Desconocido

## 3 Cadenas, listas, tuplas y diccionarios



En [el Capítulo 2](ch02.xhtml#ch02) , hicimos algunos cálculos básicos con Python y aprendimos sobre las variables. En este capítulo trabajaremos con otros elementos de los programas Python: cadenas, listas, tuplas y diccionarios. Utilizarás cadenas para mostrar mensajes en tus programas (como los mensajes "Preparados" y "Fin de partida" en un juego). También descubrirás cómo se utilizan las listas, las tuplas y los diccionarios para almacenar colecciones de cosas.

### Cadenas

Cuando programamos, solemos llamar cadena al texto *.* Piensa en una cadena como una colección de letras. Todas las letras, números y símbolos de este libro podrían ser una cadena, al igual que tu nombre y tu dirección. De hecho, el primer programa Python que creamos en [el Capítulo](ch01.xhtml#ch01) 1 utilizaba una cadena: "Hola Mundo".

### Crear cadenas

En Python, creamos una cadena entrecomillando el texto porque los lenguajes de programación necesitan distinguir entre distintos tipos de valores. (Tenemos que decirle al ordenador si un valor es un número, una cadena u otra cosa). Por ejemplo, podríamos tomar nuestra variable fred del [Capítulo 2](ch02.xhtml#ch02) y utilizarla para etiquetar una cadena:



fred = "Why do gorillas have big nostrils? Big fingers!!"

Entonces, para ver lo que hay "dentro" de fred , podríamos introducir print(fred) :

>>> print(fred)  
Why do gorillas have big nostrils? Big fingers!!

También puedes utilizar comillas simples para crear una cadena, como ésta:

>>> fred = 'What is pink and fluffy? Pink fluff!!'  
>>> print(fred)  
What is pink and fluffy? Pink fluff!!

Sin embargo, si intentas introducir más de una línea de texto para tu cadena utilizando sólo una comilla simple ( ′ ) o doble ( " ) o si empiezas con un tipo de comilla y terminas con otro, recibirás un mensaje de error en la Shell de Python. Por ejemplo, introduce la siguiente línea:

>>> fred = "How do dinosaurs pay their bills?

Verás este resultado:

SyntaxError: EOL while scanning string literal

Este es un mensaje de error que se queja de la sintaxis porque no has seguido las reglas para terminar una cadena con comillas simples o dobles.

*Sintaxis* significa la disposición y el orden de las palabras en una frase o, en este caso, la disposición y el orden de las palabras y los símbolos en un programa. Por lo tanto, *SyntaxError* significa que has hecho algo en un orden que Python no esperaba, o que Python esperaba algo que tú has pasado por alto. *EOL* significa *final de línea* , por lo que el resto del mensaje de error te dice que Python ha llegado al final de la línea y no ha encontrado una comilla doble para cerrar (o terminar) la cadena.

Para utilizar más de una línea de texto en tu cadena (lo que se llama una *cadena multilínea* ), utiliza tres comillas simples ( ’’’ ), y luego pulsa ENTER entre línea y línea, así:

>>> fred = '''How do dinosaurs pay their bills?  
 With tyrannosaurus checks!'''

Imprimamos el contenido de fred para ver si ha funcionado:

>>> print(fred)  
How do dinosaurs pay their bills?  
With tyrannosaurus checks!

### Problemas con las cadenas

Considera ahora este tonto ejemplo de una cadena, que hace que Python muestre un mensaje de error:

>>> silly\_string = 'He said, "Aren't can't shouldn't wouldn't."'  
SyntaxError: invalid syntax

En la primera línea, intentamos crear una cadena (definida como la variable silly\_string ) encerrada entre comillas simples, pero que también contiene una mezcla de comillas simples en las palabras can’t , shouldn’t , y wouldn’t , así como comillas dobles. ¡Menudo lío!

Recuerda que Python no es tan inteligente como un ser humano, así que todo lo que ve es una cadena que contiene He said, "Aren , seguida de un montón de otros caracteres que no espera. Cuando Python ve una comilla (ya sea simple o doble), espera que una cadena comience tras la primera comilla y que la cadena termine tras la siguiente comilla que coincida  (ya sea simple o doble) en esa línea. En este caso, el inicio de la cadena es la comilla simple antes de He , y el final de la cadena, en lo que respecta a Python, es la comilla simple después de n en Aren .

