

CAPÍTULO 2. VARIABLES.

STRINGS

Un *string* o cadena, es una serie de caracteres. Todo lo que esté entre comillas, **Python** lo considera una cadena y puedes usar comillas simples o dobles alrededor de cadenas como estas:

```
"This is a string."  
'This is also string.'
```

Esta flexibilidad nos permite utilizar comillas y apóstrofes dentro de las cadenas:

```
'I told my friend, "Python is my favorite language!"'  
"The language 'Python' is named after Monty Python, not the snake."  
"One of Python's strengths is its diverse and supportive community."
```

MODIFICANDO CADENAS CON MÉTODOS

Veamos el siguiente ejemplo:

```
name = "ada lovelace"  
print(name.title())
```

El método *title()* aparece después de la variable en la llamada a la función *print()*. El punto después de *name* en ***name.title()*** le dice **Python** que haga que el método *title()* actúe sobre la variable *name*. A cada método lo siguen un par de paréntesis porque los métodos suelen necesitar información adicional para hacer su trabajo y esa información se proporciona dentro de los paréntesis.

Eliminación de espacios en blanco

EJERCICIOS - TRY IT YOURSELF

2-3. Personal message: Utiliza una variable para representar el nombre de una persona e imprime un mensaje a esa persona. Tu mensaje debería ser algo simple como, ¿Qué tal Juan, no tienes cojones de seguir cuando llegas a lo difícil de Python?

2-4. Name cases: Utiliza una variable para representar el nombre de una persona e imprime el nombre de esa persona en mayúsculas, minúsculas y modo `title()` (El primer carácter de cada cadena en mayúscula).

2-5. Famous quote: Encuentra una cita de una persona famosa que admires. Imprime la cita y el nombre de su autor. Tu salida debe parecerse a lo siguiente, incluidas las comillas:

Albert Einstein once said, “A person who never made a mistake never tried anything new.”

2-6. Famous quote 2: Repite el ejercicio 2.5 pero esta vez representa el nombre de una persona famosa utilizando una variable llamada `famous_person` Compón tu mensaje y represéntalo con una nueva variable llamada `message`. Imprime tu mensaje.

2-7. Stripping names: Utiliza una variable para representar el nombre de una persona e incluye caracteres de espacio en blanco al principio y al final del nombre. Asegúrate de utilizas cada combinación de caracteres '`\t`' y '`\n`' al menos una vez. Imprime el nombre una vez de tal modo que se muestren los espacios en blanco alrededor del nombre. Luego imprime el nombre utilizando las tres funciones de stripping `lstrip()`, `rstrip()` y `strip()`.

2-8. Number eigth: Escribe operaciones de suma, resta, multiplicación y división cuyo resultado de siempre 8. Asegúrate de incluir las operaciones en una llamada a `print()` para ver los resultados. Podrías crear cuatro líneas que se parezcan a esta:

```
print(5 + 3)
```

La salida deberían ser simplemente cuatro líneas donde aparezca el número 8 en cada línea.

2-9. Favorite number: Utiliza una variable para representar tu número favorito. Luego, usando esa variable, crea un mensaje que revele tu número favorito. Imprime ese mensaje.

EL ZEN DE PYTHON

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

CAPÍTULO 3. LISTAS.

Una lista es una colección de elementos en un orden en particular. Puedes hacer una lista que incluya las letras del alfabeto, los dígitos del 0 al 9 o los nombres de todas las personas de tu familia. Puedes poner cualquier cosa que quieras en una lista y los elementos que pongas en ella no tienen por qué estar relacionados de una forma en particular. Puesto que una lista contiene normalmente más de un elemento, es una buena idea hacer plural el nombre de tu lista tal como “*cartas*”, “*números*” o “*nombres*”.

En Python los corchetes indican que es una lista y sus elementos individuales están separados por comas. A continuación un simple ejemplo de una lista que contiene tipos de bicicletas:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']

print(bicycles)
```

Si le pides a Python que imprima una lista, Python devolverá la representación de la lista incluyendo los corchetes:

```
['trek', 'cannondale', 'redline', 'specialized']

Process finished with exit code 0
```

Puesto que esta no es la salida que los usuarios quieren ver, aprendamos cómo acceder a los elementos individuales en una lista.

Las listas son colecciones ordenadas, por lo que puedes acceder a cualquier elemento en una lista diciéndole a Python la posición, o índice, o el elemento deseado. Para acceder a un elemento de la lista, escribe el nombre de la lista seguido del indice encerrado entre corchetes.

```
print(bicycles[0])
```

```
trek

Process finished with exit code 0
```

Los índices en una lista comienzan por la posición 0 y no por la posición 1, por lo que hay que tener en cuenta esta circunstancia a la hora de buscar un elemento en una lista. Así pues de la lista anterior, 'trek' estaría en la posición 0, 'cannondale' en la posición uno y así respectivamente.

Si queremos acceder al último elemento de la lista podemos utilizar el índice -1. Supongamos que queremos acceder al elemento 'specialized', pues bien podemos hacerlo de varias formas:

bicycles[-1] ó **bicycles[3]** ← Puesto que en este caso, el último elemento se encuentra en el índice 3 de la lista.

EJERCICIOS – TRY IT YOURSELF

3-1. Names: Almacena el nombre de unos cuantos amigos en una lista llamada 'names'. Imprime el nombre de cada persona accediendo a cada elemento de la lista, uno a la vez.

3-2. Greetings: Comienza con la lista utilizada en el ejercicio 3-1, pero en lugar de imprimir el nombre de cada persona, imprime un mensaje para ellos. El texto de cada mensaje debería ser el mismo pero cada mensaje debería estar personalizado con el nombre de la persona.

3-3. Your own list: Piensa en tu modo favorito de transporte, como una motocicleta o un coche y haz una lista que almacene varios ejemplos. Utiliza tu lista para imprimir una serie de frases acerca de esos elementos como "Me gustaría tener una moto Honda."

Modificar elementos de una lista

La sintaxis para modificar un elemento de una lista es muy parecida a la sintaxis para acceder a un elemento de la lista. Para modificar un elemento, utilizamos el nombre de la lista seguido del índice donde se encuentra el elemento que queremos cambiar y a continuación proporcionamos el valor que queremos proporcionar a ese elemento.

```
motorcycles = ['Honda', 'Suzuki', 'Yamaha']
print(motorcycles)

motorcycles[0] = 'Ducati'
print(motorcycles)
```

Añadir elementos a una lista.

La forma más sencilla de añadir elementos a la lista es con el método `append()`. Este método añade elementos siempre al final de la lista.

El método `append()` facilita la creación dinámica de listas. Por ejemplo, puedes empezar con una lista vacía e ir añadiendo elementos utilizando una serie de llamadas al método `append()`.

Insertar elementos a una lista.

Se pueden añadir nuevos elementos en cualquier posición de la lista simplemente especificando el índice donde queremos insertar el nuevo elemento y su correspondiente valor, como se muestra a continuación:

```
[ 'Honda', 'Suzuki', 'Yamaha', 'Ducati', 'Honda', 'Yamaha', 'Suzuki']
motorcycles.insert(2, 'Husqvarna')
print(motorcycles)
```

```
[ 'Honda', 'Suzuki', 'Husqvarna', 'Yamaha', 'Ducati', 'Honda', 'Yamaha', 'Suzuki']
```

El método `insert()` desplaza cada valor un lugar a la derecha, de tal manera que, como en el ejemplo, el elemento que se encontrara en el índice 2, pasaría ahora a ocupar el índice 3 de la lista y el 2 sería ocupado por el nuevo elemento.

Eliminar elementos de la lista.

Supongamos que durante un juego el jugador ha eliminado un alienígena o un usuario de una web quiere eliminar su perfil de nuestro sitio; son ejemplos prácticos que nos pueden llevar a querer eliminar elementos de una lista (un alien menos, la baja de la web de un usuario, etc).

Se pueden eliminar elementos de una lista atendiendo a su posición en la lista o al valor del elemento.

Si conocemos la posición del elemento que queremos eliminar de la lista podemos utilizar el método `del` en la sentencia en la que deseamos eliminar el elemento:

```
[ 'Honda', 'Suzuki', 'Husqvarna', 'Yamaha', 'Ducati', 'Honda', 'Yamaha', 'Suzuki']
```

```
del motorcycles[5]
```

```
[ 'Honda', 'Suzuki', 'Husqvarna', 'Yamaha', 'Ducati', 'Yamaha', 'Suzuki' ]
```

Como podemos comprobar, la lista antes del método `del` que hemos utilizado para eliminar el elemento '`Honda`' que hay en la posición 5 de la lista, y a continuación se muestra la lista sin el elemento '`Honda`' que se encontraba en el índice 5 de la lista.

Si queremos eliminar el primer elemento de una lista sea cual sea esta sólo tendremos que indicar el índice 0 en el método `del`, por ejemplo:

```
del motorcycles[0].
```

Una vez que eliminemos un elemento de la lista no podremos volver a recuperarlo.

Eliminar un elemento utilizando el método `pop()`.

A veces queremos eliminar el valor de un elemento después de eliminarlo de la lista. Por ejemplo, puedes querer obtener la posición x e y de un alien que acaba de ser abatido, así puedes dibujar una explosión en esa posición. En una aplicación web puedes querer eliminar un usuario de la lista de miembros activos y añadir ese usuario a otra lista de miembros inactivos.

El método `pop()` elimina el último elemento de la lista pero te permite seguir trabajando con él después de eliminarlo. El término `pop` proviene del concepto de una lista como una pila de elementos y sacamos el elemento de la parte superior de la pila. En esta analogía, la parte superior de la pila se corresponde con el final de la lista.

Popeemos una motocicleta de la lista de motocicletas:

```
popped_motorcycle = motorcycles.pop()  
print(motorcycles)  
print(popped_motorcycle)
```

```
['Honda', 'Suzuki', 'Husqvarna', 'Yamaha', 'Ducati', 'Yamaha']  
Como  
Suzuki  
podemos
```

comprobar, hemos almacenado el valor del elemento borrado en la variable *popped_motorcycle* y podemos seguir trabajando con ese valor, como por ejemplo así:

```
print(f'El valor eliminado ha sido la moto {popped_motorcycle}.')  
  
El valor eliminado ha sido la moto Suzuki.
```

Popeando elementos de cualquier posición de la lista.

Puedes utilizar *pop()* para eliminar un elemento desde cualquier posición de la lista incluyendo el índice del elemento que quieras eliminar entre paréntesis.

```
first_owned = motorcycles.pop(0)  
print(f'The first motorcycle i owned was a {first_owned.title()}.')
```

Recuerda que cada vez que utilizas *pop()* el elemento con el que lo hagas no volverá a estar almacenado en la lista.

Si no estás seguro de utilizar la sentencia *del* o el método *pop()*, existe una forma muy sencilla de decidirlo: cuando quieras eliminar un elemento de una lista y no lo vayas a volver a necesitar utiliza la sentencia *del*, si quieres poder volver a utilizarlos después de eliminarlo, utiliza el método *pop()*.

Eliminar un elemento por su valor.

A veces no conocemos la posición del elemento que queremos eliminar de la lista. Si sólo conocemos el valor del elemento que queremos eliminar podemos utilizar el método *remove()*.

Por ejemplo, supongamos que queremos eliminar el valor 'Yamaha' de la lista de motocicletas:

```
print(motorcycles)  
motorcycles.remove('Yamaha')  
print(motorcycles)
```

```
['Suzuki', 'Husqvarna', 'Yamaha', 'Ducati', 'Yamaha']
['Suzuki', 'Husqvarna', 'Ducati', 'Yamaha']
```

El código anterior le dice a Python dónde aparece 'Yamaha' en la lista y elimina ese elemento.

También puedes utilizar el método `remove()` para trabajar con el valor eliminado. Vamos a eliminar 'Ducati' e imprimir la razón por la que hemos decidido eliminarla:

