélago'  
>>> spam[2]  
'rata'  
>>> spam[3]  
'elefante'  
>>> ['gato', 'murciélago', 'rata', 'elefante'][3]  
'elefante'  
❶ >>> 'Hola, ' + spam[0]  
❷ 'Hola, gato'  
>>> 'El ' + spam[1] + ' se comió al ' + spam[0] + '  
'El murciélago se comió al gato.'
```

Tenga en cuenta que la expresión 'Hola, ' + spam[0] ❶ se evalúa como 'Hola, ' + 'gato' porque spam[0] se evalúa como la cadena 'gato' . Esta expresión, a su vez, se evalúa como el valor de cadena 'Hola, gato' ❷ .

Python le dará un mensaje de error IndexError si utiliza un índice que excede la cantidad de valores en su valor de lista.

```
>>> spam = ['gato', 'murciélago', 'rata', 'elefante']  
>>> spam[10000]  
Traceback (última llamada reciente):  
  Archivo "<pyshell#9>", línea 1, en <módulo>
```

```
spam[10000]
```

IndexError: índice de lista fuera de rango

Los índices solo pueden ser valores enteros, no flotantes. El siguiente ejemplo provocará un error TypeError :

```
>>> spam = ['cat', 'murciélagos', 'rata', 'elefante']
```

```
>>> spam[1]
```

```
'murciélagos'
```

```
>>> spam[1.0]
```

Traceback (última llamada reciente):

Archivo "<pyshell#13>", línea 1, en <módulo>

```
spam[1.0]
```

TypeError: los índices de lista deben ser números enteros o porciones, no flotantes

```
>>> spam[int(1.0)]
```

```
'murciélagos'
```

Las listas también pueden contener otros valores de lista. Se puede acceder a los valores de estas listas de listas mediante varios índices, de la siguiente manera:

```
>>> spam = [['gato', 'murciélagos'], [10, 20, 30, 40, 50]]
```

```
>>> spam[0]
```

```
['gato', 'murciélagos']
```

```
>>> spam[0][1]
```

```
'murciélagos'
```

```
>>> spam[1][4]
```

```
50
```

El primer índice indica qué valor de lista utilizar y el segundo indica el valor dentro de la lista. Por ejemplo, spam[0][1] imprime 'bat' , el segundo valor de la primera lista. Si solo utiliza un índice, el programa imprimirá el valor de lista completo en ese índice.

### ***Índices negativos***

Si bien los índices comienzan en 0 y van aumentando, también puede utilizar números enteros negativos para el índice. El valor entero -1 se refiere al último índice de una lista, el valor -2 se refiere al segundo índice anterior a la última lista, y así sucesivamente. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélagos', 'rata', 'elefante']
```

```
>>> spam[-1]
```

```
'elefante'
```

```
>>> spam[-3]
```

```
'murciélagos'
```

```
>>> 'El ' + spam[-1] + ' tiene miedo del ' + spam[-3] + '  
'El elefante tiene miedo del murciélago.'
```

### ***Obtener una lista a partir de otra lista con segmentos***

Así como un índice puede obtener un único valor de una lista, una *porción* puede obtener varios valores de una lista, en forma de una nueva lista. Una porción se escribe entre corchetes, como un índice, pero tiene dos números enteros separados por dos puntos. Observe la diferencia entre índices y porciones.

- spam[2] es una lista con un índice (un entero).
- spam[1:4] es una lista con una porción (dos números enteros).

En una porción, el primer entero es el índice donde comienza la porción. El segundo entero es el índice donde termina la porción. Una porción llega hasta el valor del segundo índice, pero no lo incluye. Una porción se evalúa como un nuevo valor de lista. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélago', 'rata', 'elefante']  
>>> spam[0:4]  
['gato', 'murciélago', 'rata', 'elefante']  
>>> spam[1:3]  
['murciélago', 'rata']  
>>> spam[0:-1]  
['gato', 'murciélago', 'rata']
```

Como atajo, puede omitir uno o ambos índices a cada lado de los dos puntos en la porción. Omitir el primer índice es lo mismo que usar 0 , o el comienzo de la lista. Omitir el segundo índice es lo mismo que usar la longitud de la lista, lo que hará que la porción llegue hasta el final de la lista. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélago', 'rata', 'elefante']  
>>> spam[:2]  
['gato', 'murciélago']  
>>> spam[1:]  
['murciélago', 'rata', 'elefante']  
>>> spam[:]  
['gato', 'murciélago', 'rata', 'elefante']
```

### ***Obtención de la longitud de una lista con la función len()***

La función len() devolverá la cantidad de valores que hay en un valor de lista que se le pasa, de la misma manera que puede contar la cantidad de caracteres en un valor de cadena. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'perro', 'alce']
>>> len(spam)
3
```

### ***Cambiar valores en una lista con índices***

Normalmente, el nombre de una variable va en el lado izquierdo de una declaración de asignación, como `spam = 42`. Sin embargo, también puede utilizar un índice de una lista para cambiar el valor en ese índice. Por ejemplo, `spam[1] = 'aardvark'` significa “Asignar el valor en el índice 1 de la lista `spam` a la cadena `'aardvark'`”. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélago', 'rata', 'elefante']
>>> spam[1] = 'oso hormiguero'
>>> spam
['gato', 'oso hormiguero', 'rata', 'elefante']
>>> spam[2] = spam[1]
>>> spam
['gato', 'oso hormiguero', 'oso hormiguero', 'elefante']
>>> spam[-1] = 12345
>>> spam
['gato', 'oso hormiguero', 'oso hormiguero', 12345]
```

### ***Concatenación de listas y replicación de listas***

Las listas se pueden concatenar y replicar como cadenas. El operador `+` combina dos listas para crear un nuevo valor de lista y el operador `*` se puede utilizar con una lista y un valor entero para replicar la lista. Ingrese lo siguiente en el shell interactivo:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

### ***Cómo eliminar valores de listas con instrucciones del***

La instrucción `del` eliminará los valores de un índice de una lista. Todos los valores de la lista posteriores al valor eliminado se moverán un índice hacia arriba. Por ejemplo, ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélagos', 'rata', 'elefante']
>>> del spam[2]
>>> spam
['gato', 'murciélagos', 'elefante']
>>> del spam[2]
>>> spam
['gato', 'murciélagos']
```

La sentencia `del` también se puede utilizar en una variable simple para eliminarla, como si fuera una sentencia de “desasignación”. Si intenta utilizar la variable después de eliminarla, obtendrá un error `NameError` porque la variable ya no existe. En la práctica, casi nunca es necesario eliminar variables simples. La sentencia `del` se utiliza principalmente para eliminar valores de listas.

### Trabajar con listas

Cuando empiezas a escribir programas, resulta tentador crear muchas variables individuales para almacenar un grupo de valores similares. Por ejemplo, si quisiera almacenar los nombres de mis gatos, podría sentirme tentado a escribir un código como este:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Cola gorda'
catName6 = 'Señorita Cleo'
```

Resulta que esta es una mala manera de escribir código. (Además, en realidad no tengo tantos gatos, lo juro). Por un lado, si el número de gatos cambia, tu programa nunca podrá almacenar más gatos que variables. Este tipo de programas también tienen mucho código duplicado o casi idéntico. Considera cuánto código duplicado hay en el siguiente programa, que debes ingresar en el editor de archivos y guardar como *allMyCats1.py*:

```
print('Ingresa el nombre del gato 1:')
catName1 = input()
print('Ingresa el nombre del gato 2:')
catName2 = input()
print('Ingresa el nombre del gato 3:')
catName3 = input()
print('Ingresa el nombre del gato 4:')
catName4 = input()
print('Ingresa el nombre del gato 5:')
```

```

catName5 = input()
print('Ingrese el nombre del gato 6:')
catName6 = input()
print('Los nombres de los gatos son:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)

```

En lugar de utilizar múltiples variables repetitivas, puede utilizar una única variable que contenga un valor de lista. Por ejemplo, aquí se muestra una versión nueva y mejorada del programa *allMyCats1.py*. Esta nueva versión utiliza una única lista y puede almacenar cualquier cantidad de gatos que el usuario escriba. En una nueva ventana del editor de archivos, introduzca el siguiente código fuente y guárdelo como *allMyCats2.py*:

```

catNames = []
while True:
    print('Ingrese el nombre del gato ' + str(len(catNames) + 1) +
        ' (O no ingrese nada para detener.):')
    name = input()
    if name == "":
        break
    catNames = catNames + [name] # concatenación de listas
print('Los nombres de los gatos son:')
for name in catNames:
    print(' ' + name)

```

Cuando ejecute este programa, el resultado será similar a esto:

Ingresar el nombre del gato 1 (o no ingresar nada para detenerlo):

**Zophie**

Ingresar el nombre del gato 2 (o no ingresar nada para detenerlo):

**Pooka**

Ingresar el nombre del gato 3 (o no ingresar nada para detenerlo):

**Simon**

Ingresar el nombre del gato 4 (o no ingresar nada para detenerlo):

**Lady Macbeth**

Ingresar el nombre del gato 5 (o no ingresar nada para detenerlo):

**Fat-tail**

Ingresar el nombre del gato 6 (o no ingresar nada para detenerlo):

**Miss Cleo**

Ingresar el nombre del gato 7 (o no ingresar nada para detenerlo):

Los nombres de los gatos son:

Zophie  
Pooka  
Simon  
Lady Macbeth  
Fat-tail  
Miss Cleo

Puede ver la ejecución de estos programas en <https://autbor.com/allmycats1/> y <https://autbor.com/allmycats2/> . La ventaja de usar una lista es que sus datos ahora están en una estructura, por lo que su programa es mucho más flexible al procesar los datos de lo que sería con varias variables repetitivas.

### ***Uso de bucles for con listas***

En [el Capítulo 2](#) , aprendiste a usar bucles for para ejecutar un bloque de código una cierta cantidad de veces. Técnicamente, un bucle for repite el bloque de código una vez para cada elemento de un valor de lista. Por ejemplo, si ejecutaste este código:

```
para i en rango(4):  
    print(i)
```

La salida de este programa sería la siguiente:

```
0  
1  
2  
3
```

Esto se debe a que el valor de retorno de `range(4)` es un valor de secuencia que Python considera similar a `[0, 1, 2, 3]` . (Las secuencias se describen en “ [Tipos de datos de secuencia](#) ” en [la página 93](#) ). El siguiente programa tiene el mismo resultado que el anterior:

```
para i en [0, 1, 2, 3]:  
    print(i)
```

El bucle for anterior en realidad recorre su cláusula con la variable `i` establecida en un valor sucesivo en la lista `[0, 1, 2, 3]` en cada iteración.

Una técnica común de Python es usar `range(len( someList ))` con un bucle for para iterar sobre los índices de una lista. Por ejemplo, ingrese lo siguiente en el shell interactivo:

```
>>> suministros = ['bolígrafos', 'grapadoras', 'lanzallamas', 'carpetas']  
>>> for i in range(len(suministros)):
```



```
... print('Índice ' + str(i) + ' en suministros es: ' + suministros[i])
```

El índice 0 en suministros es: bolígrafos

El índice 1 en suministros es: grapadoras

El índice 2 en suministros es: lanzallamas

El índice 3 en suministros es: carpetas

El uso de `range(len(supplies))` en el bucle `for` mostrado anteriormente es útil porque el código en el bucle puede acceder al índice (como la variable `i`) y al valor en ese índice (como `supplies[i]`). Lo mejor de todo es que `range(len(supplies))` iterará a través de todos los índices de `supplies`, sin importar cuántos elementos contenga.