En la última línea, Python nos dice qué tipo de error se ha producido, en este ejemplo, un error de sintaxis.

Utilizar comillas dobles en lugar de simples también produce un error:

>>> silly\_string = "He said, "Aren't can't shouldn't wouldn't.""  
SyntaxError: invalid syntax

Aquí, Python ve una cadena entre comillas dobles, que contiene las letras He said, (y un espacio). Todo lo que sigue a esa cadena (a partir de Aren’t ) provoca el error.

Esto se debe a que, desde la perspectiva de Python, todo ese material extra no debería estar ahí. Python busca la siguiente comilla coincidente y no sabe qué quieres que haga con todo lo que sigue a esa comilla en la misma línea.

La solución a este problema es una cadena multilínea, de la que aprendimos antes, utilizando *tres* comillas simples ( ''' ). Esto nos permite combinar comillas dobles y simples en nuestra cadena sin provocar errores. De hecho, si utilizamos tres comillas simples, podemos poner cualquier combinación de comillas simples y dobles dentro de la cadena (siempre que no intentemos poner tres comillas simples). La versión sin errores de nuestra cadena tiene este aspecto:



silly\_string = '''He said, "Aren't can't shouldn't wouldn't."'''

Pero espera, hay más. Si realmente quieres utilizar comillas simples o dobles para rodear una cadena en Python, en lugar de tres comillas simples, puedes añadir una barra invertida (∖) antes de cada comilla dentro de la cadena. Esto se llama *escape .* Es una forma de decirle a Python: "Sí, sé que tengo comillas dentro de mi cadena, y quiero que las ignores hasta que veas la comilla final".

Escapar cadenas puede dificultar su lectura, por lo que se considera mejor práctica utilizar cadenas multilínea. Aun así, es posible que en  te encuentres con código que utiliza escapes, por lo que es útil saber por qué están ahí las barras invertidas.

Aquí tienes algunos ejemplos de cómo funciona el escape:

➊ >>> single\_quote\_str = 'He said, "Aren\'t can\'t shouldn\'t  
 wouldn\'t."'  
➋ >>> double\_quote\_str = "He said, \"Aren't can't shouldn't   
 wouldn't.\""  
 >>> print(single\_quote\_str)  
 He said, "Aren't can't shouldn't wouldn't."  
 >>> print(double\_quote\_str)  
 He said, "Aren't can't shouldn't wouldn't."

Primero, en ➊ , creamos una cadena con comillas simples, utilizando la barra invertida delante de las comillas simples dentro de esa cadena. En ➋ , creamos una cadena con comillas dobles, y utilizamos la barra invertida delante de esas comillas en la cadena. En las líneas siguientes, imprimimos las variables que acabamos de crear. Observa que el carácter barra invertida no aparece en las cadenas cuando las imprimimos.

### Incrustar valores en cadenas

Si quieres mostrar un mensaje utilizando el contenido de una variable, puedes incrustarla en una cadena especial, llamada cadena *f* (también conocida como *cadena literal formateada* ). Coloca llaves alrededor del nombre de la variable, que será sustituido por el valor real. ( *Incrustar valores* , también conocido como *sustitución de cadenas* , es el lenguaje de programación para "insertar valores").

Por ejemplo, para que Python calcule o almacene el número de puntos que has marcado en un partido, y luego lo añada a una frase como "He marcado 10 puntos", añade un f antes de la primera comilla y luego sustituye el número 10 por la variable rodeada de llaves {} , así:

>>> myscore = 1000  
>>> message = f'I scored {myscore} points'  
>>> print(message)  
I scored 1000 points

Aquí, creamos la variable myscore con el valor 1000 y la variable message que contiene la cadena f 'I scored {myscore}  points' . En la línea siguiente, llamamos a print(message) para ver el resultado de nuestra sustitución de cadena. El resultado de imprimir este mensaje es I scored 1000 points . No necesitamos utilizar una variable para el mensaje. Podríamos hacer el mismo ejemplo y utilizar sólo esto:

print(f'I scored {myscore} points')

También podemos utilizar más de una variable en una cadena:

>>> first = 0  
>>> second = 8  
>>> print(f'What did the number {first} say to the number {second}? Nice belt!!')  
What did the number 0 say to the number 8? Nice belt!!

