# 4 Trabajar con listas



En el Capítulo 3 aprendiste a hacer una lista simple, y aprendiste a trabajar con los elementos individuales de una lista. En este capítulo aprenderás a recorrer en bucle una lista entera utilizando sólo unas pocas líneas de código, independientemente de lo larga que sea la lista. *Looping* te permite realizar la misma acción, o conjunto de acciones, con cada elemento de una lista. Como resultado, podrás trabajar eficazmente con listas de cualquier longitud, incluidas las que tienen miles o incluso millones de elementos.

## Recorrer en bucle toda una lista

A menudo querrás recorrer todas las entradas de una lista, realizando la misma tarea con cada elemento. Por ejemplo, en un juego puedes querer mover todos los elementos de la pantalla la misma cantidad. En una lista de números, quizá quieras realizar la misma operación estadística con cada elemento. O quizá quieras mostrar cada titular de una lista de artículos de un sitio web. Cuando quieras realizar la misma acción con cada elemento de una lista, puedes utilizar el bucle for de Python.

Supongamos que tenemos una lista de nombres de magos y queremos imprimir cada nombre de la lista. Podríamos hacerlo recuperando cada nombre de la lista individualmente, pero este enfoque podría causar varios problemas. Por un lado, sería repetitivo hacerlo con una lista larga de nombres. Además, tendríamos que cambiar nuestro código cada vez que cambiara la longitud de la lista. Utilizar un bucle for evita estos dos problemas al dejar que Python gestione estas cuestiones internamente.

Utilicemos un bucle for para imprimir cada nombre de una lista de magos:

**magicians.py**

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
 print(magician)

Comenzamos definiendo una lista, tal y como hicimos en el Capítulo 3. Luego definimos un bucle for. Esta línea le dice a Python que saque un nombre de la lista magicians, y lo asocie a la variable magician. A continuación, le decimos a Python que imprima el nombre que se acaba de asignar a magician. A continuación, Python repite estas dos últimas líneas, una vez por cada nombre de la lista. Puede ayudar leer este código como "Para cada mago de la lista de magos, imprime el nombre del mago". La salida es una simple impresión de cada nombre de la lista:

alice  
david  
carolina

### Un vistazo más de cerca a los bucles

Los bucles son importantes porque son una de las formas más comunes que tiene un ordenador de automatizar tareas repetitivas. Por ejemplo, en un bucle simple como el que utilizamos en *magicians.py*, Python lee inicialmente la primera línea del bucle:

for magician in magicians:

Esta línea le dice a Python que recupere el primer valor de la lista magicians y lo asocie a la variable magician. Este primer valor es 'alice'. A continuación, Python lee la siguiente línea:

print(magician)

Python imprime el valor actual de magician, que sigue siendo 'alice'. Como la lista contiene más valores, Python vuelve a la primera línea del bucle:

for magician in magicians:

Python recupera el siguiente nombre de la lista, 'david', y asocia ese valor a la variable magician. A continuación, Python ejecuta la línea:

print(magician)

Python vuelve a imprimir el valor actual de magician, que ahora es 'david'. Python repite todo el bucle una vez más con el último valor de la lista, 'carolina'. Como no hay más valores en la lista, Python pasa a la siguiente línea del programa. En este caso no viene nada después del bucle for, por lo que el programa termina.

Cuando utilices bucles por primera vez, ten en cuenta que el conjunto de pasos se repite una vez para cada elemento de la lista, sin importar cuántos elementos haya en ella. Si tienes un millón de elementos en tu lista, Python repite estos pasos un millón de veces, y normalmente muy rápido.

Ten en cuenta también al escribir tus propios bucles for que puedes elegir el nombre que quieras para la variable temporal que se asociará a cada valor de la lista. Sin embargo, es útil elegir un nombre significativo que represente un único elemento de la lista. Por ejemplo, ésta es una buena forma de iniciar un bucle for para una lista de gatos, una lista de perros y una lista general de elementos:

for cat in cats:  
for dog in dogs:  
for item in list\_of\_items:

Estas convenciones de nomenclatura pueden ayudarte a seguir la acción que se realiza en cada elemento dentro de un bucle for. Utilizar nombres en singular y plural puede ayudarte a identificar si una sección de código está trabajando con un único elemento de la lista o con toda la lista.

### Hacer más trabajo dentro de un bucle for

Puedes hacer prácticamente cualquier cosa con cada elemento de un bucle for. Basándonos en el ejemplo anterior, imprimamos un mensaje a cada mago, diciéndole que ha hecho un gran truco:

**magicians.py**

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
 print(f"{magician.title()}, that was a great trick!")

La única diferencia en este código es que componemos un mensaje para cada mago, empezando por su nombre. La primera vez que se pasa por el bucle, el valor de magician es 'alice', por lo que Python inicia el primer mensaje con el nombre 'Alice'. La segunda vez, el mensaje empezará por 'David', y la tercera, por 'Carolina'.

La salida muestra un mensaje personalizado para cada mago de la lista:

Alice, that was a great trick!  
David, that was a great trick!  
Carolina, that was a great trick!

También puedes escribir tantas líneas de código como quieras en el bucle for. Cada línea sangrada que sigue a la línea for magician in magicians se considera *inside the loop*, y cada línea sangrada se ejecuta una vez por cada valor de la lista. Por tanto, puedes hacer todo el trabajo que quieras con cada valor de la lista.

Añadamos una segunda línea a nuestro mensaje, diciéndole a cada mago que esperamos su próximo truco:

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
 print(f"{magician.title()}, that was a great trick!")  
 print(f"I can't wait to see your next trick, {magician.title()}.\n")

Como hemos sangrado ambas llamadas a print(), cada línea se ejecutará una vez por cada mago de la lista. La nueva línea ("\n") en la segunda llamada a print() inserta una línea en blanco después de cada pasada por el bucle. Esto crea un conjunto de mensajes agrupados ordenadamente para cada persona de la lista:

Alice, that was a great trick!  
I can't wait to see your next trick, Alice.  
  