### **Los operadores `in` y `not in`**

Puede determinar si un valor está o no en una lista con los operadores `in` y `not in`. Al igual que otros operadores, `in` y `not in` se utilizan en expresiones y conectan dos valores: un valor que se buscará en una lista y la lista donde puede estar encontrado. Estas expresiones se evaluarán como un valor booleano. Ingrese lo siguiente en el shell interactivo:

```
>>> 'howdy' en ['hola', 'hola', 'howdy', 'heyas']
```

Verdadero

```
>>> spam = ['hola', 'hola', 'howdy', 'heyas']
```

```
>>> 'cat' en spam
```

Falso

```
>>> 'howdy' no en spam
```

Falso

```
>>> 'cat' no en spam
```

Verdadero

Por ejemplo, el siguiente programa permite al usuario escribir el nombre de una mascota y luego verifica si el nombre está en una lista de mascotas. Abra una nueva ventana del editor de archivos, ingrese el siguiente código y guárdelo como `myPets.py`:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Ingrese un nombre de mascota:')
name = input()
if name not in myPets:
    print('No tengo una mascota llamada ' + name)
else:
    print(name + ' es mi mascota.')
```

El resultado podría verse así:

Ingrese un nombre de mascota:

**Footfoot**

No tengo una mascota llamada Footfoot

Puedes ver la ejecución de este programa en <https://autbor.com/mypets/> .

### ***El truco de las tareas múltiples***

El *truco de asignación múltiple* (técnicamente llamado *desempaquetado de tuplas* ) es un atajo que te permite asignar múltiples variables con los valores de una lista en una línea de código. Entonces, en lugar de hacer esto:

```
>>> gato = ['gordo', 'gris', 'ruidoso']
>>> tamaño = gato[0]
>>> color = gato[1]
>>> disposición = gato[2]
```

Podrías escribir esta línea de código:

```
>>> gato = ['gordo', 'gris', 'ruidoso']
>>> tamaño, color, disposición = gato
```

El número de variables y la longitud de la lista deben ser exactamente iguales, o Python le dará un ValueError :

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition, name = cat
```

Traceback (última llamada reciente):

Archivo "<pyshell#84>", línea 1, en <módulo>

size, color, disposition, name = cat

ValueError: no hay suficientes valores para descomprimir (se esperaban 4, se obtuvieron 3)

### ***Uso de la función enumerate() con listas***

En lugar de utilizar la técnica `range(len( someList ))` con un bucle for para obtener el índice entero de los elementos de la lista, puede llamar a la función `enumerate()` . En cada iteración del bucle, `enumerate()` devolverá dos valores: el índice del elemento de la lista y el elemento de la lista en sí. Por ejemplo, este código es equivalente al código de “ [Uso de bucles for con listas](#) ” en [la página 84](#) :

```
>>> suministros = ['bolígrafos', 'grapadoras', 'lanzallamas', 'carpetas']
>>> for index, item in enumerate(suministros):
...     print('Índice ' + str(índice) + ' en suministros es: ' + artículo)
```

El índice 0 en suministros es: bolígrafos

El índice 1 en suministros es: grapadoras  
El índice 2 en suministros es: lanzallamas  
El índice 3 en suministros es: carpetas

La función `enumerate()` es útil si necesita tanto el elemento como el índice del elemento en el bloque del bucle.

### ***Uso de las funciones `random.choice()` y `random.shuffle()` con listas***

El módulo `random` tiene un par de funciones que aceptan listas como argumentos. La función `random.choice()` devolverá un elemento seleccionado aleatoriamente de la lista. Ingrese lo siguiente en el shell interactivo:

```
>>> import random
>>> mascotas = ['Perro', 'Gato', 'Alce']
>>> random.choice(mascotas)
'Perro'
>>> random.choice(mascotas)
'Gato'
>>> random.choice(mascotas)
'Gato'
```

Puedes considerar `random.choice(someList)` como una forma más corta de `someList[random.randint(0, len(someList) - 1)]`.

La función `random.shuffle()` reordenará los elementos de una lista. Esta función modifica la lista en su lugar, en lugar de devolver una nueva lista. Ingrese lo siguiente en el shell interactivo:

```
>>> importar aleatorio
>>> personas = ['Alice', 'Bob', 'Carol', 'David']
>>> aleatorio.shuffle(personas)
>>> personas
['Carol', 'David', 'Alice', 'Bob']
>>> aleatorio.shuffle(personas)
>>> personas
['Alice', 'David', 'Bob', 'Carol']
```

### **Operadores de asignación aumentada**

Al asignar un valor a una variable, con frecuencia utilizará la variable en sí. Por ejemplo, después de asignar 42 a la variable `spam`, aumentaría el valor de `spam` en 1 con el siguiente código:

```
>>> correo basura = 42
>>> correo basura = correo basura + 1
```

```
>>> correo basura
```

```
43
```

Como atajo, puedes usar el operador de asignación aumentada += para hacer lo mismo:

```
>>> correo basura = 42
```

```
>>> correo basura += 1
```

```
>>> correo basura
```

```
43
```

Hay operadores de asignación aumentados para los operadores + , - , \* , / y % , descritos en [la Tabla 4-1](#) .

**Tabla 4-1:** Los operadores de asignación aumentada

Declaración de asignación aumentada	Declaración de asignación equivalente
correo no deseado += 1	correo basura = correo basura + 1
correo no deseado -= 1	correo basura = correo basura - 1
correo no deseado *= 1	correo basura = correo basura * 1
correo no deseado /= 1	correo basura = correo basura / 1
correo no deseado %= 1	correo no deseado = correo no deseado % 1

El operador += también puede realizar concatenaciones de cadenas y listas, y el operador \*= puede realizar replicaciones de cadenas y listas. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = 'Hola,'
```

```
>>> spam += ' mundo!'
```

```
>>> spam
```

```
'¡Hola mundo!'
```

```
>>> tocino = ['Zophie']
```

```
>>> tocino *= 3
```

```
>>> tocino
```

```
['Zophie', 'Zophie', 'Zophie']
```

## Métodos

Un *método* es lo mismo que una función, excepto que se “invoca” sobre un valor. Por ejemplo, si un valor de lista se almacenara en spam , se invocaría el método de lista index() (que explicaré en breve) en esa lista de la siguiente

manera: `spam.index('hello')` . La parte del método viene después del valor, separada por un punto.

Cada tipo de datos tiene su propio conjunto de métodos. El tipo de datos de lista, por ejemplo, tiene varios métodos útiles para buscar, agregar, eliminar y manipular valores en una lista.

### ***Cómo encontrar un valor en una lista con el método `index()`***

Los valores de lista tienen un método `index()` al que se le puede pasar un valor y, si ese valor existe en la lista, se devuelve el índice del valor. Si el valor no está en la lista, Python genera un error `ValueError` . Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['hola', 'hi', 'howdy', 'heyas']
>>> spam.index('hola')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (última llamada reciente):
  Archivo "<pyshell#31>", línea 1, en <módulo>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' no está en la lista
```

Cuando hay duplicados del valor en la lista, se devuelve el índice de su primera aparición. Ingrese lo siguiente en el shell interactivo y observe que `index()` devuelve 1 , no 3 :

```
>>> spam = ['Zophie', 'Pooka', 'Cola Gorda', 'Pooka']
>>> spam.index('Pooka')
1
```

### ***Cómo agregar valores a listas con los métodos `append()` e `insert()`***

Para agregar nuevos valores a una lista, utilice los métodos `append()` e `insert()` . Ingrese lo siguiente en el shell interactivo para llamar al método `append()` en un valor de lista almacenado en la variable `spam` :

```
>>> spam = ['gato', 'perro', 'murciélago']
>>> spam.append('alce')
>>> spam
['gato', 'perro', 'murciélago', 'alce']
```

La llamada anterior al método `append()` agrega el argumento al final de la lista. El método `insert()` puede insertar un valor en cualquier índice de la lista. El primer

argumento de insert() es el índice del nuevo valor y el segundo argumento es el nuevo valor que se insertará. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'perro', 'murciélago']
>>> spam.insert(1, 'pollo' )
>>> spam
['gato', 'pollo', 'perro', 'murciélago']
```

Tenga en cuenta que el código es spam.append('moose') y spam.insert(1, 'chicken') , no spam = spam.append('moose') y spam = spam.insert(1, 'chicken') . Ni append() ni insert() proporcionan el nuevo valor de spam como su valor de retorno. (De hecho, el valor de retorno de append() e insert() es None , por lo que definitivamente no querrá almacenar esto como el nuevo valor de la variable). En cambio, la lista se modifica *en el lugar* . La modificación de una lista en el lugar se cubre con más detalle más adelante en “ [Tipos de datos mutables e inmutables](#) ” en [la página 94](#) .

Los métodos pertenecen a un único tipo de datos. Los métodos append() e insert() son métodos de lista y se pueden llamar solo en valores de lista, no en otros valores como cadenas o números enteros. Ingrese lo siguiente en el shell interactivo y observe los mensajes de error AttributeError que aparecen:

```
>>> huevos = 'hola'
>>> huevos.append('mundo')
Traceback (última llamada más reciente):
  Archivo "<pyshell#19>", línea 1, en <módulo>
    huevos.append('mundo')
AttributeError: el objeto 'str' no tiene el atributo 'append'
>>> tocino = 42
>>> tocino.insert(1, 'mundo')
Traceback (última llamada más reciente):
  Archivo "<pyshell#22>", línea 1, en <módulo>
    tocino.insert(1, 'mundo')
AttributeError: el objeto 'int' no tiene el atributo 'insert'
```

### ***Cómo eliminar valores de listas con el método remove()***

Al método remove() se le pasa el valor que se eliminará de la lista en la que se lo llama. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélago', 'rata', 'elefante']
>>> spam.remove('murciélago')
>>> spam
['gato', 'rata', 'elefante']
```

Si intenta eliminar un valor que no existe en la lista, se generará un error `ValueError` . Por ejemplo, ingrese lo siguiente en el shell interactivo y observe el error que se muestra:

```
>>> spam = ['gato', 'murciélagos', 'rata', 'elefante']
```

```
>>> spam.remove('pollo')
```

Traceback (última llamada más reciente):

Archivo "<pyshell#11>", línea 1, en <módulo>

spam.remove('pollo')

`ValueError: list.remove(x): x no está en la lista`

Si el valor aparece varias veces en la lista, solo se eliminará la primera instancia del valor. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'murciélagos', 'rata', 'gato', 'sombrero', 'gato']
```

```
>>> spam.remove('gato')
```

```
>>> spam
```

```
['murciélagos', 'rata', 'gato', 'sombrero', 'gato']
```

La instrucción `del` es útil cuando se conoce el índice del valor que se desea eliminar de la lista. El método `remove()` es útil cuando se conoce el valor que se desea eliminar de la lista.

### ***Ordenar los valores de una lista con el método `sort()`***

Las listas de valores numéricos o listas de cadenas se pueden ordenar con el método `sort()` . Por ejemplo, introduzca lo siguiente en el shell interactivo:

```
>>> spam = [2, 5, 3.14, 1, -7]
```

```
>>> spam.sort()
```

```
>>> spam
```

```
[-7, 1, 2, 3.14, 5]
```

```
>>> spam = ['hormigas', 'gatos', 'perros', 'tejones', 'elefantes']
```

```
>>> spam.sort()
```

```
>>> spam
```

```
['hormigas', 'tejones', 'gatos', 'perros', 'elefantes']
```

También puede pasar `True` como argumento de palabra clave inversa para que `sort()` ordene los valores en orden inverso. Ingrese lo siguiente en el shell interactivo:

```
>>> spam.sort(reverse=True)
```

```
>>> spam
```

```
['elefantes', 'perros', 'gatos', 'tejones', 'hormigas']
```

Hay tres cosas que debes tener en cuenta sobre el método `sort()` . En primer lugar, el método `sort()` ordena la lista en su lugar; no intentes capturar el valor de retorno escribiendo código como `spam = spam.sort()` .

En segundo lugar, no se pueden ordenar listas que contengan valores numéricos y de cadena, ya que Python no sabe cómo comparar estos valores. Introduzca lo siguiente en el shell interactivo y observe el error `TypeError` :

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (última llamada reciente):
  Archivo "<pyshell#70>", línea 1, en <módulo>
    spam.sort()
TypeError: '<' no es compatible entre instancias de 'str' e 'int'
```

En tercer lugar, `sort()` utiliza el “orden ASCIIbético” en lugar del orden alfabético real para ordenar las cadenas. Esto significa que las letras mayúsculas van antes que las minúsculas. Por lo tanto, la *a* minúscula se ordena de modo que vaya *después de* la *Z* mayúscula . Por ejemplo, ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['Alicia', 'hormigas', 'Bob', 'tejones', 'Carol', 'gatos']
>>> spam.sort()
>>> spam
['Alicia', 'Bob', 'Carol', 'hormigas', 'tejones', 'gatos']
```

Si necesita ordenar los valores en orden alfabético regular, pase `str.lower` como argumento de palabra clave en la llamada al método `sort()` .

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

Esto hace que la función `sort()` trate todos los elementos de la lista como si fueran minúsculas sin cambiar realmente los valores de la lista.

### ***Cómo invertir los valores de una lista con el método `reverse()`***

Si necesita invertir rápidamente el orden de los elementos de una lista, puede llamar al método de lista `reverse()` . Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gato', 'perro', 'alce']
>>> spam.reverse()
>>> spam
['alce', 'perro', 'gato']
```



## EXCEPCIONES A LAS REGLAS DE INDENTACIÓN EN PYTHON

En la mayoría de los casos, la cantidad de sangría de una línea de código le indica a Python en qué bloque se encuentra. Sin embargo, existen algunas excepciones a esta regla. Por ejemplo, las listas pueden abarcar varias líneas en el archivo de código fuente. La sangría de estas líneas no importa; Python sabe que la lista no está terminada hasta que ve el corchete final. Por ejemplo, puede tener un código que se vea así:

```
spam = ['manzanas',
        'naranjas',
        'plátanos',
        'gatos']
print(spam)
```

Por supuesto, prácticamente hablando, la mayoría de las personas usan el comportamiento de Python para hacer que sus listas se vean bonitas y legibles, como la lista de mensajes en el programa Magic 8 Ball.