Incluso podemos poner expresiones en una cadena f, así:

>>> print(f'Two plus two equals {2 + 2}')  
Two plus two equals 4

En este ejemplo, Python evalúa la ecuación simple entre las llaves, de modo que la cadena impresa contiene el resultado.

### Multiplicar cadenas

¿Cuánto es 10 multiplicado por 5? La respuesta es 50, por supuesto. Pero, ¿qué es 10 multiplicado por *?* Aquí tienes la respuesta de Python:

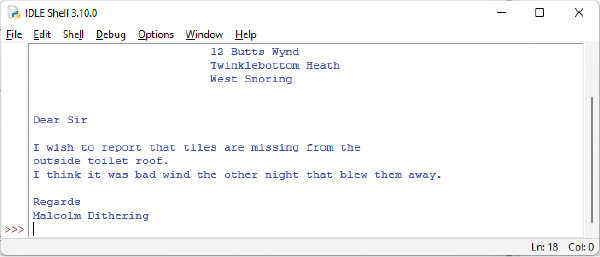
>>> print(10 \* 'a')  
aaaaaaaaaa

Los programadores de Python pueden multiplicar cadenas por varias razones, como para alinear el texto con un número específico de espacios al mostrar mensajes en la Shell de Python. Prueba a imprimir la siguiente letra en la Shell de Python (selecciona **Archivo** **▸ Nuevo** **Archivo** , e introduce el siguiente código):

spaces = ' ' \* 25  
print('{} 12 Butts Wynd')  
print('{} Twinklebottom Heath')  
print('{} West Snoring')  
print()  
print()  
print('Dear Sir')  
print()  
print('I wish to report that tiles are missing from the')  
print('outside toilet roof.')  
print('I think it was bad wind the other night that blew them away.')  
print()  
print('Regards')  
print('Malcolm Dithering')

Una vez que hayas escrito el código en la ventana de la Shell de Python, selecciona Archivo **▸** **Guardar como** . Nombra tu archivo *micarta.py .* A continuación, puedes ejecutar el código (como hemos hecho anteriormente) seleccionando **Ejecutar ▸** **Ejecutar módulo** .

En la primera línea de este ejemplo, creamos la variable spaces multiplicando un carácter de espacio por 25. A continuación, utilizamos esa variable en las tres líneas siguientes para alinear el texto a la derecha de la Shell de Python. Puedes ver el resultado de estas sentencias print a continuación:



*Figura 3-1: Ejecución del código de letras en* la Shell de Python

¿QUÉ SON LOS ARCHIVOS Y LAS CARPETAS?

Un *archivo* son datos (o información) de algún tipo que pueden almacenarse en tu ordenador. Los archivos pueden ser fotos, vídeos, libros electrónicos e incluso el informe escolar que escribiste en Microsoft Word.

Una *carpeta* (también llamada *directorio* ) es una colección de otras carpetas y archivos. Cuando hiciste clic en **Guardar como** para guardar tu archivo *micarta.py* , se almacenó en una carpeta.

Como veremos, los archivos y las carpetas son muy importantes en programación.

Además de utilizar la multiplicación para alinear, también podemos utilizarla para llenar la pantalla de mensajes molestos. Prueba con el siguiente ejemplo:

>>> print(1000 \* 'snirt')

### Las listas son más potentes que las cadenas

"Ancas de araña, dedo de rana, ala de murciélago, mantequilla de babosa y caspa de serpiente" no es una lista de la compra muy normal (a menos que seas un mago), pero la utilizaremos como primer ejemplo de las diferencias entre cadenas y listas. Podríamos almacenar esta lista de artículos en la variable wizard\_list utilizando una cadena como ésta:



>>> wizard\_list = 'spider legs, toe of frog, bat wing, slug butter, snake dandruff'  
>>> print(wizard\_list)  
spider legs, toe of frog, bat wing, slug butter, snake dandruff