David, that was a great trick!  
I can't wait to see your next trick, David.  
  
Carolina, that was a great trick!  
I can't wait to see your next trick, Carolina.

Puedes utilizar tantas líneas como quieras en tus bucles for. En la práctica, a menudo te resultará útil realizar varias operaciones diferentes con cada elemento de una lista cuando utilices un bucle for.

### Hacer algo después de un bucle for

¿Qué ocurre una vez que un bucle for ha terminado de ejecutarse? Normalmente, querrás resumir un bloque de salida o pasar a otro trabajo que tu programa deba realizar.

Cualquier línea de código después del bucle for que no esté indentada se ejecuta una vez sin repetirse. Escribamos un mensaje de agradecimiento al grupo de magos en su conjunto, dándoles las gracias por haber ofrecido un espectáculo excelente. Para mostrar este mensaje de grupo después de que se hayan impreso todos los mensajes individuales, colocamos el mensaje de agradecimiento después del bucle for, sin sangría:

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
 print(f"{magician.title()}, that was a great trick!")  
 print(f"I can't wait to see your next trick, {magician.title()}.\n")  
  
print("Thank you, everyone. That was a great magic show!")

Las dos primeras llamadas a print() se repiten una vez para cada mago de la lista, como has visto antes. Sin embargo, como la última línea no tiene sangría, sólo se imprime una vez:

Alice, that was a great trick!  
I can't wait to see your next trick, Alice.  
  
David, that was a great trick!  
I can't wait to see your next trick, David.  
  
Carolina, that was a great trick!  
I can't wait to see your next trick, Carolina.  
  
Thank you, everyone. That was a great magic show!

Cuando proceses datos utilizando un bucle for, verás que ésta es una buena forma de resumir una operación realizada en un conjunto de datos completo. Por ejemplo, puedes utilizar un bucle for para inicializar un juego recorriendo una lista de personajes y mostrando cada uno de ellos en la pantalla. Después de este bucle, puedes escribir código adicional que muestre un botón *Play Now* una vez que todos los personajes se hayan dibujado en la pantalla.

## Evitar errores de sangría

Python utiliza la sangría para determinar cómo se relaciona una línea, o un grupo de líneas, con el resto del programa. En los ejemplos anteriores, las líneas que imprimían mensajes a los magos individuales formaban parte del bucle for porque estaban indentadas. El uso de la sangría en Python hace que el código sea muy fácil de leer. Básicamente, utiliza los espacios en blanco para obligarte a escribir código con un formato ordenado y una estructura visual clara. En los programas Python más largos, verás bloques de código sangrados en diferentes niveles. Estos niveles de sangría te ayudan a tener una idea general de la organización del programa.

Cuando empieces a escribir código que dependa de una indentación adecuada, tendrás que estar atento a algunos errores comunes *indentation errors*. Por ejemplo, la gente a veces indenta líneas de código que no necesitan ser indentadas u olvida indentar líneas que necesitan ser indentadas. Ver ahora ejemplos de estos errores te ayudará a evitarlos en el futuro y a corregirlos cuando aparezcan en tus propios programas.

Examinemos algunos de los errores de sangría más comunes.

### Olvidar la sangría

Sangrar siempre la línea que sigue a la sentencia for en un bucle. Si te olvidas, Python te lo recordará:

**magicians.py**

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
❶ print(magician)

La llamada a print() ❶ debería estar sangrada, pero no lo está. Cuando Python espera un bloque indentado y no lo encuentra, te indica con qué línea ha tenido un problema:

File "magicians.py", line 3  
 print(magician)  
 ^  
IndentationError: expected an indented block after 'for' statement on line 2

Normalmente puedes resolver este tipo de error de indentación indentando la línea o líneas inmediatamente posteriores a la sentencia for.

### Olvidar sangrar las líneas adicionales

A veces tu bucle se ejecutará sin errores, pero no producirá el resultado esperado. Esto puede ocurrir cuando intentas realizar varias tareas en un bucle y te olvidas de sangrar algunas de sus líneas.

Por ejemplo, esto es lo que ocurre cuando olvidamos sangrar la segunda línea del bucle que dice a cada mago que esperamos su próximo truco:

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
 print(f"{magician.title()}, that was a great trick!")  
❶ print(f"I can't wait to see your next trick, {magician.title()}.\n")

Se supone que la segunda llamada a print() ❶ debe tener sangría, pero como Python encuentra al menos una línea con sangría después de la sentencia for, no informa de un error. Como resultado, la primera llamada a print() se ejecuta una vez por cada nombre de la lista porque está indentada. La segunda llamada a print() no está indentada, por lo que sólo se ejecuta una vez después de que el bucle haya terminado de ejecutarse. Como el valor final asociado a magician es 'carolina', ella es la única que recibe el mensaje "esperando el próximo truco":

Alice, that was a great trick!  
David, that was a great trick!  
Carolina, that was a great trick!  
I can't wait to see your next trick, Carolina.

Esto es un *logical error*. La sintaxis es código Python válido, pero el código no produce el resultado deseado porque se produce un problema en su lógica. Si esperas que una determinada acción se repita una vez para cada elemento de una lista y sólo se ejecuta una vez, determina si necesitas simplemente sangrar una línea o un grupo de líneas.