También puede dividir una sola instrucción en varias líneas utilizando el *carácter de continuación de línea* \ al final. Piense en \ como si dijera: "Esta instrucción continúa en la siguiente línea". La sangría en la línea después de una continuación de línea \ no es significativa. Por ejemplo, el siguiente es un código Python válido:

```
print('Hace ochenta y siete  
años...')
```

Estos trucos son útiles cuando quieres reorganizar líneas largas de código Python para que sean un poco más legibles.

Al igual que el método de lista `sort()` , `reverse()` no devuelve una lista. Por eso se escribe `spam.reverse()` en lugar de `spam = spam.reverse()` .

### Programa de ejemplo: Bola mágica 8 con una lista

Usando listas, puedes escribir una versión mucho más elegante del programa Magic 8 Ball del capítulo anterior. En lugar de varias líneas de instrucciones `elif` casi idénticas , puedes crear una única lista con la que el código funcione. Abre una nueva ventana del editor de archivos e ingresa el siguiente código. Guárdalo como *magic8Ball2.py* .

## importar mensajes aleatorios

- = ['Es cierto',  
'Es decididamente así',  
'Sí definitivamente',

```
'Respuesta confusa, inténtalo de nuevo',  
'Pregunta de nuevo más tarde',  
'Concéntrate y pregunta de nuevo',  
'Mi respuesta es no',  
'Las perspectivas no son muy buenas',  
'Muy dudoso']
```

```
print(messages[random.randint(0, len(messages) - 1)])
```

Puedes ver la ejecución de este programa en <https://autbor.com/magic8ball2/> .

Cuando ejecutes este programa, verás que funciona igual que el programa anterior *magic8Ball.py* .

Observa la expresión que utilizas como índice para `messages` : `random.randint(0, len(messages) - 1)` . Esto produce un número aleatorio para usar como índice, independientemente del tamaño de `messages` . Es decir, obtendrás un número aleatorio entre 0 y el valor de `len(messages) - 1` . El beneficio de este enfoque es que puedes agregar y eliminar cadenas fácilmente a la lista `messages` sin cambiar otras líneas de código. Si luego actualizas tu código, habrá menos líneas que tengas que cambiar y menos posibilidades de que introduzcas errores.

### **Tipos de datos de secuencia**

Las listas no son los únicos tipos de datos que representan secuencias ordenadas de valores. Por ejemplo, las cadenas y las listas son similares si consideramos que una cadena es una “lista” de caracteres de texto individuales. Los tipos de datos de secuencia de Python incluyen listas, cadenas, objetos de rango devueltos por `range()` y tuplas (explicados en “ [El tipo de datos de tupla](#) ” en [la página 96](#) ). Muchas de las cosas que se pueden hacer con las listas también se pueden hacer con cadenas y otros valores de tipos de secuencia: indexar, segmentar y utilizarlos con bucles `for` , con `len()` y con los operadores `in` y `not in` . Para ver esto, ingrese lo siguiente en el shell interactivo:

```
>>> nombre = 'Zophie'  
>>> nombre[0]  
'Z'  
>>> nombre[-2]  
'i'  
>>> nombre[0:4]  
'Zoph'  
>>> 'Zo' en nombre  
Verdadero  
>>> 'z' en nombre
```

Falso

```
>>> 'p' no en nombre
```

Falso

```
>>> para i en nombre:
```

```
...     print('***' + i + '***') *
```

```
**Z****
```

```
***O****
```

```
**p*****
```

```
*h*****
```

```
***j*****
```

```
**e****
```

### ***Tipos de datos mutables e inmutables***

Sin embargo, las listas y las cadenas son diferentes en un aspecto importante. Un valor de lista es un tipo de datos *mutable* : se le pueden agregar, quitar o cambiar valores. Sin embargo, una cadena es *inmutable* : no se puede cambiar. Intentar reasignar un solo carácter en una cadena da como resultado un error `TypeError` , como puede ver al ingresar lo siguiente en el shell interactivo:

```
>>> name = 'Zophie un gato'
```

```
>>> name[7] = 'the'
```

Traceback (última llamada reciente):

Archivo "<pyshell#50>", línea 1, en <módulo>

name[7] = 'the'

`TypeError`: el objeto 'str' no admite la asignación de elementos

La forma correcta de “mutar” una cadena es utilizar la segmentación y la concatenación para crear una *nueva* cadena copiando partes de la cadena anterior. Introduzca lo siguiente en el shell interactivo:

```
>>> nombre = 'Zophie una gata'
```

```
>>> nuevoNombre = nombre[0:7] + 'la' + nombre[8:12]
```

```
>>> nombre
```

```
'Zophie una gata'
```

```
>>> nuevoNombre
```

```
'Zophie la gata'
```

Usamos `[0:7]` y `[8:12]` para referirnos a los caracteres que no queremos reemplazar. Observe que la cadena original 'Zophie a cat' no se modifica, porque las cadenas son inmutables.

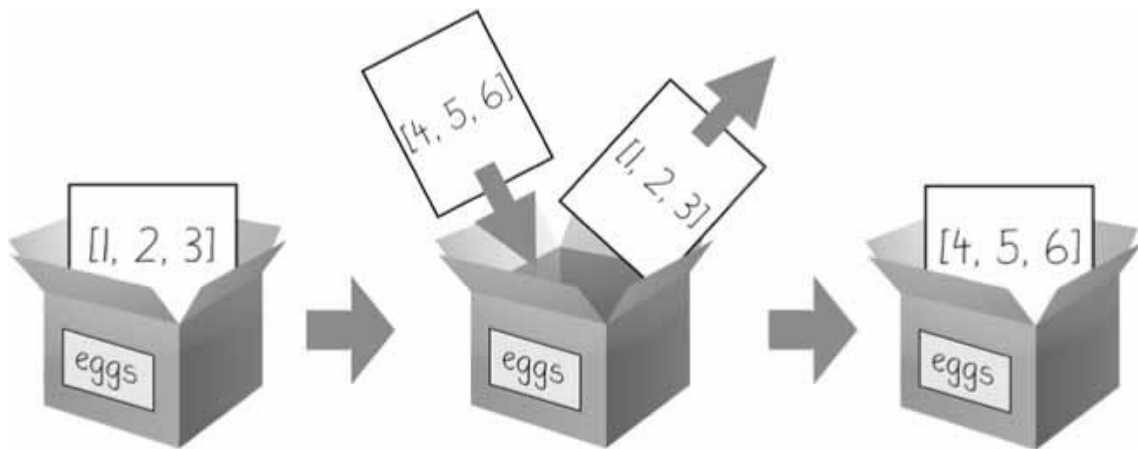
Aunque un valor de lista es mutable, la segunda línea del siguiente código no modifica la lista `eggs` :

```
>>> huevos = [1, 2, 3]
>>> huevos = [4, 5, 6]
>>> huevos
[4, 5, 6]
```

En este caso, no se modifica el valor de la lista de huevos ; en cambio, un valor de lista completamente nuevo y diferente ( [4, 5, 6] ) sobrescribe el valor de lista anterior ( [1, 2, 3] ). Esto se muestra en [la Figura 4-2](#) .

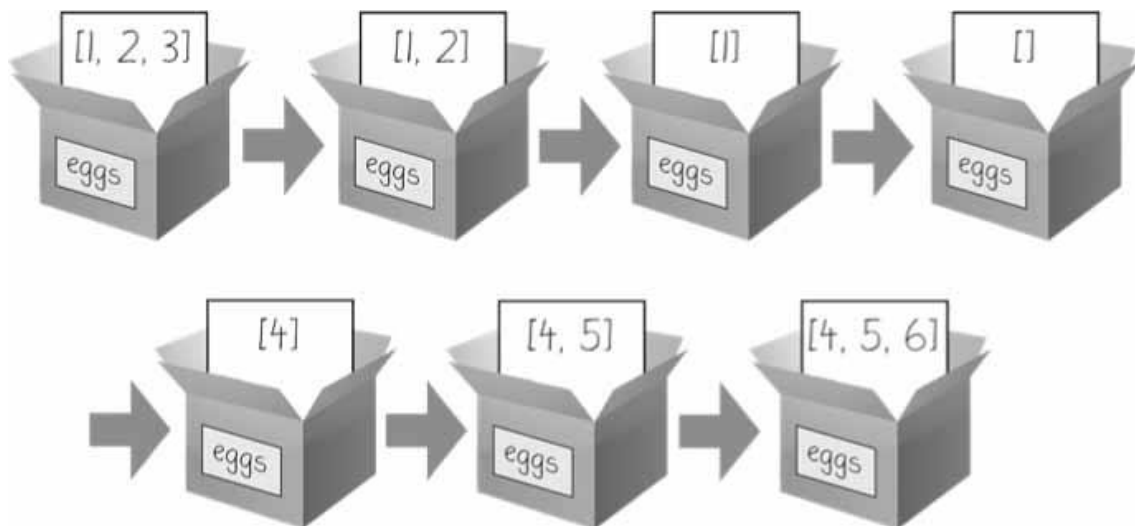
Si realmente quisieras modificar la lista original en huevos para que contenga [4, 5, 6] , tendrías que hacer algo como esto:

```
>>> huevos = [1, 2, 3]
>>> del huevos[2]
>>> del huevos[1]
>>> del huevos[0]
>>> huevos.append(4)
>>> huevos.append(5)
>>> huevos.append(6)
>>> huevos
[4, 5, 6]
```



*Figura 4-2: Cuando se ejecuta `eggs = [4, 5, 6]` , el contenido de `eggs` se reemplaza con un nuevo valor de lista.*

En el primer ejemplo, el valor de lista con el que termina `eggs` es el mismo valor de lista con el que comenzó. Solo que esta lista ha sido modificada, en lugar de sobrescribirse. [La Figura 4-3](#) muestra los siete cambios realizados en las primeras siete líneas del ejemplo de shell interactivo anterior.



*Figura 4-3: La declaración del y el método `append()` modifican el mismo valor de lista en su lugar.*

Cambiar un valor de un tipo de datos mutable (como lo hacen la declaración del y el método `append()` en el ejemplo anterior) cambia el valor en su lugar, ya que el valor de la variable no se reemplaza con un nuevo valor de lista.

Puede parecer que la distinción entre tipos mutables e inmutables no tiene sentido, pero en “ [Paso de referencias](#) ” en [la página 100](#) se explica el comportamiento diferente al llamar a funciones con argumentos mutables y con argumentos inmutables. Pero primero, conozcamos el tipo de datos tupla, que es una forma inmutable del tipo de datos lista.

### ***El tipo de datos Tupla***

El tipo de datos *de tupla* es casi idéntico al tipo de datos de lista, excepto en dos aspectos. En primer lugar, las tuplas se escriben con paréntesis ( y ) , en lugar de corchetes [ y ] . Por ejemplo, introduzca lo siguiente en el shell interactivo:

```
>>> huevos = ('hola', 42, 0.5)
>>> huevos[0]
'hola'
>>> huevos[1:3]
(42, 0.5)
>>> len(huevos)
3
```

Pero la principal diferencia entre las tuplas y las listas es que, al igual que las cadenas, son inmutables. No se pueden modificar, añadir ni eliminar los valores de las tuplas. Introduzca lo siguiente en el shell interactivo y observe el mensaje de error `TypeError` :

```
>>> huevos = ('hola', 42, 0.5)
```

```
>>> huevos[1] = 99
```

Traceback (última llamada más reciente):

Archivo "<pyshell#5>", línea 1, en <módulo>

```
    huevos[1] = 99
```

TypeError: el objeto 'tuple' no admite la asignación de elementos

Si solo tiene un valor en su tupla, puede indicarlo colocando una coma final después del valor dentro de los paréntesis. De lo contrario, Python pensará que acaba de escribir un valor dentro de paréntesis normales. La coma es lo que le permite a Python saber que se trata de un valor de tupla. (A diferencia de otros lenguajes de programación, está bien tener una coma final después del último elemento de una lista o tupla en Python). Ingrese las siguientes llamadas de función `type()` en el shell interactivo para ver la distinción:

```
>>> tipo(('hola',))
```

```
<clase 'tuple'>
```

```
>>> tipo('hola')
```

```
<clase 'str'>
```

Puedes usar tuplas para transmitir a cualquier persona que lea tu código que no tienes intención de que esa secuencia de valores cambie. Si necesitas una secuencia ordenada de valores que nunca cambie, usa una tupla. Un segundo beneficio de usar tuplas en lugar de listas es que, debido a que son inmutables y su contenido no cambia, Python puede implementar algunas optimizaciones que hacen que el código que usa tuplas sea ligeramente más rápido que el código que usa listas.

### ***Conversión de tipos con las funciones `list()` y `tuple()`***

De la misma manera que `str(42)` devolverá `'42'`, la representación en cadena del entero 42, las funciones `list()` y `tuple()` devolverán versiones en forma de lista y tupla de los valores que se les pasan. Ingrese lo siguiente en el shell interactivo y observe que el valor de retorno es de un tipo de datos diferente al valor pasado:

```
>>> tupla(['gato', 'perro', 5])
```

```
('gato', 'perro', 5)
```

```
>>> lista(['gato', 'perro', 5])
```

```
['gato', 'perro', 5]
```

```
>>> lista('hola')
```

```
['h', 'e', 'l', 'l', 'o']
```

Convertir una tupla en una lista es útil si necesita una versión mutable de un valor de tupla.