Pero también podríamos crear una *lista* , un objeto Python un tanto mágico que podemos manipular. Así es como se verían estos elementos escritos como una lista:

>>> wizard\_list = ['spider legs', 'toe of frog', 'bat wing',  
 'slug butter', 'snake dandruff']  
>>> print(wizard\_list)  
['spider legs', 'toe of frog', 'bat wing', 'slug butter',  
'snake dandruff']

Para crear una lista hay que teclear un poco más que para crear una cadena, pero una lista es más útil que una cadena porque los elementos de la lista se pueden manipular fácilmente. Podemos imprimir un elemento de la lista introduciendo un número (llamado *posición del índice* ) entre corchetes, así:

>>> print(wizard\_list[2])  
bat wing

Si esperabas que el segundo elemento fuera ‘toe of frog’ , puede que te preguntes por qué se ha impreso ‘bat wing’ . Esto es  porque las listas empiezan en la posición índice 0, de modo que el primer elemento de una lista es 0, el segundo es 1 y el **tercero** es 2. Puede que esto no tenga mucho sentido para los humanos, pero sí para los ordenadores.

También podemos cambiar un elemento de una lista. Quizá nuestro amigo mago nos acaba de decir que tenemos que cogerle lengua de caracol en vez de ala de murciélago. Así es como cambiaríamos el elemento de nuestra lista:

>>> wizard\_list[2] = 'snail tongue'  
>>> print(wizard\_list)  
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',  
'snake dandruff']

Esto cambia el elemento en la posición 2 del índice, que antes era ala de murciélago, a lengua de caracol.

También podemos mostrar una sublista de los elementos de la lista. Para ello, utiliza dos puntos ( : ) dentro de los corchetes. Por ejemplo, escribe lo siguiente para ver los elementos tercero a quinto de nuestra lista (un brillante conjunto de ingredientes para un delicioso bocadillo):



>>> print(wizard\_list[2:5])  
['snail tongue', 'slug butter', 'snake dandruff']

Escribir [2:5] equivale a decir: "Muestra los elementos desde la posición índice 2 hasta (pero sin incluir) la posición índice 5", es decir, los elementos 3, 4 y 5.

Las listas pueden utilizarse para almacenar todo tipo de elementos, como números:

>>> some\_numbers = [1, 2, 5, 10, 20]

También pueden contener cadenas:

>>> some\_strings = ['Which', 'Witch', 'Is', 'Which']

Pueden tener mezclas de números y cadenas:

>>> numbers\_and\_strings = ['Why', 'was', 6, 'afraid', 'of', 7,   
 'because', 7, 8, 9]  
>>> print(numbers\_and\_strings)  
['Why', 'was', 6, 'afraid', 'of', 7, 'because', 7, 8, 9]

Y las listas pueden incluso almacenar otras listas:

>>> numbers = [1, 2, 3, 4]  
>>> strings = ['I', 'kicked', 'my', 'toe', 'and', 'it',   
 'is', 'sore']  
>>> mylist = [numbers, strings]  
>>> print(mylist)  
[[1, 2, 3, 4], ['I', 'kicked', 'my', 'toe', 'and', 'it',  
'is', 'sore']]

Este ejemplo de lista dentro de lista crea tres variables: numbers con cuatro números, strings con ocho cadenas y mylist con numbers y strings . La tercera lista ( mylist ) sólo tiene dos elementos porque es una lista de nombres de variables, no del contenido de las variables.

Podemos intentar imprimir los dos elementos de mylist por separado:

>>> print(mylist[0])  
[1, 2, 3, 4]  
>>> print(mylist[1])  
['I', 'kicked', 'my', 'toe', 'and', 'it', 'is', 'sore']

Aquí podemos ver que mylist[0] contiene la lista de números, y mylist[1] es la lista de cadenas.