### Sangrar innecesariamente

Si sangras accidentalmente una línea que no necesita ser sangrada, Python te informa de la sangría inesperada:

**hello\_world.py**

message = "Hello Python world!"  
 print(message)

No necesitamos sangrar la llamada a print(), porque no forma parte de un bucle; por tanto, Python informa de ese error:

File "hello\_world.py", line 2  
 print(message)  
 ^  
IndentationError: unexpected indent

Puedes evitar los errores de sangrado inesperado sangrando sólo cuando tengas una razón específica para hacerlo. En los programas que estás escribiendo en este momento, las únicas líneas que debes sangrar son las acciones que quieres repetir para cada elemento de un bucle for.

### Sangrar innecesariamente después del bucle

Si accidentalmente indentas código que debería ejecutarse después de que haya terminado un bucle, ese código se repetirá una vez para cada elemento de la lista. A veces esto hace que Python informe de un error, pero a menudo provocará un error lógico.

Por ejemplo, veamos qué ocurre cuando accidentalmente aplicamos una sangría a la línea que agradecía a los magos en grupo haber realizado un buen espectáculo:

**magicians.py**

magicians = ['alice', 'david', 'carolina']  
for magician in magicians:  
 print(f"{magician.title()}, that was a great trick!")  
 print(f"I can't wait to see your next trick, {magician.title()}.\n")  
  
❶ print("Thank you everyone, that was a great magic show!")

Como la última línea ❶ está sangrada, se imprime una vez por cada persona de la lista:

Alice, that was a great trick!  
I can't wait to see your next trick, Alice.  
  
Thank you everyone, that was a great magic show!  
David, that was a great trick!  
I can't wait to see your next trick, David.  
  
Thank you everyone, that was a great magic show!  
Carolina, that was a great trick!  
I can't wait to see your next trick, Carolina.  
  
Thank you everyone, that was a great magic show!

Éste es otro error lógico, similar al de "Olvidar sangrar líneas adicionales" en la página 54. Como Python no sabe lo que intentas conseguir con tu código, ejecutará todo el código que esté escrito con una sintaxis válida. Si una acción se repite muchas veces cuando sólo debería ejecutarse una vez, probablemente necesites eliminar la sangría del código de esa acción.

### Olvidar los dos puntos

Los dos puntos al final de una sentencia for indican a Python que interprete la línea siguiente como el inicio de un bucle.

magicians = ['alice', 'david', 'carolina']  
❶ for magician in magicians  
 print(magician)

Si accidentalmente olvidas los dos puntos ❶, obtendrás un error de sintaxis porque Python no sabe exactamente lo que estás intentando hacer:

File "magicians.py", line 2  
 for magician in magicians  
 ^  
SyntaxError: expected ':'

Python no sabe si simplemente olvidaste los dos puntos, o si pretendías escribir código adicional para configurar un bucle más complejo. Si el intérprete puede identificar una posible solución, te sugerirá una, como añadir dos puntos al final de la línea, como hace aquí con la respuesta expected ':'. Algunos errores tienen soluciones fáciles y obvias, gracias a las sugerencias de los rastreadores de Python. Algunos errores son mucho más difíciles de resolver, incluso cuando el arreglo final sólo implica un único carácter. No te sientas mal si tardas mucho tiempo en encontrar una pequeña solución; no estás solo en esta experiencia.

## Inténtalo tú mismo

4-1. Pizzas: Piensa en al menos tres tipos de tu pizza favorita. Almacena estos nombres de pizza en una lista, y luego utiliza un bucle for para imprimir el nombre de cada pizza.

* Modifica tu bucle for para imprimir una frase utilizando el nombre de la pizza, en lugar de imprimir sólo el nombre de la pizza. Para cada pizza, debes tener una línea de salida que contenga una sentencia simple como *I like pepperoni pizza*.
* Añade una línea al final de tu programa, fuera del bucle for, que diga cuánto te gusta la pizza. La salida debe consistir en tres o más líneas sobre los tipos de pizza que te gustan y luego una frase adicional, como por ejemplo *I really love pizza!*

4-2. Animales: Piensa en al menos tres animales diferentes que tengan una característica común. Almacena los nombres de estos animales en una lista y, a continuación, utiliza un bucle for para imprimir el nombre de cada animal.

* Modifica tu programa para imprimir una declaración sobre cada animal, como por ejemplo *A dog would make a great pet.*
* Añade una línea al final de tu programa, indicando qué tienen en común estos animales. Podrías imprimir una frase como *Any of these animals would make a great pet!*

## Hacer listas numéricas

Existen muchas razones para almacenar un conjunto de números. Por ejemplo, necesitarás llevar un registro de las posiciones de cada personaje en un juego, y puede que quieras que también lleve un registro de las puntuaciones más altas de un jugador. En las visualizaciones de datos, casi siempre trabajarás con conjuntos de números, como temperaturas, distancias, tamaños de población o valores de latitud y longitud, entre otros tipos de conjuntos numéricos.

Las listas son ideales para almacenar conjuntos de números, y Python proporciona diversas herramientas para ayudarte a trabajar eficazmente con listas de números. Una vez que entiendas cómo utilizar estas herramientas con eficacia, tu código funcionará bien incluso cuando tus listas contengan millones de elementos.

### Uso de la función range()

La función range() de Python facilita la generación de una serie de números. Por ejemplo, puedes utilizar la función range() para imprimir una serie de números como ésta:

**first\_numbers.py**

for value in range(1, 5):  
 print(value)

Aunque parece que este código debería imprimir los números del 1 al 5, no imprime el número 5:

1  
2  
3  
4

En este ejemplo, range() sólo imprime los números del 1 al 4. Este es otro resultado del comportamiento "de uno en uno" que verás a menudo en los lenguajes de programación. La función range() hace que Python empiece a contar en el primer valor que le das, y se detiene cuando alcanza el segundo valor que le proporcionas. Como se detiene en ese segundo valor, la salida nunca contiene el valor final, que en este caso habría sido 5.