## Referencias

Como has visto, las variables “almacenan” cadenas y valores enteros. Sin embargo, esta explicación es una simplificación de lo que Python hace realmente. Técnicamente, las variables almacenan referencias a las ubicaciones de la memoria de la computadora donde se almacenan los valores. Introduce lo siguiente en el shell interactivo:

```
>>> spam = 42
>>> queso = spam
>>> spam = 100
>>> spam
100
>>> queso
42
```

Cuando asigna 42 a la variable `spam`, en realidad está creando el valor 42 en la memoria de la computadora y almacenando una *referencia* a él en la variable `spam`. Cuando copia el valor en `spam` y lo asigna a la variable `cheese`, en realidad está copiando la referencia. Tanto las variables `spam` como `cheese` hacen referencia al valor 42 en la memoria de la computadora. Cuando luego cambia el valor en `spam` a 100, está creando un nuevo valor 100 y almacenando una referencia a él en `spam`. Esto no afecta el valor en `cheese`. Los números enteros son valores *inmutables* que no cambian; cambiar la variable `spam` en realidad está haciendo que haga referencia a un valor completamente diferente en la memoria.

Pero las listas no funcionan de esta manera, porque los valores de las listas pueden cambiar; es decir, las listas son *mutables*. Aquí hay un código que hará que esta distinción sea más fácil de entender. Introdúzcalo en el shell interactivo:

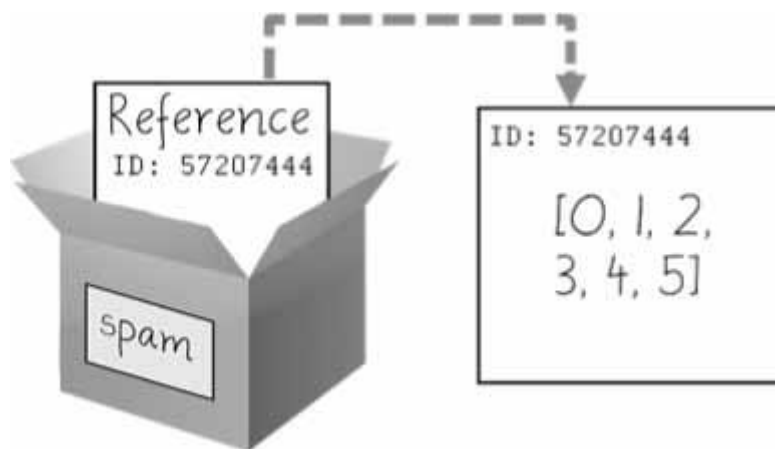
```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam # Se copia la referencia, no la lista.
❸ >>> cheese[1] = '¡Hola!' # Esto cambia el valor de la lista.
>>> spam
[0, '¡Hola!', 2, 3, 4, 5]
>>> queso # La variable queso hace referencia a la misma lista.
[0, '¡Hola!', 2, 3, 4, 5]
```

Puede que esto te parezca extraño. El código solo afectó a la lista de quesos, pero parece que tanto la lista de quesos como la de `spam` han cambiado.

Cuando creas la lista ❶, le asignas una referencia en la variable `spam`. Pero la siguiente línea ❷ copia solo la referencia de lista en `spam` a `cheese`, no el valor de

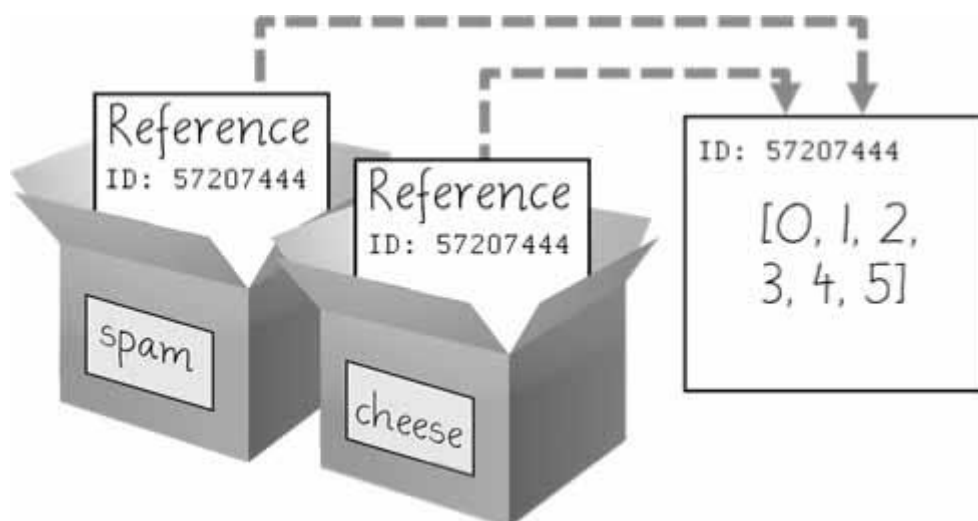
la lista en sí. Esto significa que los valores almacenados en spam y cheese ahora hacen referencia a la misma lista. Solo hay una lista subyacente porque la lista en sí nunca se copió. Entonces, cuando modificas el primer elemento de cheese ③ , estás modificando la misma lista a la que spam hace referencia.

Recuerde que las variables son como cajas que contienen valores. Las figuras anteriores de este capítulo muestran que las listas en cajas no son exactamente precisas, porque las variables de lista en realidad no contienen listas, sino *referencias* a listas. (Estas referencias tendrán números de identificación que Python usa internamente, pero puede ignorarlos). Usando cajas como metáfora para las variables, [la Figura 4-4](#) muestra lo que sucede cuando se asigna una lista a la variable spam .



*Figura 4-4: spam = [0, 1, 2, 3, 4, 5] almacena una referencia a una lista, no a la lista real.*

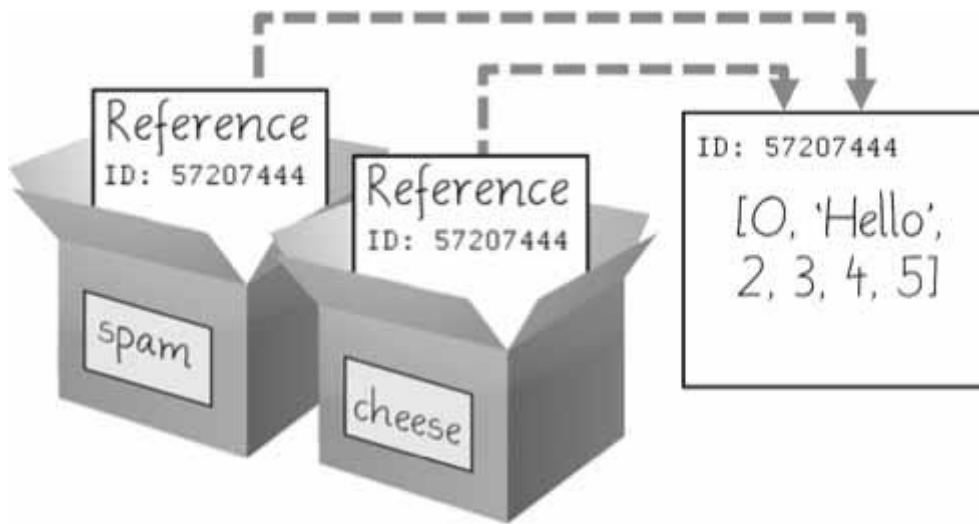
Luego, en [la Figura 4-5](#) , la referencia en spam se copia a cheese . Solo se creó y almacenó una nueva referencia en cheese , no una nueva lista. Observe cómo ambas referencias hacen referencia a la misma lista.



*Figura 4-5: spam=cheesy copia la referencia, no la lista.*



Cuando modificas la lista a la que hace referencia cheese , también cambia la lista a la que hace referencia spam , porque tanto cheese como spam hacen referencia a la misma lista. Puedes ver esto en [la Figura 4-6](#) .



*Figura 4-6: cheese[1] = '¡Hola!' modifica la lista a la que hacen referencia ambas variables.*

Aunque técnicamente las variables de Python contienen referencias a valores, la gente suele decir casualmente que la variable contiene el valor.

### **Identidad y la función id()**

Quizás te preguntes por qué el comportamiento extraño con las listas mutables de la sección anterior no ocurre con valores inmutables como números enteros o cadenas. Podemos usar la función `id()` de Python para entender esto. Todos los valores en Python tienen una identidad única que se puede obtener con la función `id()` . Introduce lo siguiente en el shell interactivo:

```
>>> id('Howdy') # El número devuelto será diferente en su máquina.  
44491136
```

Cuando Python ejecuta `id('Howdy')` , crea la cadena 'Howdy' en la memoria de la computadora. La función `id()` devuelve la dirección de memoria numérica donde se almacena la cadena . Python elige esta dirección en función de los bytes de memoria que estén libres en la computadora en ese momento, por lo que será diferente cada vez que ejecute este código.

Como todas las cadenas, 'Howdy' es inmutable y no se puede cambiar. Si "cambias" la cadena en una variable, se crea un nuevo objeto de cadena en un lugar diferente de la memoria y la variable hace referencia a esta nueva cadena. Por ejemplo, ingresa lo siguiente en el shell interactivo y observa cómo cambia la identidad de la cadena a la que hace referencia bacon :

```
>>> bacon = 'Hola'
>>> id(bacon)
44491136
>>> bacon += ' ¡mundo!' # Se crea una nueva cadena a partir de 'Hola' y ' ¡mundo!'.
>>> id(bacon) # bacon ahora hace referencia a una cadena completamente
diferente.
44609712
```

Sin embargo, las listas se pueden modificar porque son objetos mutables. El método `append()` no crea un nuevo objeto de lista, sino que cambia el objeto de lista existente. A esto lo llamamos “modificar el objeto *en el lugar*”.

```
>>> eggs = ['cat', 'dog'] # Esto crea una nueva lista.
>>> id(eggs)
35152584
>>> eggs.append('moose') # append() modifica la lista "en su lugar".
>>> id(eggs) # eggs todavía hace referencia a la misma lista que antes.
35152584
>>> eggs = ['bat', 'rat', 'cow'] # Esto crea una nueva lista, que tiene una nueva
identidad.
>>> id(eggs) # eggs ahora hace referencia a una lista completamente diferente.
44409800
```

Si dos variables hacen referencia a la misma lista (como `spam` y `cheese` en la sección anterior) y el valor de la lista cambia, ambas variables se ven afectadas porque ambas hacen referencia a la misma lista. Los métodos `append()`, `extend()`, `remove()`, `sort()`, `reverse()` y otros métodos de lista modifican sus listas en el lugar.

*El recolector de basura automático* de Python elimina todos los valores a los que no hace referencia ninguna variable para liberar memoria. No es necesario preocuparse por cómo funciona el recolector de basura, lo cual es bueno: la gestión manual de la memoria en otros lenguajes de programación es una fuente común de errores.

### **Referencias de paso**

Las referencias son particularmente importantes para comprender cómo se pasan los argumentos a las funciones. Cuando se llama a una función, los valores de los argumentos se copian en las variables de los parámetros. En el caso de las listas (y los diccionarios, que describiré en el próximo capítulo), esto significa que se utiliza una copia de la referencia para el parámetro. Para ver las consecuencias de esto, abra una nueva ventana del editor de archivos, ingrese el siguiente código y guárdelo como *passingReference.py*:

```
def huevos(algunParámetro):  
    algunParámetro.append('Hola')
```

```
spam = [1, 2, 3]  
huevos(spam)  
print(spam)
```

Tenga en cuenta que cuando se llama a `eggs()` , no se utiliza un valor de retorno para asignar un nuevo valor a `spam` . En cambio, modifica la lista existente, directamente. Cuando se ejecuta, este programa produce el siguiente resultado:

```
[1, 2, 3, 'Hola']
```

Aunque `spam` y `someParameter` contienen referencias independientes, ambos hacen referencia a la misma lista. Por este motivo, la llamada al método `append('Hello')` dentro de la función afecta a la lista incluso después de que la llamada a la función haya regresado.