```
too_expensive = 'ducati'
motorcycles.remove(too_expensive)
print(motorcycles)
print(f'La {too_expensive.title()} es demasiado cara para mí.')
```

`remove()` sólo elimina la primera coincidencia del valor indicado. Si hubiera más de un elemento con el mismo valor en la lista necesitarías utilizar un bucle para asegurarte de eliminar todas las coincidencias.

EJERCICIOS – TRY IT YOURSELF.

3-4. Guest List: Si pudieras invitar a alguien, vivo o muerto, a cenar, ¿a quién invitarias? Haz una lista que incluya al menos a tres personas a las que te gustaría invitar a cenar. Luego utiliza la lista para imprimir un mensaje a cada persona invitándola a cenar.

3-5. Changing Guest List: Has oido que uno de tus invitados no puede venir a cenar así que necesitas preparar otro conjunto de invitaciones. Tendrás que pensar en otra persona a quien invitar.

- Empieza con tu programa desde el ejercicio 3-4. Añade una llamada a `print()` al final de tu programa indicando el nombre del invitado que no va a poder asistir.
- Modifica la lista reemplazando el nombre de la persona que no puede asistir por el de la nueva persona que has invitado.
- Imprime un segundo conjunto de mensajes de invitación, uno por cada persona que todavía está en tu lista.

3-6. More Guests: Acabas de encontrar una mesa más grande para cenar, así que ahora dispones de más espacio. Piensa en tres invitados más para invitarlos a cenar.

- Empieza con tu programa desde el ejercicio 3-4. Añade una llamada a `print()` al final de tu programa informando a la gente de que has encontrado una mesa más grande para la cena.
- Utiliza `insert()` para añadir un nuevo invitado al principio de tu lista.
- Utiliza `insert()` para añadir un nuevo invitado en medio de la lista.
- Utiliza `append()` para añadir un nuevo invitado al final de la lista.
- Imprime un nuevo conjunto de invitaciones para cada persona de tu lista.

3-7. Shrinking Guest List: Te acabas de enterar de que tu nueva mesa no estará a tiempo para la cena y que sólo tiene espacio para dos invitados.

- Empieza con tu programa del ejercicio 3-6. Añade una nueva línea que imprima un mensaje diciendo que sólo puedes invitar a dos personas a cenar.
- Utiliza `pop()` para eliminar invitados de tu lista de uno en uno hasta que sólo queden dos nombres en tu lista. Cada vez que pones un nombre de tu lista imprime un mensaje a esa persona haciéndole saber que sientes mucho no poder invitarla a cenar.
- Imprime un mensaje a cada una de las dos personas que aún están en tu lista diciéndoles que permanecen invitadas.
- Utiliza `del` para eliminar los dos últimos nombres de tu lista, así que tienes una lista vacía. Imprime tu lista para asegurarte de que actualmente tienes una lista vacía al final de tu programa.

ORGANIZAR UNA LISTA.

Con frecuencia las listas se crean con un orden impredecible porque no siempre puedes controlar el orden en que tus usuarios te van a proporcionar los datos. A pesar de que esto es inevitable en la mayoría de circunstancias, con frecuencia querrás presentar tu información en un orden concreto. A veces querrás conservar el orden original de tu lista y otras veces querrás cambiar el orden.

Python proporciona diferentes maneras de organizar tus listas dependiendo de la situación.

Ordenar una lista de forma permanente con el método `sort()`.

El método `sort()` de Python hace relativamente fácil ordenar una lista. Imagina que tenemos una lista de coches y queremos cambiar el orden de la lista para almacenarlos alfabéticamente. Para hacer la tarea más simple, asumamos que todos los valores de la lista están en minúsculas.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
cars.sort()
print(cars)
```

```
['audi', 'bmw', 'subaru', 'toyota']

Process finished with exit code 0
```

Ordenar una lista de forma temporal con la función `sorted()`.

Podemos utilizar la función `sorted()` cuando queremos mantener el orden original de una lista pero queremos presentarla de forma ordenada. La función `sorted()` te permite mostrar tu lista en un orden determinado pero que no afecta al orden actual.

Probemos esta función con la lista de coches:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print('\nAqui tenemos la lista original: ')
print(cars)

print('\nY aquí tenemos la lista ordenada:')
print(sorted(cars))
| 
print('\nY aquí la lista original otra vez:')
print(cars)
```

```
Aquí tenemos la lista original:
['bmw', 'audi', 'toyota', 'subaru']
```

```
Y aquí tenemos la lista ordenada:
['audi', 'bmw', 'subaru', 'toyota']
```

```
Y aquí la lista original otra vez:
['bmw', 'audi', 'toyota', 'subaru']
```

```
Process finished with exit code 0
```

Primero imprimimos la lista original ordenada alfabéticamente, después mostramos la lista con el nuevo orden y por último mostramos que la lista aún está almacenada en su orden original.

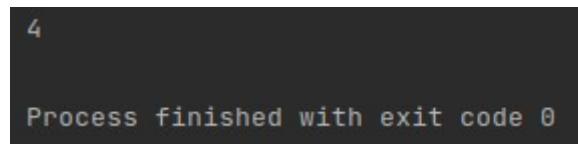
Fíjate que la lista todavía está en su orden original a pesar de haber utilizado la función `sorted()`. La función `sorted()` puede aceptar el argumento `reverse=True` si quieres mostrar la lista ordenada alfabéticamente en orden inverso.

NOTA: Ordenar una lista alfabéticamente es más complicado cuando todos los valores no están en minúsculas. Existen muchas formas de interpretar las letras mayúsculas al determinar un orden de clasificación y especificar el orden exacto puede ser más complicado de lo que queremos en este momento. Sin embargo, la mayoría de los enfoques de ordenación se basarán en lo que hemos aprendido en esta lección.

Encontrar la longitud de una lista.

Puedes encontrar fácilmente la longitud de una lista utilizando la función `len()`. La lista de este ejemplo tiene cuatro elementos por lo que su longitud es 4.

```
print(len(cars))
```



A terminal window showing the output of the code `print(len(cars))`. The output consists of the number 4 on a new line, followed by the text "Process finished with exit code 0".

```
4
Process finished with exit code 0
```

Encontrarás la función `len()` bastante útil cuando necesites identificar el número de aliens que todavía necesitan ser derribados en un juego, determinar la cantidad de datos que tienes

que administrar en una visualización, o averiguar el número de usuarios registrados en un sitio web entre otras tareas.

NOTA: Python cuenta los elementos de una lista empezando por el uno, así que no deberías encontrarte con ningún error off-by-one determinando la longitud de la lista.

EJERCICIOS – TRY IT YOURSELF.

3-8. Seeing the world: Piensa al menos cinco lugares en el mundo que te gustaría visitar:

- Almacena la localización en una lista. Asegúrate de que la lista no está en orden alfabético.
- Imprime la lista en su orden original. No te preocupes por imprimir la lista ordenadamente, simplemente imprimelo como una lista de Python sin procesar.
- Utiliza `sorted()` para imprimir tu lista en orden alfabético sin modificar la lista actual.
- Muestra que tu lista permanece en su orden original imprimiéndola.
- Utiliza `sorted` para imprimir tu lista en orden alfabético inverso sin cambiar la ordenación de la lista original.
- Muestra que tu lista todavía se encuentra en el orden original imprimiéndola.
- Utiliza `reverse()` para cambiar el orden de tu lista. Imprímela para mostrar que el orden ha cambiado.
- Utiliza `reverse()` para cambiar de nuevo el orden de la lista. Imprime la lista para mostrar que ha vuelto a su orden original.
- Utiliza `sort()` para cambiar la lista que está almacenada en orden alfabético. Imprime la lista para mostrar que el orden ha cambiado.
- Utiliza `sort()` para cambiar la lista y almacenarla en orden alfabético inverso. Imprime la lista para mostrar que el orden ha cambiado.

3-9. Dinner Guests: trabajando con uno de los programas desde los ejercicios 3-4 hasta el 3-7(página 42), utiliza `len()` para

imprimir el mensaje indicando la cantidad de gente a la que has invitado a comer.

3-10. Every Function: Piensa en algo que podrías almacenar en una lista. Por ejemplo, podrías hacer una lista de montañas, ríos, países, ciudades, idiomas, o cualquier cosa que te guste. Escribe un programa que genere una lista que contenga esos elementos y luego usa cada función mostrada en este capítulo al menos una vez.

Funciones utilizadas:

```
print()  
indice[]  
title()  
indice[-1] - Último elemento  
indice[i] = 'loquesea' - Modificar elementos  
list.append() - Añadir al final  
list.insert(posicion, 'elemento')  
del list[i] - Eliminar elemento de la lista en la posición i  
lista.pop() - Eliminar usando el valor eliminado  
list.remove('nombreDeElemento')  
list.sort() - ordenar permanentemente  
reverse=True - ordenar a la inversa  
sorted(list) - ordenar temporalmente  
list.reverse() - ordenar a la inversa permanentemente  
len(list) - longitud de la lista
```

Evitar errores de índices al trabajar con listas.

Es uno de los errores más comunes cuando se trabaja con listas. Supongamos que tienes una lista de tres elementos y preguntamos por el cuarto.

Tendríamos un *index error* o error de índice.

Si tienes un index error intenta ajustar el índice por el que estás preguntando. Ejecuta de nuevo el programa y para ver que los resultados son correctos.

Ten en cuenta que siempre que quieras acceder al último elemento de la lista hazlo usando índice -1. Esto siempre funciona, incluso si tu lista ha cambiado de tamaño desde la última vez que accediste a ella. La única vez que te puede dar error utilizar -1 es si lo utilizas con una lista vacía: *list[]*.

NOTA. Si se produce un *index error* y no sabes cómo resolverlo, intenta imprimir tu lista o la longitud de la lista. La lista puede parecerse más diferente de lo que pensabas, especialmente si ha sido configurada dinámicamente por tu programa. Comprobar la lista actual o el número exacto de elementos de tu lista puede ayudarte a resolver muchos problemas de lógica.

EJERCICIOS – TRY IT YOURSELF.

3-11. Intentional Error: Si todavía no has recibido un *index error* en tu programa, intenta provocarlo. Cambia un índice en uno de tus programas para que produzca un error. Asegúrate de corregir el error antes de cerrar tu programa.

CAPÍTULO 4 . TRABAJAR CON LISTAS.

Aprenderemos hacer un bucle para recorrer todos los elementos de la lista con apenas unas pocas líneas de código.

Querrás recorrer a menudo todas las entradas de la lista para realizar la misma tarea con cada uno de sus elementos, por ejemplo cuando en un juego quieras mover todos los elementos en la pantalla o mostrar la cabecera de una lista de artículos de en un sitio web.

Supongamos que tenemos una lista de nombres de magos y queremos imprimir cada nombre de la lista. Podríamos mostrar cada nombre de forma individual pero este comportamiento podría causarnos serios problemas.

Usemos un bucle para imprimir cada nombre de la lista de magos.

```
magicians = ['alice', 'david', 'carolina']

for magician in magicians:
    print(magician.title())
```

Este bucle *for* le dice a Python que extraiga un nombre de la lista y lo asocie a la variable *magician*. Ya dentro del bucle, la sentencia *print* imprime el valor que tiene asociado en ese momento la variable *magician*. Python repite esta sentencia para cada nombre de la lista. Podría ayudarnos a leer este código como: “Para cada mago de la lista de magos, imprime el nombre de mago.” La salida es una simple impresión del nombre de cada mago:

```
Alice  
David  
Carolina  
  
Process finished with exit code 0
```

Mirando el bucle de cerca.

El concepto de bucle es importante porque es una de las formas más comunes que tiene una computadora de automatizar tareas repetitivas. Python inicialmente lee la primera línea del bucle:

```
for magician in magicians:
```

Esta línea le dice a Python que recupere el primer valor de la lista de magos y lo asocie a la variable *magician*. El primer valor es '**alice**'. Entonces Python lee la siguiente línea:

```
print(magician.title())
```

Python imprime el valor actual de *magician*, que aún es '**alice**'. Puesto que la lista contiene más valores, Python devuelve la primera línea del bucle:

```
for magician in magicians:
```

Python recupera el siguiente nombre de la lista, '**david**' y asocia ese valor con la variable *magician*. Python entonces, ejecuta la linea:

```
print(magician.title())
```

Python imprime de nuevo el valor actual de *magician*, que ahora es '**david**'. Python repite el bucle completo una vez más con el último valor de la lista, '**carolina**'. Puesto que no hay más valores en la lista, Python se desplaza a la siguiente línea del programa. En este caso no hay nada después del bucle por lo que el programa sencillamente finaliza.

Cuando está utilizando bucles por primera vez, recuerda que cada paso se repite una vez por cada elemento de la lista sin importar cuantos elementos haya. Si tienes un millón de elementos en la lista, Python repetirá esos pasos un millón de veces, y normalmente muy rápido.

También recuerda cuando escribas tu propio bucle, que puedes elegir el nombre que quieras para la variable temporal que será asociada a cada valor de la lista. Sin embargo, ayuda mucho escoger un nombre significativo que represente a cada elemento de la lista. Por ejemplo, aquí tienes una buena manera de comenzar un bucle para una lista de gatos, de perros y una lista de elementos en general:

```
for cat in cats:  
    for dog in dogs:  
        for item in list_of_items:  
            ~
```

Estas convenciones de nomenclatura te pueden ayudar a seguir la acción que está realizando cada elemento dentro de un bucle for.

Utilizar una lista de números.

Si quieres hacer una lista de números, puedes convertir el resultado de `range()` directamente en una lista utilizando la función `list[]`.

Cuando encapsulas `list[]` en una llamada a la función `range()`, la salida que obtienes es una lista de números.

```
# Utilizar una lista de números.  
numbers = list(range(1, 6))  
print(numbers)
```

También podemos utilizar la función `range()` para decirle a Python que salte números en un rango dado. Si le pasamos un tercer argumento a `range()`, Python utilizará ese valor como tamaño de paso cuando genere los números.

Puedes crear casi cualquier conjunto de números que quieras utilizando la función `range()`.

Por ejemplo, vamos a considerar calcular los cuadrados de los 10 primeros números.

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

print(squares)
```

Estadísticas simples con listas de números.

Unas pocas funciones de Python nos ayudarán cuando trabajemos con listas de números. Por ejemplo, puedes encontrar fácilmente el mínimo, máximo y la suma de una lista de números.

List Comprehension.