Para imprimir los números del 1 al 5, utilizarías range(1, 6):

for value in range(1, 6):  
 print(value)

Esta vez la salida empieza en 1 y termina en 5:

1  
2  
3  
4  
5

Si tu salida es distinta de la que esperas cuando utilizas range(), prueba a ajustar tu valor final en 1.

También puedes pasar a range() sólo un argumento, y empezará la secuencia de números en 0. Por ejemplo, range(6) devolvería los números del 0 al 5.

### Utilizar range() para hacer una lista de números

Si quieres hacer una lista de números, puedes convertir los resultados de range() directamente en una lista utilizando la función list(). Cuando envuelvas list() en una llamada a la función range(), la salida será una lista de números.

En el ejemplo de la sección anterior, simplemente imprimimos una serie de números. Podemos utilizar list() para convertir esa misma serie de números en una lista:

numbers = list(range(1, 6))  
print(numbers)

Este es el resultado:

[1, 2, 3, 4, 5]

También podemos utilizar la función range() para decirle a Python que se salte los números de un rango determinado. Si pasas un tercer argumento a range(), Python utiliza ese valor como tamaño de paso al generar los números.

Por ejemplo, aquí tienes cómo hacer una lista de los números pares entre el 1 y el 10:

**even\_numbers.py**

even\_numbers = list(range(2, 11, 2))  
print(even\_numbers)

En este ejemplo, la función range() comienza con el valor 2 y luego suma 2 a ese valor. Suma 2 repetidamente hasta que alcanza o sobrepasa el valor final, 11, y produce este resultado:

[2, 4, 6, 8, 10]

Puedes crear casi cualquier conjunto de números que desees utilizando la función range(). Por ejemplo, considera cómo podrías hacer una lista de los 10 primeros números cuadrados (es decir, el cuadrado de cada número entero del 1 al 10). En Python, dos asteriscos (\*\*) representan exponentes. He aquí cómo podrías poner los 10 primeros números cuadrados en una lista:

**square\_numbers.py**

squares = []  
for value in range(1, 11):  
❶ square = value \*\* 2  
❷ squares.append(square)  
  
print(squares)

Empezamos con una lista vacía llamada squares. A continuación, le decimos a Python que repase cada valor del 1 al 10 utilizando la función range(). Dentro del bucle, el valor actual se eleva a la segunda potencia y se asigna a la variable square ❶. A continuación, cada nuevo valor de square se añade a la lista squares ❷. Por último, cuando el bucle ha terminado de ejecutarse, se imprime la lista de cuadrados:

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Para escribir este código de forma más concisa, omite la variable temporal square y añade cada nuevo valor directamente a la lista:

squares = []  
for value in range(1,11):  
 squares.append(value\*\*2)  
  
print(squares)

Esta línea hace el mismo trabajo que las líneas dentro del bucle for del listado anterior. Cada valor del bucle se eleva a la segunda potencia y se añade inmediatamente a la lista de cuadrados.

Puedes utilizar cualquiera de estos métodos cuando hagas listas más complejas. A veces, utilizar una variable temporal hace que tu código sea más fácil de leer; otras veces hace que el código sea innecesariamente largo. Céntrate primero en escribir código que entiendas claramente y que haga lo que quieres que haga. Luego busca enfoques más eficientes a medida que revises tu código.

### Estadísticas sencillas con una lista de números

Algunas funciones de Python son útiles cuando se trabaja con listas de números. Por ejemplo, puedes encontrar fácilmente el mínimo, el máximo y la suma de una lista de números:

>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  
>>> min(digits)  
0  
>>> max(digits)  
9  
>>> sum(digits)  
45

## Nota

Los ejemplos de esta sección utilizan listas cortas de números que caben fácilmente en la página. Funcionarían igual de bien si tu lista contuviera un millón de números o más.

### Comprensión de listas

El método descrito anteriormente para generar la lista squares consistía en utilizar tres o cuatro líneas de código. Un *list comprehension* te permite generar esta misma lista en una sola línea de código. Una comprensión de lista combina el bucle for y la creación de nuevos elementos en una sola línea, y añade automáticamente cada nuevo elemento. Las comprensiones de lista no siempre se presentan a los principiantes, pero las he incluido aquí porque lo más probable es que las veas en cuanto empieces a mirar el código de otras personas.

El siguiente ejemplo construye la misma lista de números cuadrados que has visto antes, pero utiliza una comprensión de lista:

**squares.py**

squares = [value\*\*2 for value in range(1, 11)]  
print(squares)

Para utilizar esta sintaxis, empieza con un nombre descriptivo para la lista, como squares. A continuación, abre un conjunto de corchetes y define la expresión para los valores que quieras almacenar en la nueva lista. En este ejemplo, la expresión es value\*\*2, que eleva el valor a la segunda potencia. A continuación, escribe un bucle for para generar los números que quieras introducir en la expresión, y cierra los corchetes. El bucle for de este ejemplo es for value in range(1, 11), que introduce los valores del 1 al 10 en la expresión value\*\*2. Observa que no se utilizan dos puntos al final de la expresión for.

El resultado es la misma lista de números cuadrados que has visto antes:

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Escribir tus propias comprensiones de listas requiere práctica, pero te merecerá la pena una vez que te sientas cómodo creando listas normales. Cuando estés escribiendo tres o cuatro líneas de código para generar listas y empiece a parecerte repetitivo, plantéate escribir tus propias comprensiones de listas.

## Pruébalo tú mismo

4-3. Contar hasta veinte: Utiliza un bucle for para imprimir los números del 1 al 20, ambos inclusive.