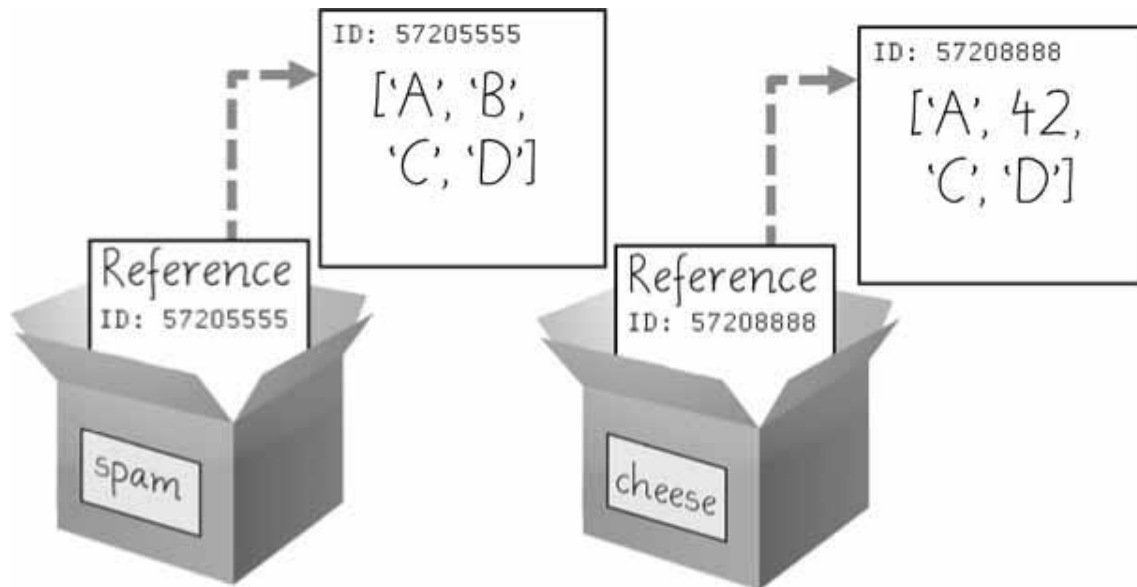
Tenga en cuenta este comportamiento: olvidar que Python maneja las variables de lista y diccionario de esta manera puede generar errores confusos.

### ***Las funciones `copy()` y `deepcopy()` del módulo `copy`***

Aunque pasar referencias suele ser la forma más práctica de trabajar con listas y diccionarios, si la función modifica la lista o el diccionario que se pasa, es posible que no desees que estos cambios se produzcan en el valor original de la lista o el diccionario. Para ello, Python proporciona un módulo llamado `copy` que proporciona las funciones `copy()` y `deepcopy()` . La primera de ellas, `copy.copy()` , se puede utilizar para hacer una copia duplicada de un valor mutable como una lista o un diccionario, no solo una copia de una referencia. Introduce lo siguiente en el shell interactivo:

```
>>> import copy  
>>> spam = ['A', 'B', 'C', 'D']  
>>> id(spam)  
44684232  
>>> cheese = copy.copy(spam)  
>>> id(cheese) # cheese es una lista diferente con una identidad diferente.  
44685832  
>>> cheese[1] = 42  
>>> spam  
['A', 'B', 'C', 'D']  
>>> cheese  
['A', 42, 'C', 'D']
```

Ahora las variables `spam` y `cheese` hacen referencia a listas separadas, por lo que solo se modifica la lista en `cheese` cuando se asigna 42 en el índice 1. Como se puede ver en [la Figura 4-7](#), los números de identificación de referencia ya no son los mismos para ambas variables porque las variables hacen referencia a listas independientes.

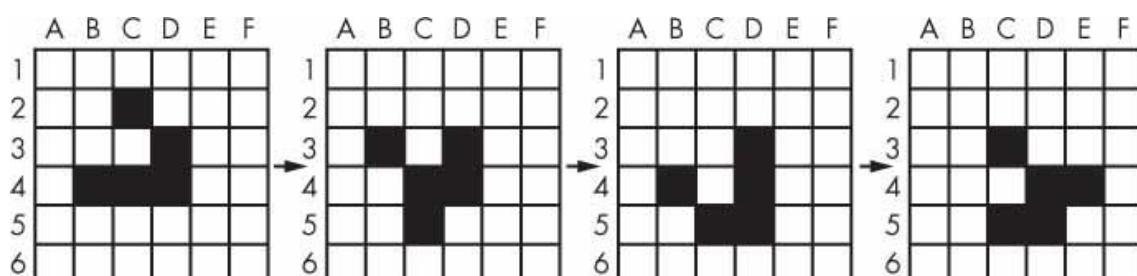


*Figura 4-7: `cheese = copy.copy(spam)` crea una segunda lista que puede modificarse independientemente de la primera.*

Si la lista que necesita copiar contiene listas, utilice la función `copy.deepcopy()` en lugar de `copy.copy()`. La función `deepcopy()` también copiará estas listas internas.

### Un programa breve: El juego de la vida de Conway

El Juego de la Vida de Conway es un ejemplo de *autómata celular*: un conjunto de reglas que rigen el comportamiento de un campo formado por celdas discretas. En la práctica, crea una animación muy agradable de ver. Puedes dibujar cada paso en un papel cuadrado, utilizando los cuadrados como celdas. Un cuadrado lleno estará “vivo” y un cuadrado vacío estará “muerto”. Si un cuadrado vivo tiene dos o tres vecinos vivos, continúa vivo en el siguiente paso. Si un cuadrado muerto tiene exactamente tres vecinos vivos, cobra vida en el siguiente paso. Todos los demás cuadrados mueren o permanecen muertos en el siguiente paso. Puedes ver un ejemplo de la progresión de pasos en [la Figura 4-8](#).



*Figura 4-8: Cuatro pasos en una simulación del Juego de la vida de Conway*

Aunque las reglas son sencillas, surgen muchos comportamientos sorprendentes. Los patrones del Juego de la vida de Conway pueden moverse, autorreplicarse o incluso imitar a las CPU. Pero en la base de todo este comportamiento complejo y avanzado se encuentra un programa bastante simple.

Podemos utilizar una lista de listas para representar el campo bidimensional. La lista interna representa cada columna de cuadrados y almacena una cadena hash '#' para los cuadrados vivos y una cadena de espacio ' ' para los cuadrados muertos. Escriba el siguiente código fuente en el editor de archivos y guarde el archivo como *conway.py* . No hay problema si no comprende bien cómo funciona todo el código; simplemente introdúzcalo y siga los comentarios y explicaciones que se proporcionan aquí lo más fielmente posible:

```
# El juego de la vida de Conway
import random, time, copy
WIDTH = 60
HEIGHT = 20

# Crea una lista de listas para las celdas:
nextCells = []
for x in range(WIDTH):
    column = [] # Crea una nueva columna.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Agrega una celda viva.
        else:
            column.append(' ') # Agrega una celda muerta.
    nextCells.append(column) # nextCells es una lista de listas de columnas.

while True: # Bucle del programa principal.
    print('\n\n\n\n\n') # Separa cada paso con nuevas líneas.
    currentCells = copy.deepcopy(nextCells)

    # Imprime currentCells en la pantalla:
    for y in range(HEIGHT):
        for x in range(WIDTH):
            print(currentCells[x][y], end='') # Imprime el # o espacio.
        print() # Imprime una nueva línea al final de la fila.

    # Calcula las celdas del siguiente paso basándose en las celdas del paso actual:
```

```

for x in range(WIDTH):
    for y in range(HEIGHT):
        # Obtiene las coordenadas vecinas:
        # ` % WIDTH` asegura que leftCoord siempre esté entre 0 y WIDTH - 1
        leftCoord = (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT

        # Cuenta el número de vecinos vivos:
        numNeighbors = 0
        if currentCells[leftCoord][aboveCoord] == '#':
            numNeighbors += 1 # El vecino superior izquierdo está vivo.
        if currentCells[x][aboveCoord] == '#':
            numNeighbors += 1 # El vecino superior está vivo.
        if currentCells[rightCoord][aboveCoord] == '#':
            numNeighbors += 1 # El vecino superior derecho está vivo.
        if currentCells[leftCoord][y] == '#':
            numNeighbors += 1 # El vecino izquierdo está vivo.
        if currentCells[rightCoord][y] == '#':
            numNeighbors += 1 # El vecino derecho está vivo.
        if currentCells[leftCoord][belowCoord] == '#':
            numNeighbors += 1 # El vecino inferior izquierdo está vivo.
        if currentCells[x][belowCoord] == '#':
            numNeighbors += 1 # El vecino inferior está vivo.
        if currentCells[rightCoord][belowCoord] == '#':
            numNeighbors += 1 # El vecino inferior derecho está vivo.

        # Establezca la celda según las reglas del Juego de la vida de Conway:
        if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
            # Las celdas vivas con 2 o 3 vecinos permanecen vivas:
            nextCells[x][y] = '#'
        elif currentCells[x][y] == ' ' and numNeighbors == 3:
            # Las celdas muertas con 3 vecinos se vuelven vivas:
            nextCells[x][y] = '#'
        else:
            # Todo lo demás muere o permanece muerto:
            nextCells[x][y] = ' '
    time.sleep(1) # Agregue una pausa de 1 segundo para reducir el parpadeo.

```

Veamos este código línea por línea, comenzando desde arriba.

```
# El juego de la vida de Conway
importar aleatorio, tiempo, copia
ANCHO = 60
ALTO = 20
```

Primero importamos módulos que contienen las funciones que necesitaremos, es decir, las funciones `random.randint()` , `time.sleep()` y `copy.deepcopy()` .

```
# Crea una lista de listas para las celdas:
nextCells = []
for x in range(WIDTH):
    column = [] # Crea una nueva columna.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Agrega una celda viva.
        else:
            column.append(' ') # Agrega una celda muerta.
    nextCells.append(column) # nextCells es una lista de listas de columnas.
```

El primer paso de nuestro autómata celular será completamente aleatorio. Necesitamos crear una estructura de datos de lista de listas para almacenar las cadenas '#' y ' ' que representan una célula viva o muerta, y su lugar en la lista de listas refleja su posición en la pantalla. Cada una de las listas internas representa una columna de células. La llamada `random.randint(0, 1)` da una probabilidad de 50/50 de que la célula comience viva o muerta.

Colocamos esta lista de listas en una variable llamada `nextCells` , porque el primer paso en nuestro bucle principal del programa será copiar `nextCells` en `currentCells` . Para nuestra estructura de datos de lista de listas, las coordenadas `x` comienzan en 0 a la izquierda y aumentan hacia la derecha, mientras que las coordenadas `y` comienzan en 0 en la parte superior y aumentan hacia abajo. Entonces, `nextCells[0][0]` representará la celda en la parte superior izquierda de la pantalla, mientras que `nextCells[1][0]` representa la celda a la derecha de esa celda y `nextCells[0][1]` representa la celda debajo de ella.

```
mientras True: # Bucle del programa principal.
    print("\n\n\n\n\n\n") # Separa cada paso con nuevas líneas.
    currentCells = copy.deepcopy(nextCells)
```

Cada iteración de nuestro bucle de programa principal será un paso único de nuestros autómatas celulares. En cada paso, copiaremos `nextCells` en `currentCells` , imprimiremos `currentCells` en la pantalla y luego usaremos las celdas en `currentCells` para calcular las celdas en `nextCells` .

```
# Imprime currentCells en la pantalla:
for y in range(HEIGHT):
    for x in range(WIDTH):
        print(currentCells[x][y], end=") # Imprime el # o espacio.
    print() # Imprime una nueva línea al final de la fila.
```

Estos bucles for anidados garantizan que imprimamos una fila completa de celdas en la pantalla, seguida de un carácter de nueva línea al final de la fila. Repetimos esto para cada fila en nextCells .

```
# Calcular las celdas del siguiente paso en función de las celdas del paso
actual:
for x in range(WIDTH):
    for y in range(HEIGHT):
        # Obtener las coordenadas vecinas:
        # ` % WIDTH ` garantiza que leftCoord siempre esté entre 0 y WIDTH - 1
        leftCoord = (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT
```

A continuación, necesitamos utilizar dos bucles for anidados para calcular cada celda para el siguiente paso. El estado activo o inactivo de la celda depende de las celdas vecinas, por lo que primero calcularemos el índice de las celdas a la izquierda, derecha, arriba y abajo de las coordenadas x e y actuales.

El operador % mod realiza un “envolvimiento”. El vecino izquierdo de una celda en la columna más a la izquierda 0 sería 0 - 1 o -1 . Para envolver esto hasta el índice de la columna más a la derecha, 59 , calculamos (0 - 1) % WIDTH . Como WIDTH es 60 , esta expresión se evalúa como 59 . Esta técnica de envolvimiento mod funciona también para los vecinos de la derecha, arriba y abajo.

```
# Cuenta el número de vecinos vivos:
numNeighbors = 0
if currentCells[leftCoord][aboveCoord] == '#':
    numNeighbors += 1 # El vecino superior izquierdo está vivo.
if currentCells[x][aboveCoord] == '#':
    numNeighbors += 1 # El vecino superior está vivo.
if currentCells[rightCoord][aboveCoord] == '#':
    numNeighbors += 1 # El vecino superior derecho está vivo.
if currentCells[leftCoord][y] == '#':
    numNeighbors += 1 # El vecino izquierdo está vivo.
if currentCells[rightCoord][y] == '#':
```



```

    numNeighbors += 1 # El vecino derecho está vivo.
if currentCells[leftCoord][belowCoord] == '#':
    numNeighbors += 1 # El vecino inferior izquierdo está vivo.
si currentCells[x][belowCoord] == '#':
    numNeighbors += 1 # El vecino de abajo está vivo.
si currentCells[rightCoord][belowCoord] == '#':
    numNeighbors += 1 # El vecino de abajo a la derecha está vivo.