Una *list comprehension* te permite generar la misma lista de cuadrados del ejemplo anterior con una sola línea de código. Una *list comprehension* combina el bucle *for* y la creación de nuevos elementos en una línea y automáticamente añadir cada nuevo elemento. Las *lists comprehension* no son para principiantes pero se incluyen aquí porque suelen aparecer en el código de otros programadores más experimentados.

El siguiente ejemplo construye la misma lista de cuadrados de números que hemos visto antes, pero utilizando *list comprehension*.

```
# List comprehension
squares2 = [value**2 for value in range(1, 11)]
print(f'List comprehension {squares2}')
```

Para utilizar esta sintaxis comenzamos con un nombre descriptivo para la lista, como *squares*. A continuación abrimos un conjunto de corchetes y definimos 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 segunda potencia. Luego escribe un bucle *for* para generar los números que quieras introducir en la expresión y cierra los corchetes.

Practica para escribir tus propias *list comprehensions*, y encontrarás que vale la pena una vez te sientas cómodo creándolas. Cuando estés escribiendo tres o cuatro líneas de código para

generar listas y comience a ser repetitivo valora escribir tu propia list comprehension.

EJERCICIOS – TRY IT YOURSELF.

4-3. Counting to Twenty: Utiliza un bucle para imprimir los números del 1 al 20 ambos inclusive.

4-4. One Million: Haz una lista de números de uno a un millón y luego utiliza un bucle para imprimir los números. (Si la salida es demasiado larga, páusala presionando *Ctrl + C* o cerrando la ventana.

4-5. Summing a Milion: Haz una lista de números desde uno a un millón y luego usa las funciones `min()` y `max()` para asegurarte actualmente empieza por uno y acaba por un millón. También utiliza la función `sum()` para ver cómo de rápido suma Python un millón de números.

4-6. Odd Numbers: Utiliza el tercer argumento de la función `range()` para hacer una lista de números impares desde el 1 al 20. Utiliza un bucle para imprimir cada número.

4-7. Threes: Haz una lista de múltiplos de 3 desde el 3 hasta el 30. Utiliza un bucle para imprimir los números de tu lista.

4-8.Cubes: Un número elevado a la tercera potencia se llama cubo. Por ejemplo, el cubo de 2 se escribe como `2**3` en Python. Haz una lista de los primeros 10 cubos (eso es, el cubo de cada entero desde 1 hasta 10) y utiliza un bucle para imprimir el valor de cada cubo.

4-9. Cube Comprehension: Utiliza una list comprehension para generar una lista de los diez primeros cubos.

Trabajar con partes de una lista.

En el capítulo 3 aprendimos a acceder elementos individuales en una lista y ahora aprenderemos como trabajar con todos los elementos de la lista. También puedes trabajar con un grupo específico de elementos de la lista que Python llama *slice*.

Devanar una lista.

Para hacer una devanado tienes que especificar el primer y el último elemento con los que quieres trabajar. Como con la función `range()`, Python se detiene un elemento antes del segundo índice que especifiquemos, es decir, si introducimos el rango `lista[1:5]`,

podremos trabajar con los elementos que se encuentren en las posiciones 1, 2, 3 y 4.

Si se omite el primer índice del devanado, Python comenzará a trabajar desde el principio de la lista.

Una sintaxis similar funciona si quieras un devanado que incluya el final de la lista. Por ejemplo, si quieras todos los elementos de la lista desde el índice 2 hasta el final de la lista, basta con omitir el segundo índice en el devanado como se muestra a continuación: `lista[2:]`

Esta sintaxis te permite generar todos los elementos desde cualquier punto de la lista, sin importar la longitud de esta. Un índice negativo devuelve un elemento a cierta distancia desde el final de la lista, por lo tanto puedes generar cualquier devanado desde el final de la lista. Por ejemplo, si queremos mostrar los tres últimos jugadores de la lista podemos utilizar el siguiente devanado: `lista[-3:]`

```
print(players[-3:])
```

```
['michael', 'florence', 'eli']
Process finished with exit code 0
```

Esto imprimiría los nombres de los tres últimos jugadores y podría continuar funcionando a medida que la lista de jugadores cambie de tamaño.

NOTA: Puedes incluir un tercer valor en los corchetes. Si lo haces, le dirás a Python cuántos elementos saltar entre los elementos especificados en el rango.

Bucle a través del devanado.

Puedes utilizar el devanado en un bucle `for` si quieras recorrer un subconjunto de elementos en una lista. En el siguiente ejemplo recorreremos los tres primeros jugadores e imprimiremos sus nombres como parte de una lista más pequeña.

```
print('\nAquí están los tres primeros jugadores de mi equipo: \n')
for pl in team[:3]:
    print(pl)

Charles
Martina
Michael

Process finished with exit code 0
```

Los devanados son muy útiles en situaciones con números. Por ejemplo, cuando estás creando un juego puedes añadir la puntuación final de un jugador a una lista cada vez que el jugador termine de jugar. Puedes obtener los mejores tres marcadores y ordenarlos en una lista en orden decreciente y obtener un devanado que incluya sólo las tres mejores puntuaciones. Cuando estás trabajando con datos puedes utilizar *slices* para procesar tus datos en porciones de un tamaño específico. O cuando estás construyendo una aplicación web, puedes utilizar *slices* para mostrar información de una serie de páginas con una cantidad apropiada de información en cada página.

Copiar una lista.

Con frecuencia querrás empezar con una lista existente y hacer una lista completamente nueva basada en la primera. Exploraremos como funciona el copiado de listas y examinemos una situación en la que copiar una lista es útil.

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

Por ejemplo, imagina que tenemos una lista de nuestras comidas favoritas y queremos hacer una lista separada 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:

```
my_foods = ['pizza', 'falafel', 'carrot cake']
friend_foods = my_foods[:]

print('My favorite foods are: ')
print(my_foods)

print("\nMy friend's favorite foods are: ")
print(friend_foods)
```

```
My favorite foods are:  
['pizza', 'falafel', 'carot cake']  
  
My friend's favorite foods are:  
['pizza', 'falafel', 'carot cake']  
  
Process finished with exit code 0
```

Para probar que tenemos dos listas separadas, añadiremos una nueva comida a cada lista y mostraremos que cada lista contiene las comidas favoritas de las personas correspondientes.

```
my_foods.append('cannoli')  
friend_foods.append('ice cream')  
  
print('\nMy favorite foods are: ')  
print(my_foods)  
  
print("\nMy friend's favorite foods are: ")  
print(friend_foods)
```

```
My favorite foods are:  
['pizza', 'falafel', 'carot cake', 'cannoli']  
  
My friend's favorite foods are:  
['pizza', 'falafel', 'carot cake', 'ice cream']  
  
Process finished with exit code 0
```

Si simplemente hubiéramos establecido friend_foods como my_foods, no produciría dos listas separadas. Por ejemplo aquí está lo que sucedería si intentamos copiar una lista sin utilizar el devanado:

```
# Esto no funciona como queremos ahora mismo
friend_foods = my_foods

my_foods.append('cannoli')
friend_foods.append('ice cream')

print('\nMy favorite foods are: ')
print(my_foods)
```

```
My favorite foods are:
['pizza', 'falafel', 'carot cake', 'cannoli', 'cannoli', 'ice cream']

Process finished with exit code 0
```

Como podemos comprobar en la salida, en lugar de almacenar una copia de `my_foods` en `friend_foods`, establecemos que `friend_foods` es igual `my_foods`. Esta sintaxis le dice a Python que asocie la nueva variable `friend_foods` con la lista que ya está asociada a `my_foods`, por lo que ahora ambas variables apuntan a la misma lista. Como resultado, cuando añadimos '`cannoli`' a `my_foods`, también aparecerá en `friend_foods`. Igualmente, '`ice cream`' aparecerá en ambas listas, a pesar de que parece que sólo se añade a `friend_foods`.

La salida muestra que ambas listas son la misma lista, y no es lo que estábamos buscando.

NOTA: No te preocupes por los detalles de este ejemplo por ahora. Básicamente si estás intentando trabajar con una copia de una lista y observas un comportamiento inesperado, asegúrate de que estás copiando la lista con un slice, tal y como hicimos en el primer ejemplo.

EJERCICIOS – TRY IT YOURSELF.

4-10. Slices: Utilizando uno de los programas escritos en este capítulo, añade más líneas al final del programa que hagan lo siguiente:

- Imprime el mensaje '*Los primeros tres elementos de la lista son:* '. Luego utiliza un slice para imprimir los tres primeros elementos de la lista de ese programa.
- Imprime el mensaje '*Los elementos que se encuentran en medio de la lista son:* '. Luego utiliza un slice para imprimir los elementos que se encuentran en medio de la lista de ese programa.
- Imprime el mensaje '*Los últimos tres elementos de la lista son:* '. Luego utiliza un slice para imprimir los tres últimos elementos de la lista de ese programa.

4-11. My Pizzas, Your Pizzas: Empieza con tu programa del ejercicio 4-1. Haz una copia de la lista de pizzas a llámala friend_pizzas. Luego haz lo siguiente:

- Añade una nueva pizza a la lista original.
- Añade una pizza diferente a la lista friend_pizzas.
- Comprueba que tienes dos listas separadas. Imprime el mensaje '*Mis pizzas favoritas son:* ', y luego utiliza un bucle *for* para imprimir la primera lista. Imprime el mensaje '*Las pizzas favoritas de mi amigo son:* ' y luego utiliza un bucle *for* para imprimir la segunda lista. Asegúrate de que cada nueva pizza se almacena en la lista apropiada.

4-12. More Loops: todas las versiones de foods.py en esta sección ha evitado utilizar bucles *for* para ahorrar espacio. Escoge una versión de foods.py y escribe dos bucles *for* para imprimir cada lista de alimentos.

Tuplas.

Las lista trabajan bien para almacenar colecciones de elementos que pueden cambiar durante la vida de nuestro programa. La capacidad para modificar listas es particularmente importante cuando estás trabajando con una lista de usuarios en un sitio web o una lista de personajes en un juego. Sin embargo, a veces querremos crear una lista de elementos que no puedan cambiar. Las

tuplas te permiten hacer justo eso. Python se refiere a los valores que no pueden cambiar como inmutables y una lista inmutable se llama tupla.

Definir una tupla.

Una tupla se parece a una lista salvo porque usa paréntesis en lugar de corchetes. Una ve que defines una tupla, puedes acceder a los elementos individualmente utilizando el índice de cada uno de ellos, como lo harías con una lista.

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

```
# Dimensiones de un rectángulo
dimensions = (200, 50)
print(dimensions[0])
print(dimensions[1])
```

Utilizamos la misma sintaxis que hemos estado utilizando para acceder a los elementos en una lista para imprimir cada elemento de la tupla individualmente:

```
200
50

Process finished with exit code 0
```

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

```
dimensions[0] = 250
```

El código intenta cambiar el valor de la primera posición pero Python devuelve un *Type Error*. Básicamente porque estamos intentando cambiar el valor a una tupla y no se puede hacer con ese tipo de objetos. Python nos dice que no podemos asignar un nuevo valor a un elemento en una tupla:

```
Traceback (most recent call last):
  File "/home/usuario/PycharmProjects/pythoAFondo/capitulo_4/tuplas/dimensions.py", line 6, in <module>
    dimensions[0] = 250
TypeError: 'tuple' object does not support item assignment
```

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

NOTA: Las tuplas se definen técnicamente por la presencia de una coma; los paréntesis hacen que sea más clara y legible. Si quieres definir una tupla con un sólo elemento, necesitas incluir una coma final. No suele tener mucho sentido construir una tupla con un sólo elementos, pero esto puede suceder cuando las tuplas se genera automáticamente.

Recorrer todos los valores de una tupla.

Puedes recorrer todos los valores de una tupla utilizando un bucle for, como hiciste con la lista:

```
for dimension in dimensions:  
    print(dimension)
```

Python devuelve todos los elementos de una tupla tal como lo haría con una lista:

```
200  
50  
  
Process finished with exit code 0
```

Escribiendo sobre una tupla.

A pesar de que una tupla no se puede modificar, puedes asignar un nuevo valor a una variable que represente a una tupla. Así que si queremos cambiar nuestras dimensiones podemos redefinir toda la tupla:

```
print('Dimensiones originales: ')  
for dimension in dimensions:  
    print(dimension)  
  
dimensions = (400, 100)  
print('\nDimensiones modificadas: ')  
for dimension in dimensions:  
    print(dimension)
```

Asociamos una nueva tupla con la variable `dimensions` e imprimimos las nuevas dimensiones. Python no genera ningún error esta vez porque reasignar una variable es válido:

```
Dimensiones originales:  
200  
50  
  
Dimensiones modificadas:  
400  
100  
  
Process finished with exit code 0
```

En comparación con las listas, las tuplas son estructuras de datos simples. Úsalas cuando quieras almacenar un conjunto de valores que podría no ser cambiado durante la vida de un programa.

EJERCICIOS – TRY IT YOURSELF.

4-13. Buffet: Un restaurante estilo buffet sólo tiene cinco comidas básicas. Piensa en cinco comidas simples y almacénalas en una tupla.

- Utiliza un bucle para imprimir cada comida que ofrece el restaurante.
- Intenta modificar uno de los elementos, y asegúrate de que Python rechace el cambio.
- El restaurante cambia su menú, reemplaza dos elementos con comida diferente. Añade una línea que reescriba la tupla y utiliza un bucle `for` para imprimir cada uno de los elementos del menú revisado.