4-4. Un millón: Haz una lista de los números del uno al millón, y luego utiliza un bucle for para imprimir los números. (Si la salida tarda demasiado, deténla pulsando CTRL-C o cerrando la ventana de salida).

4-5. Suma de un millón: Haz una lista de los números del uno al millón, y luego utiliza min() y max() para asegurarte de que tu lista empieza realmente en uno y termina en un millón. Además, utiliza la función sum() para ver lo rápido que Python puede sumar un millón de números.

4-6. Números impares: Utiliza el tercer argumento de la función range() para hacer una lista de los números impares del 1 al 20. Utiliza un bucle for para imprimir cada número.

4-7. Treses: Haz una lista de los múltiplos de 3, del 3 al 30. Utiliza un bucle for para imprimir los números de tu lista.

4-8. Cubos: Un número elevado a la tercera potencia se llama *cube*. Por ejemplo, el cubo de 2 se escribe como 2\*\*3 en Python. Haz una lista de los 10 primeros cubos (es decir, el cubo de cada número entero del 1 al 10), y utiliza un bucle for para imprimir el valor de cada cubo.

4-9. Comprensión de cubos: Utiliza una comprensión de lista para generar una lista de los 10 primeros cubos.

## Trabajar con parte de una lista

En el Capítulo 3 aprendiste a acceder a elementos individuales de una lista, y en este capítulo has aprendido a trabajar con todos los elementos de una lista. También puedes trabajar con un grupo específico de elementos de una lista, que en Python se llama *slice*.

### Cortar una lista

Para hacer un corte, especifica el índice del primer y último elemento con el que quieres trabajar. Al igual que con la función range(), Python se detiene un elemento antes del segundo índice que especifiques. Para obtener los tres primeros elementos de una lista, solicitarías los índices 0 a 3, lo que devolvería los elementos 0, 1 y 2.

El siguiente ejemplo se refiere a una lista de jugadores de un equipo:

**players.py**

players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print(players[0:3])

Este código imprime una porción de la lista. La salida conserva la estructura de la lista e incluye los tres primeros jugadores de la lista:

['charles', 'martina', 'michael']

Puedes generar cualquier subconjunto de una lista. Por ejemplo, si quieres los elementos segundo, tercero y cuarto de una lista, empezarías el trozo en el índice 1 y lo terminarías en el índice 4:

players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print(players[1:4])

Esta vez el corte comienza en 'martina' y termina en 'florence':

['martina', 'michael', 'florence']

Si omites el primer índice en un trozo, Python inicia automáticamente tu trozo al principio de la lista:

players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print(players[:4])

Sin índice inicial, Python comienza al principio de la lista:

['charles', 'martina', 'michael', 'florence']

Una sintaxis similar funciona si quieres un trozo que incluya el final de una lista. Por ejemplo, si quieres todos los elementos desde el tercero hasta el último, puedes empezar con el índice 2 y omitir el segundo índice:

players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print(players[2:])

Python devuelve todos los elementos desde el tercer elemento hasta el final de la lista:

['michael', 'florence', 'eli']

Esta sintaxis te permite obtener todos los elementos desde cualquier punto de tu lista hasta el final, independientemente de la longitud de la lista. Recuerda que un índice negativo devuelve un elemento a cierta distancia del final de una lista; por tanto, puedes dar salida a cualquier trozo del final de una lista. Por ejemplo, si queremos obtener los nombres de los tres últimos jugadores de la lista, podemos utilizar el corte players[-3:]:

players = ['charles', 'martina', 'michael', 'florence', 'eli']  
print(players[-3:])

Esto imprime los nombres de los tres últimos jugadores y seguirá funcionando a medida que la lista de jugadores cambie de tamaño.

## Nota

Puedes incluir un tercer valor entre los corchetes indicando un corte. Si se incluye un tercer valor, esto indica a Python cuántos elementos debe omitir entre los elementos del rango especificado.

### Recorrer una porción

Puedes utilizar una porción en un bucle for si quieres recorrer un subconjunto de elementos de una lista. En el siguiente ejemplo, recorremos los tres primeros jugadores e imprimimos sus nombres como parte de una lista simple:

players = ['charles', 'martina', 'michael', 'florence', 'eli']  
  
print("Here are the first three players on my team:")  
❶ for player in players[:3]:  
 print(player.title())

En lugar de recorrer toda la lista de jugadores, Python recorre sólo los tres primeros nombres ❶:

Here are the first three players on my team:  
Charles  
Martina  
Michael

Las listas son muy útiles en varias situaciones. Por ejemplo, cuando estás creando un juego, podrías añadir la puntuación final de un jugador a una lista cada vez que ese jugador termine de jugar. Luego podrías obtener las tres mejores puntuaciones de un jugador ordenando la lista en orden decreciente y tomando una porción que incluya sólo las tres primeras puntuaciones. Cuando trabajes con datos, puedes utilizar trozos para procesar tus datos en partes de un tamaño específico. O, cuando estés construyendo una aplicación web, puedes utilizar trozos para mostrar información en una serie de páginas con una cantidad adecuada de información en cada una de ellas.

### Copiar una lista

A menudo, querrás empezar con una lista existente y hacer una lista completamente nueva basada en la primera. Exploremos cómo funciona copiar una lista y examinemos una situación en la que copiar una lista resulta útil.

Para copiar una lista, puedes hacer un corte que incluya toda la lista original omitiendo el primer índice y el segundo índice ([:]). Esto le dice a Python que haga un corte que empiece en el primer elemento y termine en el último, produciendo una copia de toda la lista.