```

Para decidir si la celda en `nextCells[x][y]` debe estar viva o muerta, necesitamos contar la cantidad de vecinas vivas que tiene `currentCells[x][y]`. Esta serie de instrucciones `if` verifica cada una de las ocho vecinas de esta celda y suma 1 a `numNeighbors` por cada una viva.

```

# Establezca la celda según las reglas del Juego de la vida de Conway:
if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
    # Las células vivas con 2 o 3 vecinas permanecen vivas:
    nextCells[x][y] = '#'
elif currentCells[x][y] == ' ' and numNeighbors == 3:
    # Las células muertas con 3 vecinas cobran vida:
    nextCells[x][y] = '#'
else:
    # Todo lo demás muere o permanece muerto:
    nextCells[x][y] = ' '
time.sleep(1) # Agregue una pausa de 1 segundo para reducir el parpadeo.

```

Ahora que conocemos la cantidad de vecinos vivos de la celda en `currentCells[x][y]`, podemos establecer `nextCells[x][y]` en '#' o ' '. Después de recorrer cada posible coordenada `x` y `y`, el programa hace una pausa de 1 segundo llamando a `time.sleep(1)`. Luego, la ejecución del programa vuelve al inicio del bucle principal del programa para continuar con el siguiente paso.

Se han descubierto varios patrones con nombres como “planeador”, “hélice” o “nave espacial pesada”. El patrón de planeador, que se muestra en [la Figura 4-8](#), da como resultado un patrón que se “mueve” en diagonal cada cuatro pasos. Puedes crear un solo planeador reemplazando esta línea en nuestro programa *conway.py*:

```

    si aleatorio.randint(0, 1) == 0:

```

con esta línea:

```

    si (x, y) en ((1, 0), (2, 1), (0, 2), (1, 2), (2, 2)):

```

Puedes encontrar más información sobre los fascinantes dispositivos creados con el Juego de la vida de Conway buscando en la web. Y puedes encontrar otros programas Python breves basados en texto como este en <https://github.com/asweigart/pythonstdiogames> .

## DICCIONARIOS Y ESTRUCTURACIÓN DE DATOS



En este capítulo, abordaré el tipo de datos de diccionario, que proporciona una forma flexible de acceder a los datos y organizarlos. Luego, al combinar los diccionarios con el conocimiento de las listas del capítulo anterior, aprenderá a crear una estructura de datos para modelar un tablero de tres en raya.

### El tipo de datos del diccionario

Al igual que una lista, un *diccionario* es una colección mutable de muchos valores. Pero a diferencia de los índices de listas, los índices de diccionarios pueden utilizar muchos tipos de datos diferentes, no solo números enteros. Los índices de diccionarios se denominan *claves* y una clave con su valor asociado se denomina *par clave-valor* .

En el código, un diccionario se escribe con llaves, {} . Ingrese lo siguiente en el shell interactivo:

```
>>> myCat = {'size': 'gordo', 'color': 'gris', 'disposición': 'ruidoso'}
```

Esto asigna un diccionario a la variable myCat . Las claves de este diccionario son 'size' , 'color' y 'disposition' . Los valores de estas claves son 'fat' , 'gray' y 'loud' , respectivamente. Puede acceder a estos valores a través de sus claves:

```
>>> myCat['size']
'fat'
>>> 'Mi gato tiene ' + myCat['color'] + ' pelaje.'
'Mi gato tiene el pelaje gris.'
```

Los diccionarios aún pueden usar valores enteros como claves, al igual que las listas usan valores enteros como índices, pero no tienen que comenzar en 0 y pueden ser cualquier número.

```
>>> spam = {12345: 'Combinación de equipaje', 42: 'La respuesta'}
```

### ***Diccionarios vs. listas***

A diferencia de las listas, los elementos de los diccionarios no están ordenados. El primer elemento de una lista denominada spam sería spam[0] . Pero no existe un elemento “primero” en un diccionario. Si bien el orden de los elementos es importante para determinar si dos listas son iguales, no importa en qué orden se escriben los pares clave-valor en un diccionario. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = ['gatos', 'perros', 'alces']
>>> tocino = ['perros', 'alces', 'gatos']
>>> spam == tocino
Falso
>>> huevos = {'nombre': 'Zophie', 'especie': 'gato', 'edad': '8'}
>>> jamón = {'especie': 'gato', 'edad': '8', 'nombre': 'Zophie'}
>>> huevos == jamón
Verdadero
```

Como los diccionarios no están ordenados, no se pueden dividir como listas.

Si intenta acceder a una clave que no existe en un diccionario, aparecerá un mensaje de error KeyError , muy parecido al mensaje de error IndexError de una lista que indica que está “fuera de rango” . Ingrese lo siguiente en el shell interactivo y observe el mensaje de error que aparece porque no hay una clave "color" :

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (última llamada reciente):
  Archivo "<pyshell#1>", línea 1, en <módulo>
    spam['color']
KeyError: 'color'
```

Aunque los diccionarios no están ordenados, el hecho de que puedas tener valores arbitrarios para las claves te permite organizar tus datos de maneras poderosas. Digamos que quieres que tu programa almacene datos sobre los cumpleaños de tus amigos. Puedes usar un diccionario con los nombres como claves y los cumpleaños como valores. Abre una nueva ventana del editor de archivos e ingresa el siguiente código. Guárdalo como *birthdays.py* .

❶ cumpleaños = {'Alice': '1 de abril', 'Bob': '12 de diciembre', 'Carol': '4 de marzo'}

```
while True:
```

```
    print('Ingresa un nombre: (deja en blanco para salir)')
```

```
    nombre = input()
```

```
    if nombre == '':
```

```
        break
```

```
❷ if nombre in cumpleaños:
```

```
    ❸ print(cumpleaños[nombre] + ' es el cumpleaños de ' + nombre)
```

```
else:
```

```
    print('No tengo información de cumpleaños para ' + nombre)
```

```
    print('¿Cuál es su cumpleaños?')
```

```
    cumpleaños = input()
```

```
❹ cumpleaños[nombre] = cumpleaños
```

```
    print('Base de datos de cumpleaños actualizada.')
```

Puedes ver la ejecución de este programa en <https://autbor.com/bdaydb> . Creas un diccionario inicial y lo guardas en birthdays ❶ . Puedes ver si el nombre ingresado existe como clave en el diccionario con la palabra clave in ❷ , tal como lo hiciste para las listas. Si el nombre está en el diccionario, accedes al valor asociado usando corchetes ❸ ; si no, puedes agregarlo usando la misma sintaxis de corchetes combinada con el operador de asignación ❹ .

Cuando ejecute este programa, se verá así:

Ingresa un nombre: (deja en blanco para salir)

**Alice**

El 1 de abril es el cumpleaños de Alice

Ingresa un nombre: (deja en blanco para salir)

**Eve**

No tengo información de cumpleaños para Eve

¿Cuál es su cumpleaños?

**5 de diciembre**

Base de datos de cumpleaños actualizada.

Ingresa un nombre: (deja en blanco para salir)

**Eve**

El 5 de diciembre es el cumpleaños de Eve

Ingresa un nombre: (deja en blanco para salir)

[Por supuesto, todos los datos que introduzca en este programa se olvidarán cuando éste finalice. En el capítulo 9](#) aprenderá a guardar datos en archivos del disco duro .

## DICCIONARIOS ORDENADOS EN PYTHON 3.7

Si bien aún no están ordenados y no tienen un par clave-valor “primero”, los diccionarios en Python 3.7 y versiones posteriores recordarán el orden de inserción de sus pares clave-valor si crea un valor de secuencia a partir de ellos. Por ejemplo, observe que el orden de los elementos en las listas creadas a partir de los diccionarios de huevos y jamón coincide con el orden en el que se ingresaron:

```
>>> huevos = {'nombre': 'Zophie', 'especie': 'gato', 'edad': '8'}
>>> lista(huevos)
['nombre', 'especie', 'edad']
>>> jamón = {'especie': 'gato', 'edad': '8', 'nombre': 'Zophie'}
>>> lista(jamón)
['especie', 'edad', 'nombre']
```

Los diccionarios siguen estando desordenados, ya que no se puede acceder a los elementos que contienen mediante índices enteros como `eggs[0]` o `ham[2]` . No deberías confiar en este comportamiento, ya que los diccionarios en versiones anteriores de Python no recuerdan el orden de inserción de los pares clave-valor. Por ejemplo, observa cómo la lista no coincide con el orden de inserción de los pares clave-valor del diccionario cuando ejecuto este código en Python 3.5:

```
>>> spam = {}
>>> spam['primera clave'] = 'valor'
>>> spam['segunda clave'] = 'valor'
>>> spam['tercera clave'] = 'valor'
>>> list(spam)
['primera clave', 'tercera clave', 'segunda clave']
```

### **Los métodos `keys()`, `values()` y `items()`**

Existen tres métodos de diccionario que devolverán valores similares a listas de las claves, valores o claves y valores del diccionario: `keys()` , `values()` y `items()` . Los valores devueltos por estos métodos no son listas verdaderas: no se pueden modificar y no tienen un método `append()` . Pero estos tipos de datos (`dict_keys` , `dict_values` y `dict_items` , respectivamente) *se pueden* usar en bucles `for` . Para ver cómo funcionan estos métodos, ingrese lo siguiente en el shell interactivo:

```
>>> spam = {'color': 'rojo', 'edad': 42}
>>> para v en spam.values():
...     print(v)
```

```
rojo
42
```

Aquí, un bucle for itera sobre cada uno de los valores del diccionario de spam . Un bucle for también puede iterar sobre las claves o sobre las claves y los valores:

```
>>> para k en spam.keys():
...     print(k)
```

```
color
edad
```

```
>>> para i en spam.items():
...     print(i)
```

```
('color', 'rojo')
('edad', 42)
```

Cuando se utilizan los métodos keys() , values() y items() , un bucle for puede iterar sobre las claves, los valores o los pares clave-valor de un diccionario, respectivamente. Observe que los valores del valor dict\_items devuelto por el método items() son tuplas de la clave y el valor.

Si desea obtener una lista verdadera de uno de estos métodos, pase su valor de retorno similar a una lista a la función list() . Ingrese lo siguiente en el shell interactivo:

```
>>> spam = {'color': 'rojo', 'edad': 42}
>>> spam.keys()
dict_keys(['color', 'edad'])
>>> list(spam.keys())
['color', 'edad']
```

La línea list(spam.keys()) toma el valor dict\_keys devuelto por keys() y lo pasa a list() , que luego devuelve un valor de lista de ['color', 'age'] .

También puede utilizar el truco de asignación múltiple en un bucle for para asignar la clave y el valor a variables independientes. Introduzca lo siguiente en el shell interactivo:

```
>>> spam = {'color': 'rojo', 'edad': 42}
>>> for k, v in spam.items():
```

```
... print('Clave: ' + k + ' Valor: ' + str(v))
```

Clave: edad Valor: 42

Clave: color Valor: rojo

### ***Cómo comprobar si una clave o un valor existe en un diccionario***

Recuerde que en el capítulo anterior se dijo que los operadores `in` y `not in` permiten comprobar si existe un valor en una lista. También puede utilizar estos operadores para comprobar si existe una clave o un valor determinados en un diccionario. Introduzca lo siguiente en el shell interactivo:

```
>>> spam = {'nombre': 'Zophie', 'edad': 7}
```

```
>>> 'nombre' en spam.keys()
```

Verdadero

```
>>> 'Zophie' en spam.values()
```

Verdadero

```
>>> 'color' en spam.keys()
```

Falso

```
>>> 'color' no en spam.keys()
```

Verdadero

```
>>> 'color' en spam
```

Falso

En el ejemplo anterior, observe que "color" en spam es esencialmente una versión abreviada de escribir "color" en `spam.keys()` . Esto siempre es así: si alguna vez desea verificar si un valor es (o no es) una clave en el diccionario, simplemente puede usar la palabra clave `in` (o `not in` ) con el valor del diccionario en sí.

### ***El método get()***

Es tedioso comprobar si una clave existe en un diccionario antes de acceder al valor de esa clave. Afortunadamente, los diccionarios tienen un método `get()` que toma dos argumentos: la clave del valor que se desea recuperar y un valor de reserva que se devolverá si esa clave no existe.

Introduzca lo siguiente en el shell interactivo:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
```

```
>>> 'Traigo ' + str(picnicItems.get('cups', 0)) + ' cups.'
```

'Traigo 2 tazas.'