Estilo de tu código.

Ahora que estás escribiendo programas más largos, vale la pena conocer ideas sobre cómo diseñar tu código.

Tómate tiempo para hacer tu código tan fácil de leer como sea posible. Escribir código fácil de leer (Easy-to-read) te ayuda a realizar un seguimiento de lo que hace tu programa y ayuda a otros a entender tu código también.

Los programadores de Python han acordado una serie de convenciones de estilo para garantizar que el código de todos está estructurado aproximadamente de la misma manera. Una vez que has aprendido a escribir código Python limpio, deberías ser capaz de entender la estructura general del código Python de cualquier otra persona, siempre y cuando sigan las mismas pautas. Si esperas convertirte en un programador profesional en algún momento, deberías seguir estas pautas tan pronto como sea posible para desarrollar buenos hábitos.

La guía de estilo.

Cuando alguien quiere hacer un cambio en el lenguaje Python, escriben un **PEP** (***Python Enhancement Proposal***) o propuesta de mejora de Python. Uno de los PEP's más antiguos es **PEP 8**, que instruye a los programadores de Python sobre cómo diseñar su código. **PEP 8** es bastante largo, pero se relaciona con estructuras de código más complejas de lo que has visto hasta ahora.

La guía de estilo de Python fue escrita con el entendimiento de 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 características a tu programa dedicarás más tiempo a leer tu código. Cuando compartas tu código con otros programadores, ellos también lo leerán.

Dada la elección entre escribir código que sea más fácil de escribir o código que sea más fácil de leer, los programadores de Python casi siempre te alentarán a escribir código que sea más fácil de leer. Las siguientes pautas te ayudarán a escribir código claro desde el principio.

Indentación.

PEP 8 recomienda que uses cuatro espacios de indentación por nivel. Utilizando cuatro espacios mejoras la legibilidad y dejas espacio para múltiples niveles de indentación en cada línea.

En un documento de un procesador de texto, las personas usan el tabulador en lugar del espacio para indentar. Esto funciona bien para documentos de procesamiento de textos, pero el intérprete de Python se puede confundir cuando se mezclen espacios con tabulaciones. Cada editor de textos proporciona unas herramientas que te permiten utilizar la tecla TAB pero entonces cada tabulación se convertiría en un número determinado de espacios. Puedes definir un uso para tu tecla TAB, pero asegúrate de que tu editor esté configurado para introducir espacios en lugar de tabulaciones en tu documento.

Mezclar tabulaciones y espacios en un fichero puede causar problemas que sean difíciles de detectar. Si piensas que tienes una mezcla de tabulaciones y espacios en un fichero, puedes convertir todas las tabulaciones del fichero en espacios en la mayoría de los editores.

Longitud de línea.

Muchos programadores de Python recomiendan que cada línea sea inferior a 80 caracteres. Históricamente, esta guía se desarrolló porque la mayoría de las computadoras sólo podía incluir 79 caracteres en una sola línea en una ventana de terminal. En la actualidad, las personas pueden incluir líneas mucho más largas en sus pantallas, pero existen otras razones para adherirse al estándar de 79 caracteres de longitud de línea. Los programadores profesionales suelen tener muchos ficheros abiertos en la misma pantalla y utilizando el estándar de longitud de línea les permite ver las líneas completas en dos o tres archivos que estén abiertos uno al lado del otro. PEP 8 también recomienda limitar todos los comentarios a 72 caracteres por línea porque muchas de las herramientas que generan documentación automática para grandes proyectos añaden formato a los caracteres al principio de cada línea comentada.

Las pautas de PEP 8 para la longitud de línea no son inamovibles y algunos equipos prefieren limitar los caracteres a 99. No te

preocupes demasiado por la longitud de líneas de tu código mientras estás aprendiendo, pero ten en cuenta que las personas que trabajan de forma colaborativa siempre siguen las pautas del PEP 8. La mayoría de editores te permiten configurar una señal visual, normalmente una línea vertical en la pantalla, que te muestra dónde están esos límites.

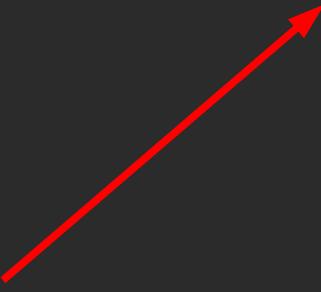
```
menu3 = [print(menu) for menu in menus]
menu3

# 2. Intenta modificar uno de los elementos, y asegúrate de que Python rechace el cambio.
#menus[3] = 'dorayaki'

# 3. El restaurante cambia su menú, reemplaza dos elementos con comida diferente.
# Añade una línea que reescriba la tupla y utiliza un bucle for para imprimir cada uno
# de los elementos del menú revisado.
menus = ('entrantes', 'fideuá', 'tepenyaki', 'dorayaki', 'nigri sushi')
for menu in menus:
    print(menu)

# Apartado 3 con list comprehension
nuevo_menu = [menu for menu in menus]
print(nuevo_menu)

nuevo_menu2 = [print(menu) for menu in menus]
nuevo_menu2
```



NOTA: El apéndice B te muestra cómo configurar tu editor de texto para que siempre introduzca cuatro espacios cada vez que pulses la tecla TAB y te muestre una guía vertical para ayudarte a seguir el límite de 79 caracteres.

Líneas en blanco.

Para agrupar visualmente partes de tu programa, utiliza líneas en blanco. Deberías usar líneas en blanco para organizar tus ficheros, pero no abuses. Siguiendo los ejemplos provistos en este libro debes lograr un equilibrio correcto. Por ejemplo, si tienes cinco líneas de código que construyen una lista y luego otras tres líneas que hacen algo con esa lista, se considera apropiado dejar una línea en blanco entre las dos secciones. Sin embargo, no deberías colocar tres o cuatro líneas en blanco entre las dos secciones.

Las líneas en blanco no afectan al comportamiento de tu código, pero sí a la legibilidad. El intérprete de Python utiliza indentación horizontal para interpretar el significado de tu código, pero no tiene en cuenta el espacioado vertical.

Otras pautas de estilo.

PEP 8 tiene muchas recomendaciones de estilo adicionales, pero la mayoría de las guías hacen referencia a programas más complejos de los que estás escribiendo en este momento. A medida que aprendas estructuras de Python más complejas, compartiré las partes importantes de las directrices PEP 8.

EJERCICIOS – TRY IT YOURSELF.

4-14. PEP 8: Consulta la guía de estilo original PEP 8 en '<https://python.org/dev/peps/pep-0008/>' No utilizarás mucho de ella ahora mismo, pero puede ser interesante ojearla.

4-15. Code Review: Escoge tres de los programas que has escrito en este capítulo y modificalos para cumplir con el PEP 8.

CAPÍTULO 5. SENTENCIAS IF.

Las sentencias if te permiten examinar el estado actual de un programa y responder apropiadamente a ese estado. Aplicaremos estos conceptos a las listas, por lo que escribiremos un bucle que maneje la mayoría de los elementos de una lista de una manera determinada, pero otros elementos con valores específicos de una manera diferente.

Un ejemplo sencillo.

Este ejemplo corto muestra cómo las pruebas con if te permiten responder a situaciones especiales correctamente. Imaginemos una lista de coches y queremos imprimir el nombre de cada coche. Los nombres de los coches son nombres propios por lo que la mayoría de los coches se suelen imprimir con la primera letra en mayúscula. Sin embargo, el valor 'bmw' podríamos imprimirla con todas las letras en mayúsculas. El siguiente código recorre una lista de nombres de automóviles y busca el valor 'bmw'. Siempre que el valor sea 'bmw', se imprime con todas las letras en mayúsculas, en lugar de poner sólo en mayúsculas la primera letra.

```
cars = ['audi', 'bmw', 'subaru', 'toyota']
for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

```
Audi
BMW
Subaru
Toyota

Process finished with exit code 0
```

El bucle de este ejemplo, primero comprueba si el valor actual de car es 'bmw'. Si lo es, lo imprime en *uppercase*. Si el valor de car es cualquier otro que no sea 'bmw', se imprime en *title case*.

Pruebas condicionales.

En el corazón de cada sentencia *if*, hay una expresión que puede ser evaluada como *True* o *False* y se llama Prueba Condicional (*Conditional Test*). Python utiliza los valores *True* y *False* para decidir si se debe ejecutar el código de una instrucción *if*. Si la prueba se evalúa como *False*, Python ignora el código a continuación de la sentencia *if*.

Ignorar las mayúsculas cuando se comprueba la igualdad.

Probemos la igualdad en mayúsculas/minúsculas en Python. Por ejemplo, dos valores con diferente capitalización no se consideran iguales:

```
car = 'Audi'
print(car == 'audi')
```

```
False

Process finished with exit code 0
```

Si necesitas comparar mayúsculas y minúsculas, este comportamiento es ventajoso para nosotros, pero si no necesitamos o no queremos tener en cuenta la capitalización de las letras, podemos convertir

los valores de las variables a minúsculas antes de realizar la comparación.

```
car = 'Audi'  
print(car.lower() == 'audi')
```

```
True  
  
Process finished with exit code 0
```

Esta prueba devolvería True sin importar cómo esté formateado el valor 'Audi' porque la prueba no distingue entre mayúsculas y minúsculas. La función `lower()` no cambia el valor original almacenado en `car`, por lo que puedes hacer cualquier tipo de comparación sin que afecte a la variable original.

```
car = 'Audi'  
print(car.lower() == 'audi')  
print(car)
```

```
True  
Audi  
  
Process finished with exit code 0
```

Los sitios web aplican ciertas reglas para los datos que los usuarios introducen de manera similar a esta. Por ejemplo, un sitio puede utilizar una prueba condicional como esta para asegurarse de que todos los usuarios tiene realmente un nombre único y que no sólo varíe la capitalización del nombre de usuario de otra persona. Cuando alguien envía un nuevo nombre de usuario, ese nuevo nombre de usuario se convierte a minúsculas y se compara con las versiones en minúscula de todos los nombres de usuario. Durante la comprobación un nombre de usuario como 'John' será rechazado si cualquier variación de 'john' ya estuviera en uso.

Comprobar la no igualdad.

Cuando queremos determinar si dos valores no son iguales, podemos combinar un signo de exclamación y un signo igual (`!=`). El signo de exclamación representa no, como en mucho lenguajes de programación.

Utilicemos otra sentencia `if` para examinar cómo utilizar el operador de no igualdad. Almacenaremos los ingredientes de la

pizza que hemos pedido y luego imprimiremos un mensaje si la persona no pidió anchoas.

```
requested_toppings = 'mushrooms'

if requested_toppings != 'anchovies':
    print('No pongas anchovies!!')
```

```
No pongas anchovies!!

Process finished with exit code 0
```

La mayoría de las expresiones condicionales que escribamos tratarán de probar la igualdad, pero a veces resultará más eficiente probar la no igualdad.

Comparaciones numéricas.

Probar valores numéricos es bastante sencillo. Por ejemplo, el siguiente código comprueba si una persona es mayor de 18 años:

```
edad = 18
print(edad == 18)
```

```
True

Process finished with exit code 0
```

También puedes comprobar si dos números no son iguales. Por ejemplo, el siguiente código imprime un mensaje si la respuesta dada no es correcta:

```
answer = 17

if answer != 42:
    print('Esa no es la respuesta correcta. Por favor, inténtalo de nuevo.'
```

```
Esa no es la respuesta correcta. Por favor, inténtalo de nuevo.
```

Puedes

```
Process finished with exit code 0
```

incluir varias comparaciones matemáticas en tu sentencia condicional como menor que `<`, menor o igual que `<=`, mayor que `>`, mayor o igual que `>=`, etc.

Cada comparación matemática se puede utilizar como parte de una sentencia *if* que puede ayudarte a detectar las condiciones exactas que te interesen.

Comprobar condiciones múltiples.

Podrías querer comprobar múltiples condiciones al mismo tiempo. Por ejemplo, a veces podrías necesitar que dos condiciones sean verdaderas para tomar una decisión, otras veces podría ser suficiente con que sólo una de las condiciones lo sea. Las palabras clave *and* y *or* pueden ayudarte en estos casos.

Utilizar y comprobar condiciones múltiples.

Para comprobar que dos condiciones son verdaderas simultáneamente utilizamos la palabra clave *and*, combinada con dos pruebas condicionales. Si alguna de las pruebas falla o lo hacen ambas la expresión se evaluará como *False*.

Por ejemplo, quieras comprobar si dos personas son mayores de 21 años utilizando el siguiente test:

```
edad_0 = 22  
edad_1 = 18  
  
print(edad_0 >= 21 and edad_1 >= 21)
```

```
False
```

```
Process finished with exit code 0
```

Para mejorar la legibilidad se pueden usar paréntesis alrededor de las pruebas individuales, pero no son obligatorios. En caso de usarlos las pruebas quedarían así:

```
(edad_0 >= 21) and (edad_1 >= 21)
```

Uso o comprobación de condiciones múltiples.

La palabra clave *or* también te permite comprobar condiciones múltiples, pero pasa cuando alguna o ambas pruebas individuales pasan. Un expresión *or* falla sólo cuando ambas pruebas individuales fallan. Es decir, *False or True == True*, *True or False == True*, *True or True == True* y *False or False == False*.

Consideremos de nuevo dos edades, pero esta vez busquemos que sólo una de ellas sea mayor de 21:

```
edad_0 = 22  
edad_1 = 18  
  
print(edad_0 >= 21 or edad_1 >= 21)
```

```
True
```

```
Process finished with exit code 0
```

El siguiente ejemplo, ambas condiciones son falsas:

```
edad_0 = 18
print(edad_0 >= 21 or edad_1 >= 21)
```

```
False
```

```
Process finished with exit code 0
```

Comprobar si un valor está en una lista.

A veces es necesario comprobar si una lista contiene determinado valor antes de realizar cualquier acción. Por ejemplo, puede querer comprobar si un nombre de usuario ya existe en la lista actual de nombres de usuario antes de completar el registro de una persona en un sitio web. En un proyecto de mapeado, puedes querer comprobar si la localización enviada ya existe en la lista de localizaciones conocidas.

Para encontrar si un valor en concreto existe en una lista, utilizaremos la palabra clave *in*. Consideremos algún código que puedas escribir para una pizzería. Haremos una lista de ingredientes pedidos por un cliente para una pizza y luego comprobaremos si determinados ingredientes están en la lista.