Por ejemplo, imagina que tenemos una lista de nuestras comidas favoritas y queremos hacer otra lista de las comidas que le gustan a un amigo. A este amigo le gusta todo lo que hay en nuestra lista hasta ahora, así que podemos crear su lista copiando la nuestra:

**foods.py**

my\_foods = ['pizza', 'falafel', 'carrot cake']  
❶ friend\_foods = my\_foods[:]  
  
print("My favorite foods are:")  
print(my\_foods)  
  
print("\nMy friend's favorite foods are:")  
print(friend\_foods)

Primero, hacemos una lista de los alimentos que nos gustan llamada my\_foods. Luego hacemos una nueva lista llamada friend\_foods. Hacemos una copia de my\_foods pidiendo un trozo de my\_foods sin especificar ningún índice ❶, y asignamos la copia a friend\_foods. Cuando imprimimos cada lista, vemos que ambas contienen los mismos alimentos:

My favorite foods are:  
['pizza', 'falafel', 'carrot cake']  
  
My friend's favorite foods are:  
['pizza', 'falafel', 'carrot cake']

Para demostrar que en realidad tenemos dos listas separadas, añadiremos un nuevo alimento a cada lista y demostraremos que cada lista lleva la cuenta de los alimentos favoritos de la persona correspondiente:

my\_foods = ['pizza', 'falafel', 'carrot cake']  
❶ friend\_foods = my\_foods[:]  
  
❷ my\_foods.append('cannoli')  
❸ friend\_foods.append('ice cream')  
  
print("My favorite foods are:")  
print(my\_foods)  
  
print("\nMy friend's favorite foods are:")  
print(friend\_foods)

Copiamos los elementos originales de my\_foods en la nueva lista friend\_foods, como hicimos en el ejemplo anterior ❶. A continuación, añadimos un nuevo alimento a cada lista: añadimos 'cannoli' a my\_foods ❷, y añadimos 'ice cream' a friend\_foods ❸. A continuación, imprimimos las dos listas para ver si cada uno de estos alimentos está en la lista adecuada:

My favorite foods are:  
['pizza', 'falafel', 'carrot cake', 'cannoli']  
  
My friend's favorite foods are:  
['pizza', 'falafel', 'carrot cake', 'ice cream']

La salida muestra que 'cannoli' aparece ahora en nuestra lista de alimentos favoritos, pero 'ice cream' no. Podemos ver que 'ice cream' aparece ahora en la lista de nuestro amigo pero 'cannoli' no. Si simplemente hubiéramos establecido friend\_foods igual a my\_foods, no produciríamos dos listas separadas. Por ejemplo, esto es lo que ocurre cuando intentas copiar una lista sin utilizar un corte:

my\_foods = ['pizza', 'falafel', 'carrot cake']  
  
# This doesn't work:  
friend\_foods = my\_foods  
  
my\_foods.append('cannoli')  
friend\_foods.append('ice cream')  
  
print("My favorite foods are:")  
print(my\_foods)  
  
print("\nMy friend's favorite foods are:")  
print(friend\_foods)

En lugar de asignar una copia de my\_foods a friend\_foods, establecemos friend\_foods igual a my\_foods. Esta sintaxis en realidad le dice a Python que asocie la nueva variable friend\_foods a la lista que ya está asociada a my\_foods, de modo que ahora ambas variables apuntan a la misma lista. Como resultado, cuando añadamos 'cannoli' a my\_foods, también aparecerá en friend\_foods. Del mismo modo, 'ice cream' aparecerá en ambas listas, aunque parezca que sólo se ha añadido a friend\_foods.

La salida muestra que ahora ambas listas son iguales, que no es lo que queríamos:

My favorite foods are:  
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']  
  
My friend's favorite foods are:  
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']

## Nota

No te preocupes por los detalles de este ejemplo por ahora. Si intentas trabajar con una copia de una lista y observas un comportamiento inesperado, asegúrate de que estás copiando la lista utilizando un corte, como hicimos en el primer ejemplo.

## Pruébalo tú mismo

4-10. Rebanadas: Utilizando uno de los programas que has escrito en este capítulo, añade varias líneas al final del programa que hagan lo siguiente:

* Imprimir el mensaje *The first three items in the list are:*. A continuación, utiliza una rebanada para imprimir los tres primeros elementos de la lista de ese programa.
* Imprime el mensaje *Three items from the middle of the list are:*. A continuación, utiliza un corte para imprimir tres elementos de la mitad de la lista.
* Imprime el mensaje *The last three items in the list are:*. A continuación, utiliza un corte para imprimir los tres últimos elementos de la lista.

4-11. Mis Pizzas, Tus Pizzas: Empieza con tu programa del Ejercicio 4-1(página 56). Haz una copia de la lista de pizzas y llámala friend\_pizzas. Después, haz lo siguiente:

* Añade una nueva pizza a la lista original.
* Añade otra pizza a la lista friend\_pizzas.
* Demuestra que tienes dos listas separadas. Imprime el mensaje *My favorite pizzas are:*, y luego utiliza un bucle for para imprimir la primera lista. Imprime el mensaje *My friend’s favorite pizzas are:*, y luego utiliza un bucle for para imprimir la segunda lista. Asegúrate de que cada nueva pizza se almacena en la lista adecuada.

4-12. Más bucles: Todas las versiones de *foods.py* de esta sección han evitado utilizar bucles for al imprimir, para ahorrar espacio. Elige una versión de *foods.py*, y escribe dos bucles for para imprimir cada lista de alimentos.

## Tuplas

Las listas funcionan bien para almacenar colecciones de elementos que pueden cambiar a lo largo de la vida de un programa. La capacidad de modificar listas es especialmente importante cuando trabajas con una lista de usuarios en un sitio web o una lista de personajes en un juego. Sin embargo, a veces querrás crear una lista de elementos que no puedan cambiar. Las tuplas te permiten hacer precisamente eso. Python se refiere a los valores que no pueden cambiar como *immutable*, y una lista inmutable se llama *tuple*.