### Añadir elementos a una lista

Para añadir elementos al final de una lista, utilizamos la función append . Por ejemplo, para añadir un eructo de oso (seguro que existe) a la lista de la compra del mago, introduce lo siguiente:

>>> wizard\_list.append('bear burp')  
>>> print(wizard\_list)  
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',   
'snake dandruff', 'bear burp']

Puedes seguir añadiendo más objetos mágicos a la lista del mago de la misma forma, así:

>>> wizard\_list.append('mandrake')  
>>> wizard\_list.append('hemlock')  
>>> wizard\_list.append('swamp gas')

Ahora la lista del mago tiene este aspecto:

>>> print(wizard\_list)  
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',  
'snake dandruff', 'bear burp', 'mandrake', 'hemlock', 'swamp gas']

¡Está claro que el mago está listo para hacer magia!

### Eliminar elementos de una lista

Para eliminar elementos de una lista, utiliza el comando del (abreviatura de *eliminar* ). Por ejemplo, para eliminar el quinto elemento de la lista del asistente, caspa de serpiente, haz esto

>>> del wizard\_list[4]  
>>> print(wizard\_list)  
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',  
'bear burp', 'mandrake', 'hemlock', 'swamp gas']

**NOTA**

*Recuerda que las posiciones empiezan en cero, por lo que wizard\_list[4] se refiere en realidad al quinto elemento de la lista.*

Prueba a eliminar los elementos que acabamos de añadir (mandrágora, cicuta y gas de los pantanos) introduciendo lo siguiente:

>>> del wizard\_list[7]  
>>> del wizard\_list[6]  
>>> del wizard\_list[5]  
>>> print(wizard\_list)  
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',  
'bear burp']

### Aritmética de listas

Podemos unir listas sumándolas, igual que sumamos números, utilizando el signo más ( + ). Por ejemplo, supongamos que tenemos dos listas: list1 , que contiene los números del 1 al 4, y list2 , que contiene algunas palabras. Podemos sumarlas utilizando el signo + , así:

>>> list1 = [1, 2, 3, 4]  
>>> list2 = ['I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']  
>>> print(list1 + list2)  
[1, 2, 3, 4, 'I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']

También podemos sumar las dos listas y asignar el resultado a otra variable:

>>> list1 = [1, 2, 3, 4]  
>>> list2 = ['I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']  
>>> list3 = list1 + list2  
>>> print(list3)  
[1, 2, 3, 4, 'I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']

Y podemos multiplicar una lista por un número. Por ejemplo, para multiplicar list1 por 5, escribimos list1 \* 5 :

>>> list1 = [1, 2]  
>>> print(list1 \* 5)  
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]

Esto le dice a Python que repita list1 cinco veces, dando como resultado 1, 2, 1, 2, 1, 2, 1, 2, 1, 2 . En cambio, la división ( / ) y la resta ( - ) sólo dan errores, como en estos ejemplos:

>>> list1 / 20  
Traceback (most recent call last):  
File "<pyshell>", line 1, in <module>  
 list1 / 20  
TypeError: unsupported operand type(s) for /: 'list' and 'int'  
  
>>> list1 - 20  
Traceback (most recent call last):  
File "<pyshell>", line 1, in <module>  
 list1 - 20  
TypeError: unsupported operand type(s) for -: 'list' and 'int'

Pero, ¿por qué? Bueno, unir listas con + y repetir listas con \* son operaciones bastante sencillas. Estos conceptos también tienen sentido en el mundo real. Por ejemplo, si te diera dos listas de la compra en papel y te dijera: "Suma estas dos listas", podrías escribir todos los elementos en otra hoja de papel en orden, de punta a punta. Lo mismo ocurriría si te dijera: "Multiplica esta lista por 3". Podrías imaginarte escribiendo una lista con todos los elementos de la lista tres veces en otra hoja de papel.

Pero, ¿cómo dividirías una lista? Por ejemplo, piensa cómo dividirías en dos una lista de seis números (del 1 al 6). Aquí tienes tres formas diferentes:

[1, 2, 3] [4, 5, 6]  
[1] [2, 3, 4, 5, 6]  
[1, 2, 3, 4] [5, 6]

¿Dividiríamos la lista por la mitad, la dividiríamos después del primer elemento, o simplemente elegiríamos un lugar al azar y la dividiríamos ahí? No hay una respuesta sencilla, y cuando le pides a Python que divida una lista, tampoco sabe qué hacer. Por eso responde con un error.