```
requested_toppings_2 = ['mushrooms', 'onions', 'pineapple']
print('mushrooms' in requested_toppings_2)
print('pepperoni' in requested_toppings_2)
```

```
True
```

```
False
```

```
Process finished with exit code 0
```

Comprobar si un valor No está en una lista.

Otras veces es importante saber si un valor no aparece en una lista. Puedes utilizar la palabra clave *not in* en esta situación. Por ejemplo, consideremos una lista de usuarios que tienen

prohibido comentar en un foro. Puedes comprobar si un usuario ha sido baneado antes de permitir que esa persona envíe un comentario.

```
banned_users = ['andrew', 'carolina', 'david']
user = 'marie'

if user not in banned_users:
    print(f'{user.title()}, puedes publicar una respuesta si lo deseas.')
```

```
Marie, puedes publicar una respuesta si lo deseas.

Process finished with exit code 0
```

Expresiones booleanas.

A medida que aprendas más acerca de la programación, escucharás el término '*expresiones booleanas*' en algún momento. Una expresión booleana es sólo otro nombre para probar los condicionales. Un valor booleano puede ser True o False, al igual que el valor de una expresión condicional después de ser evaluada.

Los valores booleanos se suelen utilizar para hacer un seguimiento de determinadas condiciones tales como que un juego se esté ejecutando o que un usuario pueda editar cierto contenido en un sitio web:

```
game_active = True
can_edit = False
```

Los valores booleanos te proporcionan una forma eficiente de hacer un seguimiento del estado de un programa o de una condición en particular que es importante en tu programa.

EJERCICIOS – TRY IT YOURSELF.

5-1. Conditional Tests: Escribe una serie de test condicionales. Imprime un frase describiendo cada test y tu predicción para el resultado de cada test. Tu código debería parecerse a este:

- Mira de cerca tus resultados y asegúrate de que comprendes por qué cada línea se evalúa a True o False.
- Crea al menos diez test que tengan al menos cinco pruebas evaluadas a True y otras cinco a False.

5-2. More Conditional Tests: No tienes que limitar el número de test que has creado a sólo diez. Si quieres probar más comparaciones escribe más test y añádelos a *conditional_tests.py*. Ten al menos un resultado True y otro False para cada uno de los siguientes:

- Un test de igualdad y no igualdad con cadenas.
- Un test utilizando el método *lower()*.
- Test numéricos que incluyan igualdad y no igualdad, mayor que, mayor o igual que y menor que y menor o igual que.
- Test utilizando las palabras clave *and* y *or*.
- Test si un elemento está en una lista.
- Test si un elemento no está en una lista.

Sentencias if.

Cuando comprendas las pruebas condicionales puedes empezar a escribir sentencias *if*. Existen varios tipos diferentes de instrucciones *if* y tu elección de cuál usar, depende del número de condiciones que necesites testear. Hemos visto varios ejemplos de instrucciones *if* en el tema acerca de las pruebas, pero ahora profundicemos en el tema.

Instrucciones if sencillas.

El tipo más sencillo de instrucción *if* tiene una sola prueba y una sola acción:

```
if conditional_test:  
    do something
```

Si la prueba condicional se evalúa a True, Python ejecuta el código a continuación de la instrucción *if*, pero si se evalúa como False, Python ignorará el código a continuación del *if*. Es decir, si se cumple la condición (True), se ejecuta el código que hay dentro del *if*, en caso contrario (False), se ignora ese código y el programa lee la siguiente instrucción del programa.

Supongamos que tenemos una variable que representa la edad de una persona y queremos saber si esta persona es mayor para votar. El siguiente código comprueba si esa persona puede votar:

```
age = 19  
  
if age >= 18:  
    print('Eres lo suficientemente mayor para votar.')
```

```
Eres lo suficientemente mayor para votar.  
Process finished with exit code 0
```

La indentación desempeña el mismo papel en las instrucciones *if* que en los bucles *for*.

Todas las líneas indentadas después de la instrucción *if* se ejecutarán si se ha pasado el test y todo el bloque de las líneas indentadas se ignorará si no se pasa el test.

Puedes tener tantas líneas de código como quieras en el bloque siguiente a la instrucción *if*. Añadamos otra línea a la salida si la persona es lo suficientemente mayor para votar preguntando si la persona ya se ha registrado para votar.

```
age = 19  
if age >= 18:  
    print('Eres lo suficientemente mayor para votar.')  
    print('¿Te has registrado ya para votar?')
```

El test condicional ha pasado y ambas llamadas a *print()* están indentadas, así que se imprimen ambas líneas.

```
Eres lo suficientemente mayor para votar.  
¿Te has registrado ya para votar?  
Process finished with exit code 0
```

Si el valor de *age* es menor que 18 este programa no produciría ninguna salida.

Instrucciones *if-else*.

Con frecuencia querrás realizar una acción cuando un test condicional pase y otra acción diferente en todos los demás casos. La sintaxis de Python *if-else* lo hace posible.

Un bloque *if-else* es parecido a una instrucción *if* sencilla, pero la sentencia *else* te permite definir una acción o un conjunto de acciones que serán ejecutadas cuando el test condicional falle.

Mostraremos el mismo mensaje que teníamos previamente, si la persona es suficientemente mayor para votar, pero esta vez añadiremos un mensaje para quien no es lo suficiente mayor para votar.

```
age = 17
if age >= 18:
    print('Eres lo suficientemente mayor para votar.')
    print('¿Te has registrado ya para votar?')
else:
    print('Lo siento, eres muy joven para votar.')
    print('Por favor, regístrate para votar tan pronto como cumplas los 18.')
```

Este

```
Lo siento, eres muy joven para votar.
Por favor, regístrate para votar tan pronto como cumplas los 18.
```

```
Process finished with exit code 0
```

código funciona porque sólo tiene dos posibles situaciones para evaluar: una persona es mayor para votar o no lo es. La estructura *if-else* funciona bien en situaciones en las que quieras que Python siempre ejecute una de las dos acciones. En una cadena de *if-else* simple como esta, una de las dos acciones será ejecutada.

La cadena *if-elif-else*.

Con frecuencia necesitarás comprobar más de dos situaciones posibles y para evaluarlas puedes usar la sintaxis de Python *if-elif-else*. Python sólo ejecuta un bloque en esta cadena. Ejecuta cada test condicional hasta que uno de ellos es lo que buscamos. Cuando una comprobación da el ok, el código que sigue a esa comprobación se ejecuta y Python descarta el resto de comprobaciones.

Muchas situaciones del mundo real implican más de dos condiciones posibles. Por ejemplo, consideremos un parque de atracciones que cobra diferentes tarifas por diferentes grupos de edad:

- Menores de 4 años: gratis.
- Entre 4 y 18 años: 25 euros.
- 18 o mayores de 18: 40 euros.

¿Cómo podemos determinar la tasa para aplicar a una persona con una sentencia *if*? El siguiente código comprueba el grupo de edad

de una persona y luego imprime un mensaje con el precio de admisión.

```
age = 12

if age < 4:
    print('Tu entrada tienen coste 0.')
elif age < 18:
    print('Tu entrada tiene un coste de 25 Euros.')
else:
    print('Tu entrada tiene un coste de 40 Euros.')
```

Tu entrada tiene un coste de 25 Euros.
Process finished with exit code 0

Pero en lugar de imprimir el precio de admisión dentro del bloque *if-elif-else*, sería más concreto establecer sólo el precio dentro del *if-elif-else* y luego hacer una simple llamada a *print()* que se ejecute después de que la cadena haya sido evaluada:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
else:
    price = 40

print(f'Tu entrada cuesta {price} euros.')
```

Tu entrada cuesta 25 euros.
Process finished with exit code 0

Este código produce el mismo resultado que el ejemplo anterior, pero en este caso, el propósito de la cadena *if-elif-else* es más concreto. En lugar de determinar un precio y mostrar un mensaje, simplemente se determina el precio de admisión. Además de ser más eficiente, este código revisado es más fácil de modificar que el enfoque original. Para cambiar el texto del mensaje de salida,

necesitarías cambiar solamente una llamada a `print()` en lugar de tres llamadas por separado.

Utilizar múltiples bloques `elif`.

Puedes utilizar tantos bloques `elif` en tu código como quieras. Por ejemplo, si el parque de atracciones implementara un descuento para personas mayores, podrías añadir otro test condicional al código para determinar aquellas personas que pueden acceder al descuento de mayores. Digamos que cualquiera de 65 años o más pagaría la mitad de una entrada normal, o 20 euros:

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f'Tu entrada cuesta {price} euros.')
```

La mayoría del código no ha cambiado, el segundo bloque `elif` ahora comprueba para asegurarse de que una persona es menor de 65 años antes de asignarle el precio de la entrada a 40 Euros. Ten en cuenta que el valor asignado en el bloque `else`, debe cambiarse a 20 euros porque las únicas edades que llegan a este bloque son personas de 65 años o más.

Omitir el bloque `else`.

Python no requiere un bloque `else` al final de una cadena `if-elif`. A veces un bloque `else` es útil, otras veces es más recomendable usar una declaración `elif` que capte la condición específica que nos interesa.

```

age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f'Tu entrada cuesta {price} euros.')

```

El bloque `else` es una declaración general. Coincide con cualquier condición que no haya sido compatible con una comprobación específica `if` o `elif` y eso a veces puede incluir datos no válidos o incluso maliciosos. Si tiene una especificación final de una condición que estás probando, considera utilizar un bloque final `elif` y omitir el bloque `else`. Como resultado, te asegurarás de que tu código se ejecutará solo en las condiciones correctas.

Prueba de múltiples condiciones.

La cadena `if-elif-else` es poderosa, pero sólo es apropiado utilizarla cuando tan sólo necesitas pasar una prueba. Tan pronto como Python encuentra una prueba satisfactoria, omite el resto de las comprobaciones. Este comportamiento es beneficioso porque es eficiente y te permite comprobar una condición específica.

Sin embargo, a veces es necesario comprobar todas las condiciones de interés. En este caso podrías usar una serie de sentencias `if` simples sin bloques `elif` o `else`. Esta técnica tiene sentido cuando más de una condición puede ser verdadera (`True`) y tú quieres actuar en cada condición que lo sea. Consideremos el ejemplo de la pizzería. Si alguien quiere dos ingredientes en la pizza necesitarás estar seguro de incluir ambos ingredientes en su pizza:

```

requested_toppings = ['champiñones', 'extra de queso']

if 'champiñones' in requested_toppings:
    print('champiñones añadidos.')
if 'pepperoni' in requested_toppings:
    print('pepperoni añadido.')
if 'extra de queso' in requested_toppings:
    print('extra de queso añadido.')

print('\nHas terminado de hacer tu pizza!!')

```

Debido a que cada condición de este ejemplo ha sido evaluada, ambos ingredientes ('champiñones' y 'extra de queso') se han añadido a la pizza.

```
champiñones añadidos.  
extra de queso añadido.  
  
Has terminado de hacer tu pizza!!  
  
Process finished with exit code 0
```

Este código podría no funcionar adecuadamente si utilizamos un bloque *if-elif-else*, porque el código podría detenerse después de la primera comprobación verdadera. Este es el ejemplo:

```
requested_toppings = ['champiñones', 'extra de queso']  
  
if 'champiñones' in requested_toppings:  
    print('champiñones añadidos.')  
elif 'pepperoni' in requested_toppings:  
    print('pepperoni añadido.')  
elif 'extra de queso' in requested_toppings:  
    print('extra de queso añadido.')  
  
print('\nHas terminado de hacer tu pizza!!')
```

Este sería el resultado de ejecutar este código:

```
champiñones añadidos.  
  
Has terminado de hacer tu pizza!!  
  