### Definir una tupla

Una tupla tiene el mismo aspecto que una lista, salvo que utilizas paréntesis en lugar de corchetes. Una vez definida una tupla, puedes acceder a los elementos individuales utilizando el índice de cada elemento, igual que harías con una lista.

Por ejemplo, si tenemos un rectángulo que siempre debe tener un tamaño determinado, podemos asegurarnos de que su tamaño no cambie poniendo las dimensiones en una tupla:

**dimensions.py**

dimensions = (200, 50)  
print(dimensions[0])  
print(dimensions[1])

Definimos la tupla dimensions, utilizando paréntesis en lugar de corchetes. A continuación, imprimimos cada elemento de la tupla individualmente, utilizando la misma sintaxis que hemos estado utilizando para acceder a los elementos de una lista:

200  
50

Veamos qué ocurre si intentamos cambiar uno de los elementos de la tupla dimensions:

dimensions = (200, 50)  
dimensions[0] = 250

Este código intenta cambiar el valor de la primera dimensión, pero Python devuelve un error de tipo. Como estamos intentando modificar una tupla, cosa que no se puede hacer con ese tipo de objeto, Python nos dice que no podemos asignar un nuevo valor a un elemento de una tupla:

Traceback (most recent call last):  
 File "dimensions.py", line 2, in <module>  
 dimensions[0] = 250  
TypeError: 'tuple' object does not support item assignment

Esto es beneficioso porque queremos que Python lance un error cuando una línea de código intente cambiar las dimensiones del rectángulo.

## Nota

Las tuplas se definen técnicamente por la presencia de una coma; los paréntesis les dan un aspecto más ordenado y legible. Si quieres definir una tupla con un solo elemento, debes incluir una coma al final:

my\_t = (3,)

No suele tener sentido construir una tupla con un solo elemento, pero puede ocurrir cuando las tuplas se generan automáticamente.

### Recorrer todos los valores de una tupla

Puedes recorrer todos los valores de una tupla utilizando un bucle for, igual que hacías con una lista:

dimensions = (200, 50)  
for dimension in dimensions:  
 print(dimension)

Python devuelve todos los elementos de la tupla, igual que lo haría con una lista:

200  
50

### Escribir sobre una tupla

Aunque no puedes modificar una tupla, puedes asignar un nuevo valor a una variable que represente una tupla. Por ejemplo, si quisiéramos cambiar las dimensiones de este rectángulo, podríamos redefinir toda la tupla:

dimensions = (200, 50)  
print("Original dimensions:")  
for dimension in dimensions:  
 print(dimension)  
  
dimensions = (400, 100)  
print("\nModified dimensions:")  
for dimension in dimensions:  
 print(dimension)

Las cuatro primeras líneas definen la tupla original e imprimen las dimensiones iniciales. A continuación, asociamos una nueva tupla a la variable dimensions, e imprimimos los nuevos valores. Esta vez Python no produce ningún error, porque reasignar una variable es válido:

Original dimensions:  
200  
50  
  
Modified dimensions:  
400  
100

Comparadas con las listas, las tuplas son estructuras de datos sencillas. Utilízalas cuando quieras almacenar un conjunto de valores que no deban modificarse a lo largo de la vida de un programa.

## Pruébalo tú mismo

4-13. Buffet: Un restaurante tipo buffet sólo ofrece cinco alimentos básicos. Piensa en cinco comidas sencillas y almacénalas en una tupla.

* Utiliza un bucle for para imprimir cada uno de los alimentos que ofrece el restaurante.
* Intenta modificar uno de los elementos, y asegúrate de que Python rechaza el cambio.
* El restaurante cambia su menú, sustituyendo dos de los elementos por alimentos diferentes. Añade una línea que reescriba la tupla y, a continuación, utiliza un bucle for para imprimir cada uno de los elementos del menú revisado.

## Dar estilo a tu código

Ahora que estás escribiendo programas más largos, es una buena idea aprender a dar estilo a tu código de forma coherente. Tómate el tiempo necesario para que tu código sea lo más fácil de leer posible. Escribir código fácil de leer te ayuda a seguir la pista de lo que hacen tus programas y ayuda también a los demás a entender tu código.

Los programadores de Python han acordado una serie de convenciones de estilo para garantizar que el código de todos se estructure más o menos de la misma manera. Una vez que hayas aprendido a escribir código Python limpio, deberías ser capaz de entender la estructura general del código Python de cualquier otra persona, siempre que siga las mismas pautas. Si esperas convertirte en un programador profesional en algún momento, deberías empezar a seguir estas directrices lo antes posible para desarrollar buenos hábitos.

### La Guía de Estilo

Cuando alguien quiere hacer un cambio en el lenguaje Python, escribe un *Python Enhancement Proposal (PEP)*. Uno de los PEP más antiguos es *PEP 8*, que instruye a los programadores de Python sobre cómo dar estilo a su código. La PEP 8 es bastante larga, pero gran parte de ella se refiere a estructuras de codificación más complejas que las que has visto hasta ahora.

La guía de estilo de Python se escribió teniendo en cuenta que el código se lee más a menudo de lo que se escribe. Escribirás tu código una vez y luego empezarás a leerlo cuando empieces a depurarlo. Cuando añadas funciones a un programa, pasarás más tiempo leyendo tu código. Cuando compartas tu código con otros programadores, ellos también leerán tu código.

Si te dan a elegir entre escribir código que sea más fácil de escribir o código que sea más fácil de leer, los programadores de Python casi siempre te animarán a escribir código que sea más fácil de leer. Las siguientes directrices te ayudarán a escribir código claro desde el principio.