Por la misma razón, no puedes añadir nada que no sea una lista a una lista. Por ejemplo, esto es lo que ocurre cuando intentamos añadir el número 50 a list1 :

>>> list1 + 50  
Traceback (most recent call last):  
 File "<pyshell>", line 1, in <module>  
 list1 + 50  
TypeError: can only concatenate list (not "int") to list

¿Por qué se produce un error? Bien, ¿qué significa añadir 50 a una lista? ¿Significa añadir 50 a cada elemento? ¿Y si los elementos no son números? ¿Significa añadir el número 50 al final o al principio de la lista?

En programación informática, los comandos deben funcionar exactamente igual cada vez que los introduces. Tu ordenador sólo ve las cosas en blanco y negro. Pídele que tome una decisión complicada, y se echará las manos a la cabeza con errores.

### Tuplas

Una *tupla* es como una lista que utiliza paréntesis, como en este ejemplo:

>>> fibs = (0, 1, 1, 2, 3)  
>>> print(fibs[3])  
2

Aquí definimos la variable fibs como los números 0, 1, 1, 2 y 3. A continuación, al igual que con una lista, imprimimos el elemento en la posición 3 del índice de la tupla utilizando print(fibs[3]) .

La principal diferencia entre una tupla y una lista es que una tupla no puede cambiar una vez que la has creado. Por ejemplo, si intentamos sustituir el primer valor de la tupla fibs por el número 4 (igual que sustituimos los valores en nuestro wizard\_list ), obtendremos un mensaje de error:

>>> fibs[0] = 4  
Traceback (most recent call last):  
File "<pyshell>", line 1, in <module>  
 fibs[0] = 4  
TypeError: 'tuple' object does not support item assignment

¿Por qué utilizar una tupla en lugar de una lista? Bueno, a veces es útil utilizar algo que sabes que nunca puede cambiar. Si creas una tupla con dos elementos dentro, siempre tendrá esos dos elementos.

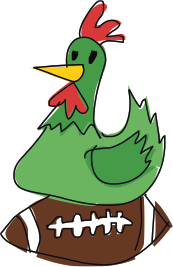
### Diccionarios en Python

En Python, un *dict* (abreviatura de *diccionario* ) es una colección de cosas, como las listas y las tuplas. La diferencia entre los dicts y las listas o tuplas es que cada elemento de un dict tiene una *clave* y un *valor* correspondiente *.*

Por ejemplo, supongamos que tenemos una lista de personas y sus deportes favoritos. Podríamos poner esta información en una lista Python, con el nombre de la persona seguido de su deporte, así

>>> favorite\_sports = ['Ralph Williams, Football',  
 'Michael Tippett, Basketball',  
 'Edward Elgar, Baseball',  
 'Rebecca Clarke, Netball',  
 'Ethel Smyth, Badminton',  
 'Frank Bridge, Rugby']

Si te pregunto cuál es el deporte favorito de Rebecca Clarke, podrías hojear la lista y encontrar que la respuesta es el netball. Pero, ¿y si la lista incluyera a 100 (o muchas más) personas?



Si almacenamos esta misma información en un diccionario, con el nombre de la persona como clave y su deporte favorito como valor, el código sería así:

>>> favorite\_sports = {'Ralph Williams' : 'Football',   
 'Michael Tippett' : 'Basketball',  
 'Edward Elgar' : 'Baseball',  
 'Rebecca Clarke' : 'Netball',  
 'Ethel Smyth' : 'Badminton',  
 'Frank Bridge' : 'Rugby'}

Utilizamos dos puntos para separar cada clave de su valor, y cada clave y cada valor van rodeados de comillas simples. Observa también que los elementos de un diccionario se encierran entre llaves ({}), no entre paréntesis o corchetes. El resultado es un dict (donde cada clave apunta a un valor concreto), como se muestra en [la Tabla 3-1 .](ch03.xhtml#ch03tab01)

Tabla**3-1:** Claves que apuntan a valores en un diccionario de deportes favoritos

| Clav**e** | Valo**r** |
| --- | --- |
| Ralph Williams | Fútbol |
| Michael Tippett | Baloncesto |
| Edward Elgar | Béisbol |
| Rebecca Clarke | Netball |
| Ethel Smyth | Bádminton |
| Frank Bridge | Rugby |

Ahora, para obtener el deporte favorito de Rebecca Clarke, accedemos a nuestro diccionario favorite\_sports utilizando su nombre como clave, así:

>>> print(favorite\_sports['Rebecca Clarke'])  
Netball

Y la respuesta es Netball.