Process finished with exit code 0
```

Como podemos comprobar, no se añade el ingrediente 'extra de queso' debido a que Python, una vez que encuentra una condición

verdadera en el bloque *if-elif-else*, ignora el código a continuación y no continuará buscando ninguna comprobación más. En resumen, si sólo quieras ejecutar un bloque de código, utiliza la cadena *if-elif-else*. Si necesitas ejecutar más de un bloque utiliza una serie de sentencias *if* independientes.

EJERCICIOS - TRY IT YOURSELF.

5-3. Alien Colors #1:Imagina un alien que acaba de ser eliminado en un juego. Crea una variable llamada *alien_color* y asígnale un valor de 'green', 'yellow' o 'red'.

- Escribe una sentencia *if* para comprobar si el color del alien es verde. Si lo es, imprime un mensaje que indique que el jugador acaba de ganar 5 puntos.
- Escribe una versión de este programa que pase una prueba *if* y otra que no lo haga (la versión que falle no tendrá salida) .

5-4. Alien Colors #2: Escoge un color para un alien como hiciste en el ejercicio 5-3 y escribe una cadena *if-else*.

- Si el color del alien es verde, imprime una frase que indique que el jugador acaba de ganar 5 puntos por disparar al alien.
- Si el color del alien no es verde, imprime una frase que indique que el jugador acaba de ganar 10 puntos.
- Escribe una versión de este programa que ejecute un bloque *if* y otra que ejecute un bloque *else*.

5-5. Alien Colors #3: Convierte la cadena *if-else* del ejercicio 5-4 en una cadena *if-elif-else*.

- Si el alien es verde, imprime un mensaje que indique que el jugador ha ganado 5 puntos.
- Si el alien es amarillo, imprime un mensaje que indique que el jugador ha ganado 10 puntos.
- Si el alien es rojo, imprime un mensaje que indique que el jugador ha ganado 15 puntos.

Escribe tres versiones de este programa. Asegúrate de que cada mensaje se imprime para el color adecuado del alien.

5-6. Stages of Life: Escribe una cadena if-elif-else que determine la etapa de la vida de una persona. Establece una variable para la edad y luego:

- Si la persona es menor de 2 años, imprime un mensaje que indique que esa persona es un bebé.
- Si la persona tiene al menos 2 años pero menos de 4, imprime un mensaje que indique que esa persona es un niño pequeño.
- Si la persona tiene al menos 4 años pero menos de 13, imprime un mensaje que indique que esa persona es un chico.
- Si la persona tiene al menos 13 años pero menos de 20, imprime un mensaje que indique que esa persona es un adolescente.
- Si la persona tiene al menos 20 años pero menos de 65, imprime un mensaje que indique que esa persona es un adulto.
- Si la persona tiene 65 años o más, imprime un mensaje que indique que esa persona es mayor.

5-7. Favorite Fruit: Haz una lista de tus frutas favoritas y luego escribe una serie de sentencias if para buscar ciertas frutas en tu lista.

- Haz una lista de tus tres frutas favoritas y llámala `favorite_fruits`.
- Escribe cinco sentencias. Cada una debe comprobar si un determinado tipo de fruta está en tu lista. Si lo está, el bloque if debe imprimir una frase como ¡Realmente te gustan los plátanos!

Usar instrucciones if con listas.

Puedes hacer un trabajo interesante cuando combinás sentencias `if` con listas. Puedes observar los valores especiales que necesitan ser tratados de distinta manera a otros valores de la lista. Puedes gestionar las condiciones cambiantes de manera eficiente como la disponibilidad de ciertos elementos en un restaurante a través de un cambio. También puedes empezar a probar que tu código funciona como esperabas en todas las situaciones posibles.

Comprobar elementos especiales.

Este capítulo comienza con un ejemplo simple que mostraba cómo manejar un valor especial como 'bmw' que necesitaba ser impreso en formatos diferentes de otros valores de la lista. Ahora que tienes conocimientos básicos de pruebas condicionales y sentencias if, miremos más de cerca cómo observar valores especiales en una lista y manejarlos apropiadamente.

Continuemos con el ejemplo de la pizzería. La pizzería muestra un mensaje cuando se añade un ingrediente a la pizza y está lista. El código para esta acción se puede escribir de manera más eficiente haciendo una lista de los ingredientes que ha pedido el cliente y usar un bucle para informar de cada ingrediente añadido a la pizza.

```
requested_toppings = ['champiñones', 'pimientos verdes', 'extra de queso']

for requested_topping in requested_toppings:
    print(f'Añadiendo {requested_topping}.')

print('\nSe terminado de hacer tu pizza!')
```

Pero ¿qué pasaría si la pizzería se queda sin pimientos verdes? Una sentencia if dentro del bucle puede ayudara a manejar adecuadamente esta situación:

```
requested_toppings = ['champiñones', 'pimientos verdes', 'extra de queso']

for requested_topping in requested_toppings:
    if requested_topping == 'pimientos verdes':
        print('Lo sentimos, nos hemos quedado sin pimientos verdes')
    else:
        print(f'Añadiendo {requested_topping}.')

print('\nSe terminado de hacer tu pizza!')
```

La salida muestra cómo se ha manejado adecuadamente cada ingrediente solicitado:

```
Añadiendo champiñones.  
Lo sentimos, nos hemos quedado sin pimientos verdes  
Añadiendo extra de queso.  
  
Se terminado de hacer tu pizza!  
  
Process finished with exit code 0
```

Comprobar que una lista no esté vacía.

Hasta ahora hemos asumido que cada lista tiene al menos un elemento, pero ahora dejaremos que sean los usuarios quienes proporcionen la información almacenada en las listas, por lo que no podremos dar por hecho que una lista tiene elementos cada vez que se ejecute un bucle. En estas situaciones, es común comprobar si la lista está vacía antes de entrar en el bucle.

Como ejemplo, comprobemos si la lista de los ingredientes está vacía antes de hacer la pizza. Si lo está, solicitaremos al usuario que confirme si quiere una pizza sin ingredientes y si no lo está, haremos la pizza tal y como la cocinamos en los ejemplos anteriores:

```
requested_toppings = []  
  
if requested_toppings:  
    for requested_topping in requested_toppings:  
        print(f'Añadiendo {requested_topping}.')  
    print('\nHemos terminado tu pizza!')  
else:  
    print('¿Estás seguro de que no quieres ingredientes?')
```

Como la lista está vacía, la salida pregunta al usuario si realmente no quiere ingredientes:

```
¿Estás seguro de que no quieres ingredientes?  
  
Process finished with exit code 0
```

En caso de que no lo estuviera, se irían añadiendo los ingredientes de la lista de ingredientes a la pizza, ojo, no a la lista, sino a la pizza.

Utilizar múltiples listas.

¿Qué pasaría si un cliente quiere patatas fritas en su pizza? Podemos usar listas y sentencias if para asegurarnos de esa entrada de datos antes de continuar ejecutando el programa.

Hay que tener mucho cuidado con las peticiones inusuales. En el siguiente ejemplo se definen dos listas, la primera es una lista de ingredientes disponibles en la pizzería, la segunda es la lista de ingredientes que el usuario ha solicitado:

```
available_toppings = ['mushrooms', 'olives', 'green peppers', 'pepperoni', 'pineapple', 'extra cheese']
requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping in available_toppings:
        print(f'Añadiendo {requested_topping} a tu pizza.')
    else:
        print(f'Lo sentimos, no tenemos {requested_topping}.')
print('\nTu pizza está terminada!')
```

Primero definimos una lista de ingredientes disponibles, que podría ser una tupla en el caso de que la pizzería tuviera una selección fija de ingredientes que no cambiara.

A continuación hacemos una lista de ingredientes que quiere el cliente y después recorremos la lista de ingredientes del cliente con un bucle for y una vez dentro del bucle comprobamos si cada ingrediente que solicita el cliente se encuentra en la lista de ingredientes disponibles; si lo está, se añade a la pizza, en caso contrario se muestra un mensaje de no disponibilidad del ingrediente. Cuando se ha terminado de comparar la lista del cliente con la de la pizzería, se sale del bucle y se muestra un mensaje indicando que ya se ha terminado la pizza.

```
Añadiendo mushrooms a tu pizza.
Lo sentimos, no tenemos french fries.
Añadiendo extra cheese a tu pizza.

Tu pizza está terminada!

Process finished with exit code 0
```

En unas pocas líneas de código hemos manejado una situación del mundo real de forma muy efectiva.

EJERCICIOS – TRY IT YOURSELF .

5-8. Hello Admin: Haz una lista de cinco o más usuarios, incluyendo el nombre ‘admin’. Imagina que estás imprimiendo un código que imprimirá un saludo a cada usuario después de que se identifiquen en un sitio web. El bucle recorrerá la lista e imprimirá un saludo a cada usuario:

- Si el usuario es ‘admin’, imprime un saludo especial como ‘*Hola admin, ¿te gustaría ver un informe de estado?*’
- De lo contrario imprime un saludo genérico como ‘*Hola Jaden, gracias por iniciar sesión nuevamente.*’

5-9. No Users: Añade una comprobación if a hello_admin.py para asegurarte de que la lista de usuarios no está vacía.

- Si lo está, imprime el mensaje ‘Necesitamos encontrar algunos usuarios! ’.
- Elimina todos los nombres de usuario de tu lista y asegúrate de que el mensaje se imprime correctamente.

5-10. Cheking Usernames: Haz lo siguiente para crear un programa que simule como un sitio web comprueba que cada uno tiene un nombre de usuario único.

- Haz una lista de cinco o más usuarios llamada current_users.
- Haz otra lista de cinco nombres de usuario llamada new_users. Asegúrate de que uno o dos nombres de usuario también están en la lista current_users.
- Recorre la lista new_users para ver si cada nombre de usuario ya se ha utilizado. Si lo ha hecho, imprime un mensaje para que la persona introduzca un nuevo nombre de usuario. Si el nombre de usuario no ha sido utilizado, imprime un mensaje diciendo que ese nombre está disponible.
- Asegúrate de que tu comparación no distingue entre mayúsculas y minúsculas. Si ‘John’ ya está en uso, ‘JOHN’ no es un nombre válido. (Para hacer esto necesitas hacer una copia de

current_users que contenga una versión en minúsculas de todos los nombres de usuario existentes.)

5-11. Ordinal Numbers: Los números ordinales indican su posición en una lista como primero o segundo. La mayoría de los números ordinales terminan en th, excepto 1, 2 y 3 (first, second, third).

- Almacena números del 1 al 9 en una lista.
- Recorre la lista.
- Utiliza una cadena if-elif-else dentro del bucle para imprimir la terminación ordinal apropiada para cada número. Tu salida debería leer “1st 2nd 3rd 4th 5th 6th 7th 8th 9th” y cada resultado debería estar en una línea separada.

Aplicar estilo a las instrucciones if.

En cada ejemplo de este capítulo hemos visto buenos hábitos de estilo. La única recomendación que proporciona PEP8 para las pruebas condicionales es usar un sólo espacio alrededor de los operadores de comparación como ==, >=, <=. Por ejemplo:

Esto:

```
if number < 4:
```

Es mejor que esto:

```
if number<4:
```

Ese espacio no afecta a la forma en la que Python interpreta el código, simplemente hace que este sea más fácil de leer tanto para uno mismo como para otras personas.

CAPÍTULO 6. DICCIONARIOS.

En este capítulo aprenderás cómo utilizar los diccionarios de Python que te permitirán conectar partes de información relacionada. Aprenderás cómo acceder a la información una vez que esté en el diccionario y cómo modificar esa información. Debido a que los diccionarios pueden almacenar una ingente cantidad de información, aprenderás a recorrer los datos de un diccionario a través de un bucle.

Comprender los diccionarios te permitirá modelar una variedad de objetos del mundo real con mayor precisión. Podrás crear diccionarios que representen una persona y luego almacenar tanta información como quieras acerca de esa persona. Puedes almacenar su nombre, edad, domicilio, trabajo y cualquier otro aspecto de una persona que puedas describir. Podrás almacenar dos tipos de información que se pueda emparejar como una lista de palabras, una lista de montañas y sus altitudes y así sucesivamente.

Un diccionario sencillo.

Imagina un juego de alienígenas que tenga diferentes colores y puntuaciones. Este sencillo diccionario almacena información acerca de un alien en particular.

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

El diccionario almacena el color y la puntuación del alien. Las dos últimas líneas acceden y muestran esa información, como se muestra a continuación:

```
green
5
```

Al igual que con la mayoría de conceptos nuevos de programación, el uso de diccionarios requiere práctica.

Trabajar con diccionarios.

Un diccionario en Python es una colección de pares clave-valor. Cada clave está conectada con un valor y puedes usar una clave para acceder al valor asociado a esa clave. El valor de una clave puede ser un número, una cadena, una lista o incluso otro diccionario. De hecho puedes utilizar cualquier objeto que puedas crear en Python como valor en un diccionario.

En Python, un diccionario se delimita por llaves, con una serie de pares clave-valor dentro de las llaves como se muestra en el ejemplo anterior.

Un par clave-valor es un conjunto de valores asociados entre sí. Cuando proporcionas una clave, Python devuelve el valor asociado a esa clave. Cada clave está conectada a su valor por los dos puntos y cada par clave-valor esta separado por comas. Puedes almacenar tantos pares clave-valor como quieras en un diccionario.

El diccionario más simple tiene exactamente un par clave-valor como se muestra en esta versión modificada del diccionario `alien_0`:

```
alien_0 = {'color': 'green'}
```

Este diccionario almacena una parte de información acerca del `alien_0`, llamada color del alien. La cadena '`color`' es una clave en este diccionario y su valor asociado es '`green`'.

Acceso a valores en un diccionario.

Para obtener el valor asociado a una clave, asigna el nombre del diccionario y luego coloca la clave dentro de un conjunto de corchetes como se muestra a continuación.