```
>>> 'Traigo ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
```

'Traigo 0 huevos.'

Como no hay ninguna clave "eggs" en el diccionario picnicItems , el método get() devuelve el valor predeterminado 0. Sin utilizar get() , el código habría generado un mensaje de error, como el del siguiente ejemplo:

```
>>> picnicItems = {'manzanas': 5, 'tazas': 2}
>>> 'Traigo ' + str(picnicItems['huevos']) + ' huevos.'
Traceback (última llamada más reciente):
  Archivo "<pyshell#34>", línea 1, en <módulo>
    'Traigo ' + str(picnicItems['huevos']) + ' huevos.'
KeyError: 'huevos'
```

### ***El método setdefault()***

A menudo, tendrás que establecer un valor en un diccionario para una clave determinada solo si esa clave aún no tiene un valor. El código se parece a esto:

```
spam = {'nombre': 'Pooka', 'edad': 5}
si 'color' no está en spam:
    spam['color'] = 'negro'
```

El método setdefault() ofrece una forma de hacer esto en una sola línea de código. El primer argumento que se pasa al método es la clave que se debe verificar y el segundo argumento es el valor que se debe establecer en esa clave si la clave no existe. Si la clave existe, el método setdefault() devuelve el valor de la clave. Ingrese lo siguiente en el shell interactivo:

```
>>> spam = {'nombre': 'Pooka', 'edad': 5}
>>> spam.setdefault('color', 'negro')
'negro'
>>> spam
{'color': 'negro', 'edad': 5, 'nombre': 'Pooka'}
>>> spam.setdefault('color', 'blanco')
'negro'
>>> spam
{'color': 'negro', 'edad': 5, 'nombre': 'Pooka'}
```

La primera vez que se llama a setdefault() , el diccionario en spam cambia a {'color': 'black', 'age': 5, 'name': 'Pooka'} . El método devuelve el valor 'black' porque este es ahora el valor establecido para la clave 'color' . Cuando se llama a spam.setdefault('color', 'white') a continuación, el valor de esa clave *no* cambia a 'white' , porque spam ya tiene una clave llamada 'color' .

El método setdefault() es un buen atajo para garantizar que exista una clave. Aquí hay un programa breve que cuenta la cantidad de veces que aparece cada letra en



una cadena. Abra la ventana del editor de archivos e ingrese el siguiente código, guardándolo como *characterCount.py* :

```
mensaje = 'Era un día frío y luminoso de abril, y los relojes estaban dando  
las trece.'  
count = {}
```

```
for character in message:
```

```
    ❶ count.setdefault(character, 0)
```

```
    ❷ count[character] = count[character] + 1
```

```
print(count)
```

Puede ver la ejecución de este programa en <https://author.com/setdefault> . El programa recorre cada carácter de la cadena de la variable *message* y cuenta la frecuencia con la que aparece cada carácter. La llamada al método *setdefault()* ❶ garantiza que la clave esté en el diccionario *count* (con un valor predeterminado de 0 ) para que el programa no genere un error *KeyError* cuando se ejecute *count[character] = count[character] + 1* ❷ . Cuando ejecute este programa, la salida se verá así:

```
{ ' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2,  
'i': 6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1 }
```

En el resultado, puede ver que la letra *c* minúscula aparece 3 veces, el carácter de espacio aparece 13 veces y la letra *A* mayúscula aparece 1 vez. Este programa funcionará sin importar qué cadena esté dentro de la variable de mensaje , ¡incluso si la cadena tiene millones de caracteres!

### **Impresión bonita**

Si importa el módulo *pprint* en sus programas, tendrá acceso a las funciones *pprint()* y *pformat()* que “imprimirán de forma ordenada” los valores de un diccionario. Esto resulta útil cuando desea una visualización más clara de los elementos de un diccionario que la que ofrece *print()* . Modifique el programa *characterCount.py* anterior y guárdelo como *prettyCharacterCount.py* .

#### **import pprint**

```
message = 'Era un día frío y luminoso de abril y los relojes estaban dando  
las trece.'  
count = {}
```

```
for character in message:
```

```
    count.setdefault(character, 0)
```

```
count[character] = count[character] + 1
```

**pprint.pprint** (count)

Puedes ver la ejecución de este programa en <https://autbor.com/pprint/> . Esta vez, cuando se ejecuta el programa, el resultado se ve mucho más limpio, con las claves ordenadas.

```
{': 13,  
'; 1,  
': 1,  
'A': 1,  
'I': 1,  
--snip--  
't': 6,  
'w': 2,  
'y': 1}
```

La función pprint.pprint() es especialmente útil cuando el propio diccionario contiene listas o diccionarios anidados.

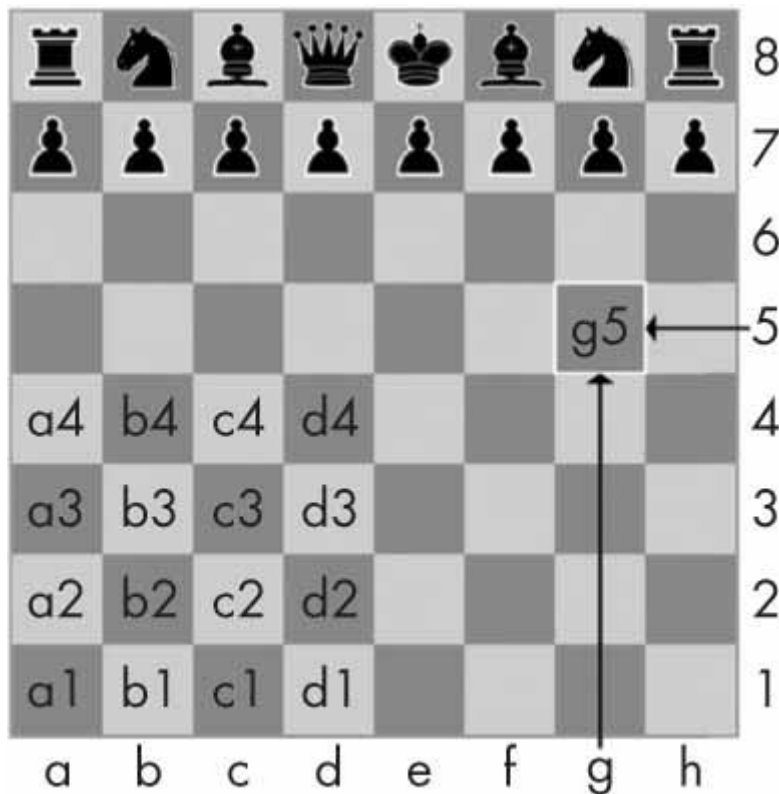
Si desea obtener el texto embellecido como un valor de cadena en lugar de mostrarlo en la pantalla, llame a pprint.pformat() . Estas dos líneas son equivalentes entre sí:

```
pprint.pprint(algúnValorDelDiccionario)  
print(pprint.pformat(algúnValorDelDiccionario))
```

### **Uso de estructuras de datos para modelar cosas del mundo real**

Incluso antes de Internet, era posible jugar una partida de ajedrez con alguien que estuviera al otro lado del mundo. Cada jugador instalaba un tablero de ajedrez en su casa y luego se turnaban para enviarse postales describiendo cada movimiento. Para ello, los jugadores necesitaban una forma de describir de forma inequívoca el estado del tablero y sus movimientos.

En la *notación algebraica del ajedrez* , los espacios en el tablero de ajedrez se identifican mediante un número y una letra, como en [la Figura 5-1](#) .



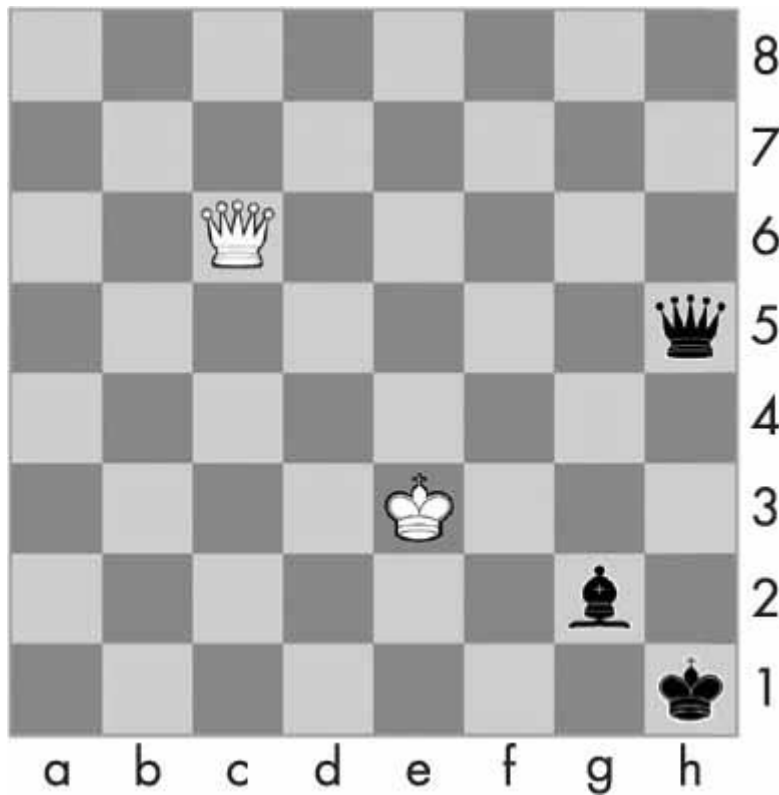
*Figura 5-1: Las coordenadas de un tablero de ajedrez en notación algebraica de ajedrez*

Las piezas de ajedrez se identifican con letras: *K* para el rey, *Q* para la dama, *R* para la torre, *B* para el alfil y *N* para el caballo. Para describir un movimiento se utilizan la letra de la pieza y las coordenadas de su destino. Un par de estos movimientos describe lo que sucede en un solo turno (con las blancas avanzando primero); por ejemplo, la notación 2. *Cf3 Cc6* indica que las blancas movieron un caballo a f3 y las negras movieron un caballo a c6 en el segundo turno de la partida.

La notación algebraica implica algo más que esto, pero el punto es que puedes describir de manera inequívoca una partida de ajedrez sin necesidad de estar frente a un tablero de ajedrez. ¡Tu oponente puede incluso estar al otro lado del mundo! De hecho, ni siquiera necesitas un tablero de ajedrez físico si tienes buena memoria: puedes simplemente leer las jugadas de ajedrez enviadas por correo y actualizar los tableros que tienes en tu imaginación.

Las computadoras tienen buena memoria. Un programa en una computadora moderna puede almacenar fácilmente miles de millones de cadenas como '2. *Cf3 Cc6*'. Así es como las computadoras pueden jugar al ajedrez sin tener un tablero físico. Modelan datos para representar un tablero de ajedrez y usted puede escribir código para trabajar con este modelo.

Aquí es donde pueden entrar en juego las listas y los diccionarios. Por ejemplo, el diccionario {'1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'} podría representar el tablero de ajedrez de [la Figura 5-2](#).

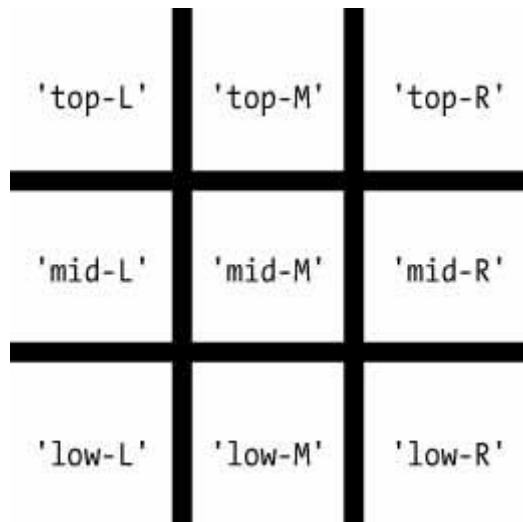


*Figura 5-2: Un tablero de ajedrez modelado por el diccionario '1h': 'bking', '6c': 'wqueen', '2g': 'bbishop', '5h': 'bqueen', '3e': 'wking'}*

Pero para otro ejemplo, utilizaremos un juego un poco más simple que el ajedrez: el tres en raya.

### **Un tablero de tres en raya**

Un tablero de tres en raya parece un gran símbolo numeral (#) con nueve espacios que pueden contener una X, una O o un espacio en blanco. Para representar el tablero con un diccionario, puede asignar a cada espacio una clave con un valor de cadena, como se muestra en [la Figura 5-3](#).



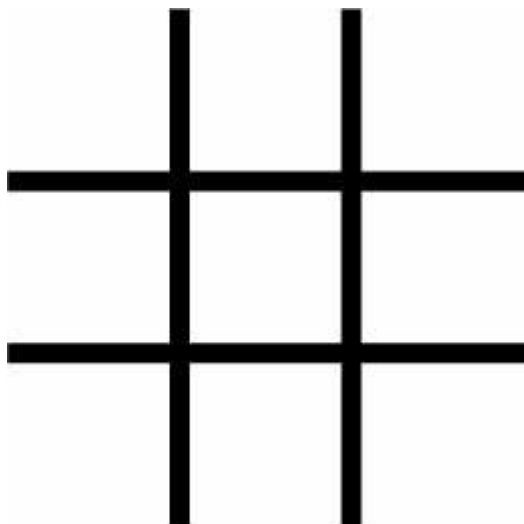
*Figura 5-3: Las ranuras de un tablero de tres en raya con sus teclas correspondientes*

Puedes usar valores de cadena para representar lo que hay en cada ranura del tablero: 'X', 'O' o ' ' (un espacio). Por lo tanto, necesitarás almacenar nueve cadenas. Puedes usar un diccionario de valores para esto. El valor de cadena con la clave 'top-R' puede representar la esquina superior derecha, el valor de cadena con la clave 'low-L' puede representar la esquina inferior izquierda, el valor de cadena con la clave 'mid-M' puede representar el medio, y así sucesivamente.

Este diccionario es una estructura de datos que representa un tablero de tres en raya. Guarde este tablero como diccionario en una variable llamada `theBoard` . Abra una nueva ventana del editor de archivos e ingrese el siguiente código fuente, guardándolo como `ticTacToe.py` :