### Sangría

PEP 8 recomienda que utilices cuatro espacios por nivel de sangría. Utilizar cuatro espacios mejora la legibilidad, al tiempo que deja espacio para varios niveles de sangría en cada línea.

En un documento de procesamiento de textos, la gente suele utilizar tabuladores en lugar de espacios para sangrar. Esto funciona bien para los documentos de procesamiento de textos, pero el intérprete de Python se confunde cuando se mezclan tabuladores con espacios. Todos los editores de texto ofrecen una opción que te permite utilizar la tecla TAB, pero que convierte cada tabulación en un número determinado de espacios. Definitivamente, debes utilizar la tecla TAB, pero asegúrate también de que tu editor está configurado para insertar espacios en lugar de tabuladores en tu documento.

Mezclar tabuladores y espacios en tu archivo puede causar problemas muy difíciles de diagnosticar. Si crees que tienes una mezcla de tabuladores y espacios, puedes convertir todos los tabuladores de un archivo en espacios en la mayoría de los editores.

### Longitud de línea

Muchos programadores de Python recomiendan que cada línea tenga menos de 80 caracteres. Históricamente, esta directriz se desarrolló porque en la mayoría de los ordenadores sólo cabían 79 caracteres en una sola línea en una ventana de terminal. Actualmente, la gente puede hacer caber líneas mucho más largas en sus pantallas, pero existen otras razones para adherirse a la longitud de línea estándar de 79 caracteres.

Los programadores profesionales suelen tener varios archivos abiertos en la misma pantalla, y utilizar la longitud de línea estándar les permite ver líneas enteras en dos o tres archivos que están abiertos uno al lado del otro en la pantalla. PEP 8 también recomienda que limites todos tus comentarios a 72 caracteres por línea, porque algunas de las herramientas que generan documentación automática para proyectos grandes añaden caracteres de formato al principio de cada línea comentada.

Las directrices de PEP 8 sobre la longitud de las líneas no son inamovibles, y algunos equipos prefieren un límite de 99 caracteres. No te preocupes demasiado por la longitud de línea en tu código mientras aprendes, pero ten en cuenta que las personas que trabajan en colaboración casi siempre siguen las directrices de PEP 8. La mayoría de los editores te permiten configurar una señal visual, normalmente una línea vertical en tu pantalla, que te indica dónde están estos límites.

## Nota

El Apéndice B te muestra cómo configurar tu editor de texto para que inserte siempre cuatro espacios cada vez que pulses la tecla TAB y te muestre una directriz vertical que te ayude a seguir el límite de 79 caracteres.

### Líneas en blanco

Para agrupar partes de tu programa visualmente, utiliza líneas en blanco. Debes utilizar líneas en blanco para organizar tus archivos, pero no lo hagas en exceso. Siguiendo los ejemplos de este libro, conseguirás el equilibrio adecuado. Por ejemplo, si tienes cinco líneas de código que construyen una lista y luego otras tres líneas que hacen algo con esa lista, es apropiado colocar una línea en blanco entre las dos secciones. Sin embargo, no debes colocar tres o cuatro líneas en blanco entre las dos secciones.

Las líneas en blanco no afectarán a la ejecución de tu código, pero sí a su legibilidad. El intérprete de Python utiliza la sangría horizontal para interpretar el significado de tu código, pero no tiene en cuenta el espaciado vertical.

### Otras directrices de estilo

PEP 8 tiene muchas recomendaciones de estilo adicionales, pero la mayoría de las directrices se refieren a programas más complejos que lo que estás escribiendo en este momento. A medida que aprendas estructuras de Python más complejas, compartiré las partes relevantes de las directrices de la PEP 8.

## Pruébalo tú mismo

4-14. PEP 8: Consulta la guía de estilo original de PEP 8 en [https://python.org/dev/peps/pep-0008.](https://python.org/dev/peps/pep-0008) No usarás mucho de ella ahora, pero puede ser interesante hojearla.

4-15. Revisión del código: Elige tres de los programas que has escrito en este capítulo y modifica cada uno de ellos para que cumpla con PEP 8.

* Utiliza cuatro espacios para cada nivel de sangría. Configura tu editor de texto para que inserte cuatro espacios cada vez que pulses la tecla TAB, si aún no lo has hecho (en el Apéndice B encontrarás instrucciones sobre cómo hacerlo).
* Utiliza menos de 80 caracteres en cada línea, y configura tu editor para que muestre una línea vertical en la posición del carácter 80.
* No utilices excesivamente líneas en blanco en tus archivos de programa.

## Resumen

En este capítulo has aprendido a trabajar eficazmente con los elementos de una lista. Aprendiste a trabajar a través de una lista utilizando un bucle for, cómo Python utiliza la sangría para estructurar un programa y cómo evitar algunos errores comunes de sangría. Aprendiste a hacer listas numéricas sencillas, así como algunas operaciones que puedes realizar con listas numéricas. Aprendiste a cortar una lista para trabajar con un subconjunto de elementos y a copiar listas correctamente utilizando un corte. También aprendiste sobre las tuplas, que proporcionan cierto grado de protección a un conjunto de valores que no deben cambiar, y cómo dar estilo a tu código, cada vez más complejo, para que sea fácil de leer.

En el Capítulo 5, aprenderás a responder adecuadamente a diferentes condiciones utilizando las sentencias if. Aprenderás a encadenar conjuntos relativamente complejos de pruebas condicionales para responder adecuadamente exactamente al tipo de situación o información que estés buscando. También aprenderás a utilizar las sentencias if mientras recorres una lista en bucle para realizar acciones específicas con los elementos seleccionados de una lista.