```
alien_0 = {'color': 'green'}
print(alien_0['color'])
```

Esto devuelve el valor asociado a la clave '`color`' desde el diccionario `alien_0`.

```
green
```

Puedes tener un número ilimitado de pares clave-valor en un diccionario. Por ejemplo, este el diccionario original `alien_0` con dos pares clave-valor:

```
alien_0 = {'color': 'green', 'points': 5}
```

Ahora puedes acceder al color o a la puntuación de `alien_0`. Si un jugador elimina este alien, puedes buscar cuántos puntos deberías ganar usando un código como este:

```
alien_0 = {'color': 'green', 'points': 5}

new_points = alien_0['points']
print(f'Acabas de ganar {new_points} puntos!')
```

Una vez que hemos definido el diccionario, almacenamos en la variable `new_points` el valor asociado a la clave '`points`' de nuestro diccionario, de la manera que se muestra en la imagen anterior. A continuación imprimimos una frase acerca de cuántos puntos acaba de ganar el jugador:

```
Acabas de ganar 5 puntos!  
Process finished with exit code 0
```

Añadir nuevos pares clave-valor.

Los diccionarios son estructuras dinámicas y puedes añadir nuevos pares clave-valor en cualquier momento. Por ejemplo, para añadir un nuevo par clave-valor hay que proporcionar el nombre del diccionario seguido del nuevo valor de la clave encerrada entre corchetes.

Añadamos dos nuevas piezas de información al diccionario `alien_0`: Las coordenadas `x` e `y` que te ayudarán a mostrar al alien en una posición particular en la pantalla. Situemos al alien en la esquina izquierda de la pantalla 25 píxeles por debajo del borde superior. Debido a que las coordenadas de la pantalla comienzan generalmente en la esquina superior izquierda, colocaremos al alienígena en el borde izquierdo de la pantalla estableciendo la coordenada `x` a 0 y 25 píxeles desde la parte superior estableciendo la coordenada `y` en 25 positivos tal y como se muestra a continuación:

```
alien_0 = {'color': 'green', 'points': 5}  
print(alien_0)  
  
alien_0['x_position'] = 0  
alien_0['y_position'] = 25  
print(alien_0)
```

Comenzamos definiendo el mismo diccionario con el que hemos estado trabajando. Cuando lo imprimimos, mostramos una instantánea de su información. Luego añadimos un nuevo par clave-valor al diccionario: clave '`x_position`' y valor '`0`' y hacemos lo mismo para la clave '`y_position`'. Al imprimir el diccionario modificado podemos ver dos pares clave-valor adicionales.

```
{'color': 'green', 'points': 5}
La      {'color': 'green', 'points': 5, 'x_position': 0, 'y_position': 25}

Process finished with exit code 0
```

versión final del diccionario contiene cuatro pares clave-valor. Los dos originales que especifican el color y la puntuación y dos más que especifican la posición del alien.

NOTA: A partir de Python 3.7 los diccionarios conservan el orden en que fueron definidos. Cuando imprimes un diccionario o lo recorres a través de sus elementos, verás esos elementos en el mismo orden en que fueron añadidos al diccionario.

Empezando con un diccionario vacío.

Unas veces es conveniente o incluso necesario, empezar con un diccionario vacío y después añadir cada nuevo elemento. Para empezar a completar un diccionario vacío, declara un diccionario con un conjunto de llaves vacías y luego añade cada par clave-valor en su propia línea. Por ejemplo, aquí tenemos cómo construir el diccionario *alien_0* siguiendo este enfoque:

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Aquí hemos definido un diccionario vacío y luego le hemos añadido el color y la puntuación. El resultado es el diccionario que hemos estado usando en los ejemplos anteriores:

```
{'color': 'green', 'points': 5}

Process finished with exit code 0
```

Normalmente usarás diccionarios vacíos cuando almacenes datos proporcionados por el usuario o cuando escribas código que genere un gran número de pares clave-valor que se emparejen automáticamente.

Modificar valores de un diccionario.

Para modificar un valor en un diccionario, se da el nombre del diccionario con la clave entre corchetes y luego el nuevo valor que quieras asociar con esa clave. Por ejemplo, consideremos que un alien cambia de verde a amarillo a medida que avanzas en el juego:

```
print(f"El color del alien es {alien_0['color']}")  
  
alien_0['color'] = 'yellow'  
print(f"El color del alien ahora es {alien_0['color']}")
```

La salida muestra que, en efecto, el alien ha cambiado de verde a amarillo.

```
El color del alien es green  
El color del alien ahora es yellow  
  
Process finished with exit code 0
```

Para un ejemplo más interesante, vamos a rastrear la posición de un alien que puede moverse a diferentes velocidades. Almacenaremos un valor que represente la velocidad actual del alien y luego lo utilizaremos para determinar lo lejos que se puede desplazar a la derecha:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}  
print(f"Posición original: {alien_0['x_position']}")  
  
# Mueve el alien a la derecha.  
# Determinar los lejos que se ha movido el alien en base a su velocidad actual.  
if alien_0['speed'] == 'slow':  
    x_increment = 1  
elif alien_0['speed'] == 'medium':  
    x_increment = 2  
else:  
    # Esto debe ser un alien rápido  
    x_increment = 3  
  
# La nueva posición es la nueva posición más el incremento.  
alien_0['x_position'] = alien_0['x_position'] + x_increment  
  
print(f"Nueva posición: {alien_0['x_position']}")
```

Debido a que esta es una velocidad de alien media, su posición se desplazará dos unidades a la derecha.

```
Posición original: 0
Nueva posición: 2

Process finished with exit code 0
```

Esta técnica es bastante genial: cambiando un valor en el diccionario del alien puedes cambiar todo el comportamiento del alien. Por ejemplo, para cambiar la velocidad media del alien a una velocidad rápida puede añadir la línea:

```
alien_0['speed'] = 'fast'
```

El bloque *if-elif-else* asignaría un valor mayor a *x_increment* la próxima vez que se ejecute el código.

Eliminar pares clave-valor.

Cuando ya no necesites una parte de la información almacenada en un diccionario puedes utilizar la sentencia **del** para eliminar completamente un par clave-valor. Todo lo que **del** necesita es el nombre del diccionario y la clave que quieras eliminar.

Por ejemplo, eliminemos la clave '*points*' del diccionario '*alien*', junto con su valor.

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

En este ejemplo, hemos eliminado la clave 'points' del diccionario 'alien_0' y hemos eliminado también el valor asociado a esa clave. La salida muestra que la clave 'points' y el valor 5 se han eliminado pero el resto del diccionario no ha sido afectado.

```
{'color': 'green', 'points': 5}
{'color': 'green'}

Process finished with exit code 0
```

NOTA: Hay que ser cuidadosos, ya que los pares clave-valor se eliminarán de forma permanente.

Diccionario de objetos similares.

El ejemplo previo implicaba almacenar diferentes tipos de información acerca de un objeto, un alien en un juego. También puedes utilizar un diccionario para almacenar un tipo de información sobre varios objetos. Por ejemplo, supongamos que quiere encuestar a varias personas y preguntarles cuál es su lenguaje de programación favorito. Un diccionario es lo más útil para almacenar los resultados de una simple encuesta, como esta:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}
```

Como puedes ver, hemos dividido un diccionario más grande en varias líneas. Cada clave es el nombre de una persona que ha respondido a la encuesta y cada valor es su elección de lenguaje. Cuando sepas que vas a necesitar más de una línea para definir un

diccionario, presiona *ENTER* después de la llave de apertura. Luego indenta la siguiente línea un nivel (4 espacios) y escribe el primer par clave-valor seguido de una coma. Desde este punto en adelante, cuando pulses *ENTER* tu editor indentará automáticamente los siguientes pares clave-valor para que coincidan con el primero.

Una vez que has terminado de definir el diccionario, añade un cierre de llave en una nueva línea después del último par clave-valor e indéntala un nivel para alinearla con las claves del diccionario. Se considera una buena práctica incluir una coma después del último par clave-valor, así estás preparado para añadir un nuevo par en la línea siguiente.

NOTA: Muchos editores tienen algunas funcionalidades que te ayudan a formatear listas extendidas y diccionarios de una forma similar a este ejemplo. También hay otras maneras aceptables de formatear diccionarios grandes, por lo que es posible que observes formatos ligeramente diferentes en tu editor o en otras fuentes.

Para usar este diccionario, proporciona el nombre de una persona que haya realizado la encuesta, puedes ver fácilmente cuál es su lenguaje favorito. Para ver el lenguaje favorito de Sarah:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

language = favorite_languages['sarah'].title()
print(f'El lenguaje favorito de Sarah es {language}.')
```

Asignamos el valor ('c') de la clave ('Sarah') a la variable 'language'. Creando una variable aquí hacemos más limpia la llamada a *print()*. La salida muestra el lenguaje favorito de Sarah:

```
El lenguaje favorito de Sarah es C.
```

```
Process finished with exit code 0
```

Puedes utilizar esta misma sintaxis con cualquier individuo representado en el diccionario.

Utilizar `get()` para acceder a los valores.

Utilizar claves entre corchetes para recuperar el valor que te interesa de un diccionario puede causar un problema potencial: si la clave que buscas no existe obtendrás un error.

Veamos qué ocurre cuando buscamos la puntuación de un alien que no tienen configurada dicha puntuación.

```
alien_0 = {'color': 'green', 'speed': 'slow'}
print(alien_0['points'])
```

Esto da como resultado un rastreo que muestra un `KeyError`:

```
Traceback (most recent call last):
  File "/home/usuario/PycharmProjects/pythoAFondo/capitulo_6_diccionarios/alien_no_points.py", line 4, in <module>
    print(alien_0['points'])
KeyError: 'points'

Process finished with exit code 1
```

Puedes aprender más acerca de capturar errores como este en general en el capítulo 10. Para diccionarios, específicamente puedes usar el método `get()` para configurar un valor por defecto que será devuelto en caso de que la clave solicitada no exista. El método `get()` requiere una clave como primer argumento. Como segundo argumento opcional, puedes pasar el valor a devolver si la clave no existe:

```
alien_0 = {'color': 'green', 'speed': 'slow'}
```

Si la

```
clave point_value = alien_0.get('points', 'No hay puntuación asignada.')
'point_value'
```

`ts'` existe en el diccionario, obtendrás el valor correspondiente. Si no existe, obtendremos el valor por defecto que hemos pasado como segundo parámetro opcional, en este caso, ‘*No hay puntuación asignada.*’ en lugar de un mensaje de error.

```
No hay puntuación asignada.  
Process finished with exit code 0
```

Si hay la posibilidad de que la clave que buscas no exista, considera utilizar el método `get()` en lugar de una notación de corchetes.

NOTA: Si no defines el segundo argumento en la llamada a `get()` y la clave no existe, Python devolverá el valor `None`. El valor especial `None` significa que “no existe el valor”. Esto no es un error: es un valor especial que quiere indicar la ausencia de un valor. Veremos más usos de `None` en el capítulo 8.

EJERCICIOS – TRY IT YOURSELF.

6-1. Person: Usa un diccionario para almacenar información acerca de una persona que conozcas. Almacena su primer nombre, apellido, edad la ciudad en la que vive. Podrías tener claves como `first_name`, `last_name`, `age` y `city`. Imprime cada parte de la información almacenada en el diccionario.

6-2. Favorite Numbers: Usa un diccionario para almacenar números favoritos de personas. Piensa cinco nombres y úsalos como claves de tu diccionario. Piensa en el número favorito de cada persona y almacénalo como valor. Imprime el nombre de cada persona y su número favorito. Para mayor diversión, sondea algunos amigos y obtén datos reales para tu programa.

6-3. Glosario: Un diccionario Python se puede usar para modelar un diccionario actual, sin embargo, para evitar confusión, llamémosle glosario.

- Piensa cinco palabras de programación que hayas aprendido en los capítulos anteriores. Usa esas palabras como claves en tu glosario y almacena sus significados como valores.
- Imprime cada palabra y su significado con un formato de salida claro. Puedes imprimir la palabra seguida de una coma y después su significado, o imprimir la palabra en una línea y luego imprimir su significado indentado en una segunda línea. Utiliza el carácter newline(\n) para insertar un línea en blanco entre cada par palabra-significado en la salida.

Recorrer en bucle un diccionario.

Un sencillo diccionario de Python puede contener sólo unos pocos pares de clave-valor o millones de ellos. Debido a que un diccionario puede contener grandes cantidades de datos, Python permite recorrerlos a través de un bucle. Los diccionarios se pueden utilizar para almacenar información de varias maneras, por lo tanto, existen diferentes maneras de recorrerlos con un bucle. También puedes recorrerlo todos los pares clave-valor, a través de las claves o a través de los valores.

Recorrer con un bucle todos los pares clave-valor.