Para eliminar un valor de un diccionario, utiliza su clave. Por ejemplo, eliminemos Ethel Smyth:

>>> del favorite\_sports['Ethel Smyth']  
>>> print(favorite\_sports)  
{'Rebecca Clarke': 'Netball', 'Michael Tippett': 'Basketball',   
'Ralph Williams': 'Football', 'Edward Elgar': 'Baseball',   
'Frank Bridge': 'Rugby'}

Para sustituir un valor en un dict, también utilizamos su clave. Supongamos que necesitamos cambiar el deporte favorito de Ralph Williams de Fútbol a Hockey sobre hielo. Podemos hacerlo así

>>> favorite\_sports['Ralph Williams'] = 'Ice Hockey'  
>>> print(favorite\_sports)  
{'Rebecca Clarke': 'Netball', 'Michael Tippett': 'Basketball',   
'Ralph Williams': 'Ice Hockey', 'Edward Elgar': 'Baseball',   
'Frank Bridge': 'Rugby'}

Sustituimos el deporte favorito Fútbol por Hockey sobre hielo utilizando la clave Ralph Williams.

Como ves, trabajar con diccionarios es parecido a trabajar con listas y tuplas, salvo que no puedes unir diccionarios con el operador más ( + ). Si intentas hacerlo, recibirás un mensaje de error, como en el ejemplo siguiente:

>>> favorite\_sports = {'Rebecca Clarke': 'Netball',  
 'Michael Tippett': 'Basketball',  
 'Ralph Williams': 'Ice Hockey',  
 'Edward Elgar': 'Baseball',  
 'Frank Bridge': 'Rugby'}  
>>> favorite\_colors = {'Malcolm Warner' : 'Pink polka dots',  
 'James Baxter' : 'Orange stripes',  
 'Sue Lee' : 'Purple paisley'}  
>>> favorite\_sports + favorite\_colors  
  
Traceback (most recent call last):  
File "<pyshell>", line 1, in <module>  
 favorite\_sports + favorite\_colors  
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'

Unir diccionarios no tiene sentido para Python, así que se da por vencido.

### Lo que has aprendido

En este capítulo has aprendido cómo Python utiliza cadenas para almacenar texto y cómo utiliza listas y tuplas para manejar múltiples elementos. Has visto que los elementos de las listas pueden modificarse, y que puedes unir una lista a otra lista, pero que los valores de una tupla no pueden cambiar. También aprendiste a utilizar diccionarios para almacenar valores con claves que los identifican.

### Rompecabezas de programación

A continuación se presentan algunos experimentos que puedes probar tú mismo. Las respuestas se encuentran en http://python-for-kids.com [*.*](http://python-for-kids.com)

#### #nº 1: Aficiones

Haz una lista de tus aficiones favoritas y dale a la lista el nombre de variable games . Ahora haz una lista de tus comidas favoritas y dale a la variable el nombre foods . Une las dos listas y llama al resultado favorites . Por último, imprime la variable favorites .

#### #nº 2: Contar combatientes

Si hay tres edificios con 25 ninjas escondidos en cada tejado y dos túneles con 40 samuráis escondidos dentro de cada túnel, ¿cuántos ninjas y samuráis están a punto de entrar en combate? (Puedes hacer esto con una ecuación en la Shell de Python).

#### #3: ¡Saludos!

Crea dos variables: una que apunte a tu nombre y otra a tus apellidos. Ahora crea una cadena y utiliza marcadores de posición para imprimir tu nombre con un mensaje utilizando esas dos variables, como "¡Hola, Brando Ickett!".

#### #4: Carta multilínea

Toma la carta que creamos anteriormente en el capítulo e intenta imprimir exactamente el mismo texto utilizando una sola llamada a print (y una cadena multilínea).