```
elTablero = {'arriba-I': ' ', 'arriba-M': ' ', 'arriba-D': ' ',  
             'medio-I': ' ', 'medio-M': ' ', 'medio-D': ' ',  
             'bajo-I': ' ', 'bajo-M': ' ', 'bajo-D': ' '}
```

La estructura de datos almacenada en la variable `theBoard` representa el tablero de tres en raya de [la Figura 5-4](#) .

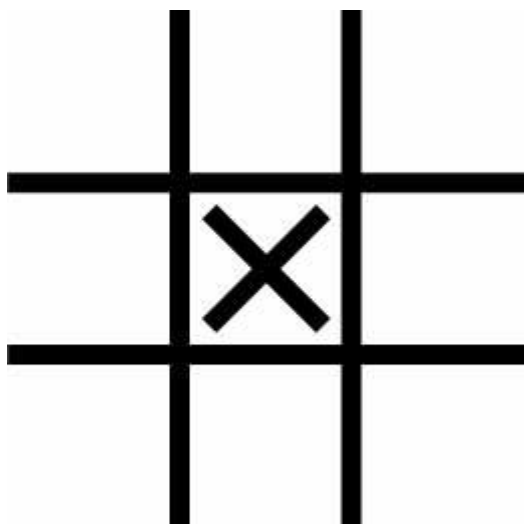


*Figura 5-4: Un tablero de tres en raya vacío*

Dado que el valor de cada clave en el tablero es una cadena de un solo espacio, este diccionario representa un tablero completamente despejado. Si el jugador X fue el primero y eligió el espacio del medio, podría representar ese tablero con este diccionario:

```
elTablero = {'arriba-L': '', 'arriba-M': '', 'arriba-R': '',
             'medio-L': '', 'medio-M': 'X', 'medio-R': '',
             'bajo-L': '', 'bajo-M': '', 'bajo-R': ''}
```

La estructura de datos en theBoard ahora representa el tablero de tres en raya de [la Figura 5-5](#).

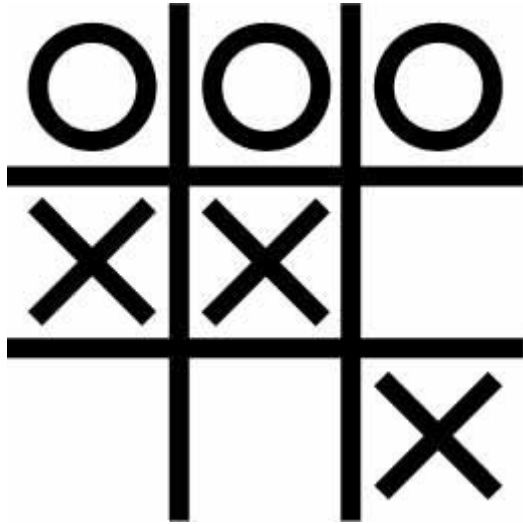


*Figura 5-5: El primer movimiento*

Un tablero en el que el jugador O ha ganado colocando O en la parte superior podría verse así:

```
elTablero = {'arriba-I': 'O', 'arriba-M': 'O', 'arriba-D': 'O',
             'medio-I': 'X', 'medio-M': 'X', 'medio-D': '',
             'bajo-I': '', 'bajo-M': '', 'bajo-D': 'X'}
```

La estructura de datos en theBoard ahora representa el tablero de tres en raya de [la Figura 5-6](#).



*Figura 5-6: El jugador O gana.*

Por supuesto, el jugador solo ve lo que se imprime en la pantalla, no el contenido de las variables. Vamos a crear una función para imprimir el diccionario del tablero en la pantalla. Realicemos la siguiente adición a *ticTacToe.py* (el código nuevo está en negrita):

```
elTablero = {'arriba-I': '', 'arriba-M': '', 'arriba-D': '',
             'medio-I': '', 'medio-M': '', 'medio-D': '',
             'bajo-I': '', 'bajo-M': '', 'bajo-D': '' }

def imprimirTablero(tablero):
    print(tablero['arriba-I'] + '|' + tablero['arriba-M'] + '|' + tablero['arriba-D'])
    print('-+-+-')
    print(tablero['medio-I'] + '|' + tablero['medio-M'] + '|' + tablero['medio-D'])
    print('-+-+-')
    print(tablero['bajo-I'] + '|' + tablero['bajo-M'] + '|' + tablero['bajo-D'])
imprimirTablero(elTablero)
```

Puedes ver la ejecución de este programa en <https://autbor.com/tictactoe1/>. Cuando ejecutes este programa, printBoard() imprimirá un tablero de tres en raya en blanco.

```
||
-+-+-
||
```

```
--+-+--  
||
```

La función `printBoard()` puede manejar cualquier estructura de datos de tres en raya que le pases. Prueba a cambiar el código por el siguiente:

```
elTablero = {'arriba-I': 'O', 'arriba-M': 'O', 'arriba-D': 'O', 'medio-I': 'X', 'medio-M':  
'X', 'medio-D': ' ', 'bajo-I': ' ', 'bajo-M': ' ', 'bajo-D': 'X'}
```

```
def imprimirTablero(tablero):  
    print(tablero['arriba-I'] + '|' + tablero['arriba-M'] + '|' + tablero['arriba-D'])  
    print('--+-+--')  
    print(tablero['medio-I'] + '|' + tablero['medio-M'] + '|' + tablero['medio-D'])  
    print('--+-+--')  
    print(tablero['bajo-I'] + '|' + tablero['bajo-M'] + '|' + tablero['bajo-D'])  
imprimirTablero(elTablero)
```

Puede ver la ejecución de este programa en <https://autbor.com/tictactoe2/> .  
Ahora, cuando ejecute este programa, la nueva placa se imprimirá en la pantalla.

```
O|O|O  
--+-+--  
X|X|  
--+-+--  
| |X
```

Como creaste una estructura de datos para representar un tablero de tres en raya y escribiste código en `printBoard()` para interpretar esa estructura de datos, ahora tienes un programa que “modela” el tablero de tres en raya. Podrías haber organizado tu estructura de datos de otra manera (por ejemplo, usando claves como 'TOP-LEFT' en lugar de 'top-L' ), pero mientras el código funcione con tus estructuras de datos, tendrás un programa que funcionará correctamente.

Por ejemplo, la función `printBoard()` espera que la estructura de datos del juego tres en raya sea un diccionario con claves para las nueve ranuras. Si al diccionario que pasaste le faltara, por ejemplo, la tecla 'L central' , tu programa ya no funcionaría.

```
O|O|O  
--+-+--
```

Traceback (última llamada reciente):

```
Archivo "ticTacToe.py", línea 10, en <módulo>  
    printBoard(theBoard)  
Archivo "ticTacToe.py", línea 6, en printBoard
```



```
print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
KeyError: 'mid-L'
```

Ahora, agreguemos el código que permite a los jugadores ingresar sus movimientos. Modifique el programa *ticTacToe.py* para que se vea así:

```
elTablero = {'arriba-I': ' ', 'arriba-M': ' ', 'arriba-D': ' ', 'medio-I': ' ', 'medio-M': ' ', 'medio-D': ' ', 'bajo-I': ' ', 'bajo-M': ' ', 'bajo-D': ' '}
```

```
def imprimirTablero(tablero):
    print(tablero['arriba-I'] + '|' + tablero['arriba-M'] + '|' + tablero['arriba-D'])
    print('-+-+-')
    print(tablero['centro-I'] + '|' + tablero['centro-M'] + '|' + tablero['centro-R'])
    print('-+-+-')
    print(tablero['bajo-I'] + '|' + tablero['bajo-M'] + '|' + tablero['bajo-R'])
turno = 'X'
for i in range(9):
    ❶ printTablero(elTablero)
    print('Giro para ' + turno + '. ¿Movimiento en qué espacio?')
    ❷ movimiento = input()
    ❸ elTablero[movimiento] = turno
    ❹ if turno == 'X':
        turno = 'O'
    else:
        turno = 'X'
    printTablero(elTablero)
```

Puedes ver la ejecución de este programa en <https://autbor.com/tictactoe3/> . El nuevo código imprime el tablero al comienzo de cada nuevo turno ❶ , obtiene el movimiento del jugador activo ❷ , actualiza el tablero de juego en consecuencia ❸ y luego cambia el jugador activo ❹ antes de pasar al siguiente turno.

Cuando ejecute este programa, se verá así:

```
||
-+-+
||
-+-+
||
Turno para X. ¿En qué espacio te mueves?
M media
||
```

--+-+--

|X|

--+-+--

||

--snip--

O|O|X

--+-+--

X|X|O

--+-+--

O| |X

Turno para X. ¿En qué espacio te mueves?

**M baja**

O|O|X

--+-+--

X|X|O

--+-+--

O|X|X

Este no es un juego de tres en raya completo (por ejemplo, nunca comprueba si un jugador ha ganado), pero es suficiente para ver cómo se pueden usar las estructuras de datos en los programas.

## NOTA

*Si tienes curiosidad, el código fuente de un programa completo de tres en raya se describe en los recursos disponibles en <https://nostarch.com/automatestuff2/>.*

## Diccionarios y listas anidadas

Modelar un tablero de tres en raya fue bastante simple: el tablero solo necesitaba un único valor de diccionario con nueve pares clave-valor. A medida que modele cosas más complicadas, puede descubrir que necesita diccionarios y listas que contengan otros diccionarios y listas. Las listas son útiles para contener una serie ordenada de valores, y los diccionarios son útiles para asociar claves con valores. Por ejemplo, aquí hay un programa que usa un diccionario que contiene otros diccionarios de los artículos que los invitados traen a un picnic. La función `totalBrought()` puede leer esta estructura de datos y calcular la cantidad total de un artículo que traen todos los invitados.

```
allGuests = {'Alice': {'manzanas': 5, 'pretzels': 12},
             'Bob': {'sándwiches de jamón': 3, 'manzanas': 2},
             'Carol': {'tazas': 3, 'tartas de manzana': 1}}
```

```
def totalBrought(guests, item):
    numBrought = 0
    ❶ for k, v in guests.items():
        ❷ numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Número de cosas que se traen:')
print(' - Manzanas ' + str(totalBrought(allGuests, 'manzanas')))
print(' - Tazas ' + str(totalBrought(allGuests, 'tazas')))
print(' - Tortas ' + str(totalBrought(allGuests, 'pasteles')))
print(' - Sándwiches de jamón ' + str(totalTraídos(todosLosInvitados, 'sándwiches de jamón')))
print(' - Tartas de manzana ' + str(totalTraídos(todosLosInvitados, 'tartas de manzana')))
```

Puedes ver la ejecución de este programa en <https://author.com/guestpicnic/> . Dentro de la función totalBrought() , el bucle for itera sobre los pares clave-valor en guests ❶ . Dentro del bucle, la cadena del nombre del invitado se asigna a k , y el diccionario de elementos de picnic que traerán se asigna a v . Si el parámetro item existe como una clave en este diccionario, su valor (la cantidad) se agrega a numBrought ❷ . Si no existe como una clave, el método get() devuelve 0 para agregarlo a numBrought .

La salida de este programa se ve así:

Número de cosas que se traen:

- Manzanas 7
- Tazas 3
- Pasteles 0
- Sándwiches de jamón 3
- Tartas de manzana 1

Puede parecer algo tan sencillo de modelar que no necesitaría molestarse en escribir un programa para hacerlo. Pero tenga en cuenta que esta misma función totalBrought() podría manejar fácilmente un diccionario que contenga miles de invitados, cada uno de los cuales trae *miles* de elementos diferentes para un picnic. ¡Tener esta información en una estructura de datos junto con la función totalBrought() le ahorraría mucho tiempo!

Puedes modelar cosas con estructuras de datos de la forma que quieras, siempre y cuando el resto del código de tu programa pueda funcionar correctamente con el modelo de datos. Cuando comiences a programar, no te preocupes tanto por la

forma "correcta" de modelar los datos. A medida que adquieras más experiencia, es posible que se te ocurran modelos más eficientes, pero lo importante es que el modelo de datos funcione para las necesidades de tu programa.