Antes de explorar los diferentes enfoques del bucle, consideremos un nuevo diccionario diseñado para almacenar información acerca de un usuario en un sitio web. El siguiente diccionario podría almacenar el nombre de usuario, nombre y apellido de una persona:

```
user_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
}
```

Puedes acceder a una parte sencilla de la información del user_0 basado en lo que ya has aprendido en este capítulo. Pero ¿y si quisieras ver todo lo almacenado en el diccionario de este usuario? Para hacerlo puedes recorrer el diccionario con un bucle *for*:

```
user_0 = {  
    'username': 'efermi',  
    'first': 'enrico',  
    'last': 'fermi',  
}  
  
for key, value in user_0.items():  
    print(f"\nClave: {key}")  
    print(f"Valor: {value}")
```

Como se muestra en la imagen, hemos escrito un bucle for para recorrer el diccionario y se han creado nombres para dos variables que contendrán la clave y el valor de cada par clave-valor. Puedes elegir el nombre que quieras para estas dos variables. Este código puede funcionar también si has usado abreviaturas para los nombres de las variables como:

```
for k, v in user_0.items()
```

La segunda parte de la sentencia `for` incluye el nombre del diccionario seguido del método `items()`, que devuelve una lista de los pares clave-valor. El bucle `for` asigna cada uno de esos pares a las dos variables. En ese ejemplo, usamos las variables para imprimir cada claves seguida de su valor asociado.

```
Clave: username
Valor: efermi

Clave: first
Valor: enrico

Clave: last
Valor: fermi

Process finished with exit code 0
```

Recorrer con un bucle todos los pares clave-valor funciona particularmente bien para diccionarios como el de ejemplo de `lenguajes_favoritos.py`, que almacenan el mismo tipo de información para muchas claves diferentes. Si recorres a través de un bucle el diccionario `lenguajes_favoritos.py` obtendrás el nombre de cada persona y su lenguaje de programación favorito. Debido a que las claves siempre se refieren al nombre de una persona y el valor siempre se refiere al lenguaje, usaremos las variables `name` y `language` en el bucle en lugar de `key` y `value`. Esto hará más fácil seguir lo que está ocurriendo dentro del bucle.

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

for name, language in favorite_languages.items():
    print(f"El lenguaje favorito de {name.title()} es {language.title()}.")
```

El resultado se muestra de una forma bastante clara:

```
El lenguaje favorito de Jen es Python.  
El lenguaje favorito de Sarah es C.  
El lenguaje favorito de Edward es Ruby.  
El lenguaje favorito de Phil es Python.  
  
Process finished with exit code 0
```

Este tipo de bucles funcionaría igual de bien si nuestro diccionario almacenara el resultado de encuestar a mil o incluso a un millón de personas.

Recorrer todas las claves de un diccionario.

El método `keys()` se utiliza cuando no necesitas trabajar con todos los valores de un diccionario. Recorramos a través de un bucle el diccionario `favorite_languages` e imprimamos los nombres de cada persona que ha realizado la encuesta:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'ruby',  
    'phil': 'python',  
}  
  
for name in favorite_languages.keys():  
    print(name.title())
```

El bucle `for` le dice a Python que extraiga todas las claves del diccionario `favorite_languages` y las asigne una cada vez a la variable `name`. La salida muestra los nombres de cada persona que hizo la encuesta:

```
Jen  
Sarah  
Edward  
Phil  
  
Process finished with exit code 0
```

Recorrer las claves es el comportamiento por defecto cuando recorres un diccionario, por lo que este código:

```
for name in favorite_languages:
```

Haría exactamente lo mismo que este:

```
for name in favorite_languages.keys():
```

Puedes escoger utilizar el método keys() explícitamente si eso hace que tu código sea más legible o puedes omitirlo si lo deseas. Puedes acceder al valor asociado con cualquier clave que te interese dentro del bucle mediante la clave actual. Imprimamos un mensaje de una pareja de amigos sobre el lenguaje que han escogido. Recorreremos con un bucle los nombres del diccionario como hicimos anteriormente, pero cuando el nombre coincide con uno de nuestros amigos, mostraremos un mensaje sobre su lenguaje favorito:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

friends = ['phil', 'sarah']
for name in favorite_languages.keys():
    print(name.title())

    if name in friends:
        language = favorite_languages[name].title()
        print(f"\t{name.title()}, ya veo que amas {language}.")
```

Hacemos una lista de amigos ('friends') de los que queremos imprimir un mensaje. Dentro del bucle imprimimos los nombres de todas las personas, pero con la sentencia if indicamos (dentro del bucle for) que aquellas que se encuentren en la lista de amigos, además indiquen cuál es su lenguaje favorito, por lo que imprimiremos todos los nombres pero nuestros amigos recibirán un mensaje especial:

```
Jen
Sarah
    Sarah, ya veo que amas C.
Edward
Phil
    Phil, ya veo que amas Python.

Process finished with exit code 0
```

También puedes usar el método `keys()` para averiguar si una persona en concreto que ha sido encuestada. Esta vez averigüemos si Erin hizo la encuesta:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

if 'erin' not in favorite_languages.keys():
    print('Erin, por favor, haz la encuesta.')
```

El método `keys()` no es sólo para bucles: en realidad devuelve una lista de todas las claves; en el `if` simplemente comprueba si 'Erin' está en la lista porque si no lo está imprime un mensaje invitándola a realizar la encuesta:

```
Erin, por favor, haz la encuesta.

Process finished with exit code 0
```

Recorrer con un bucle las claves de un diccionario en un orden en particular.

A partir de Python 3.7, recorrer un diccionario devuelve los elementos en el mismo orden en que se insertaron. A veces, sin embargo, querrás recorrer un diccionario en un orden diferente.

Una manera de hacerlo es ordenar las claves a medida que van siendo devueltas en el bucle. Puedes utilizar la función sorted() para obtener una copia de las claves en orden:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'ruby',  
    'phil': 'python',  
}  
  
for name in sorted(favorite_languages.keys()):  
    print(f'{name.title()}, gracias por hacer la encuesta.')
```

Esta sentencia for es como otras salvo porque hemos envuelto la función sorted() alrededor del método *dictionary.keys()*. Esto le dice a Python que liste todas las claves del diccionario y ordene dicha lista antes de recorrerla. La salida muestra quienes ha realizado la encuesta con sus nombres en orden alfabético:

```
Edward, gracias por hacer la encuesta.  
Jen, gracias por hacer la encuesta.  
Phil, gracias por hacer la encuesta.  
Sarah, gracias por hacer la encuesta.
```

```
Process finished with exit code 0
```

Recorrer todos los valores de un diccionario.

Si estás interesado principalmente en los valores que contiene un diccionario, puedes utilizar el método values() para devolver una lista de valores sin las claves.

Por ejemplo, di que sólo necesitas una lista de todos los lenguajes de programación elegidos en nuestra encuesta de lenguajes de programación sin el nombre de las personas que han hecho cada elección:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

print('Los siguientes lenguajes han sido mencionados: ')
for language in favorite_languages.values():
    print(language.title())
```

La instrucción `for` aquí extrae cada valor del diccionario y lo asigna a la variable `language`. Cuando se imprimen esos valores, obtenemos una lista de todos los lenguajes elegidos:

```
Los siguientes lenguajes han sido mencionados:
Python
C
Ruby
Python

Process finished with exit code 0
```

Este enfoque extrae todos los valores del diccionario sin comprobar si existen repeticiones. Esto puede funcionar bien con un pequeño número de valores, pero en una encuesta con un gran número de entrevistados, esto daría como resultado una lista con muchas repeticiones. Para ver la elección de cada lenguaje sin repeticiones podemos usar `set`. `Set` es una colección en la cual cada elemento debe ser único:

```
favorite_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
}

print('Los siguientes lenguajes han sido mencionados: ')
for language in set(favorite_languages.values()):
    print(language.title())
```

Cuando envuelves una lista con elementos duplicados dentro de una función `set()`, Python identifica los elementos únicos en la lista y construye un conjunto con ellos. En este ejemplo usamos `set()` para extraer los lenguajes no repetidos en `favorite_languages.values()`. El resultado es una lista de lenguajes no repetidos que han sido mencionados por las personas de la encuesta:

```
Los siguientes lenguajes han sido mencionados:  
Ruby  
C  
Python  
  
Process finished with exit code 0
```

NOTA: también puedes construir un conjunto de elementos no repetidos usando directamente llaves y separando los elementos por comas:

```
languages = {'python', 'ruby', 'python', 'c'}  
print(languages)  
  
{'c', 'ruby', 'python'}  
  
Process finished with exit code 0
```

Es fácil confundir conjuntos con diccionarios porque ambos están envueltos en llaves. Cuando ves llaves, pero no ves pares clave-valor, probablemente estés viendo un conjunto. A diferencia de las listas y los diccionarios, los conjuntos no conservan los elementos en ningún orden específico.

EJERCICIOS – TRY IT YOURSELF.

6-4. Glossary: Ahora que sabes cómo recorrer un diccionario, limpia el código del ejercicio 6-3 reemplazando tus series de llamadas a `print()` con un bucle que recorra las claves y los valores del diccionario. Cuando estés seguro de que tu bucle funciona, añade cinco términos Python más a tu diccionario. Cuando ejecutes tu programa de nuevo, esas nuevas palabras y significados deberían ser incluidas automáticamente en la salida.

6-5. Rivers: Haz un diccionario que contenga los tres mayores ríos y el país que atraviesa cada uno. Un par clave-valor puede ser ‘nile’ : ‘egypt’.

- Utiliza un bucle para imprimir una frase de cada río como, ‘*El Nilo recorre Egipto.*’
- Utiliza un bucle para imprimir el nombre de cada río incluido en el diccionario.
- Utiliza un bucle para imprimir el nombre de cada país incluido en el diccionario.

6-6. Polling: Utiliza el código de *favorite_languages.py*.

- Haz una lista de las personas que han hecho la encuesta. Incluye algunos nombres que ya estén en el diccionario y otros que no lo estén.
- Recorre la lista de personas que han hecho la encuesta. Si la han hecho, imprime un mensaje de agradecimiento por responder, si no lo han hecho, imprime un mensaje invitándolas a hacerla.

Nesting.

A veces querrás almacenar múltiples diccionarios en una lista o una lista de elementos como valor en un diccionario. A esto se le conoce como anidar. Puedes anidar diccionarios dentro de una lista, una lista de elementos dentro de un diccionario o incluso un diccionario dentro de otro diccionario. El anidamiento es una característica muy poderosa como demuestran los siguientes ejemplos.

Una lista de diccionarios.

El diccionario alien_0 contiene variedad de información acerca de un alien, pero no tiene espacio para almacenar información acerca de un segundo alien y mucho menos de una pantalla llena de aliens. ¿Cómo puedes manejar una flota de alienígenas? Una forma es hacer una lista de aliens en la que cada alien es un diccionario de información ese alien. Por ejemplo, el siguiente código construye una lista de tres aliens:

```

alien_0 = {'color': 'green', 'points': 5}
alien_1 = {'color': 'yellow', 'points': 10}
alien_2 = {'color': 'red', 'points': 15}

aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)

```

Primero creamos tres diccionarios, cada uno representando a un alien. Luego almacenamos esos diccionarios en una lista llamada `aliens`. Finalmente recorremos la lista e imprimimos cada alien:

```

{'color': 'green', 'points': 5}
{'color': 'yellow', 'points': 10}
{'color': 'red', 'points': 15}

Process finished with exit code 0

```

Un ejemplo más realista podría involucrar más de tres aliens con código que genere automáticamente cada alien. En el siguiente ejemplo utilizamos `range()` para crear una flota de 30 aliens:

```

# Creamos una lista vacía para almacenar los aliens.
aliens = []

Este

# Generamos 30 aliens
for alien_number in range(30):
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

# Mostramos los 5 primeros aliens
for alien in aliens[:5]:
    print(alien)

print("...")

# Mostramos cuántos aliens se han creado
print(f"Número total de aliens: {len(aliens)}")

```

Este ejemplo comienza con una lista vacía que contendrá todos los aliens que sean creados. La función `range()` devuelve una serie de números que sólo le dice a Python cuántas veces queremos que se repita el bucle. Cada vez que el bucle se ejecuta creamos un nuevo

alien y luego lo añadimos a la lista aliens. A continuación utilizamos un slice (devanado) para mostrar los cinco primeros aliens y luego imprimimos la longitud de la lista para demostrar que realmente hemos generado una flota completa de 30 aliens:

```
{'color': 'green', 'points': 5, 'speed': 'slow'}  
...  
Número total de aliens: 30  
  
Process finished with exit code 0
```

Estos aliens tienen las mismas características pero Python considera a cada uno como un objeto diferente, lo que nos permite modificar cada alien de manera individual.

¿Cómo podríamos trabajar con un grupo de aliens como este? Imagina que un aspecto del juego es que algunos aliens cambien de color y se muevan más rápido a medida que progrese el juego. Cuando sea el momento de cambiar el color, podemos usar un bucle for y una sentencia if para cambiar el color de los aliens. Por ejemplo, para cambiar los primeros tres aliens a amarillo, velocidad media que valen 10 puntos cada uno, podemos hacer esto:

```
for alien in aliens[:3]:  
    if alien['color'] == 'green':  
        alien['color'] = 'yellow'  
        alien['speed'] = 'medium'  
        alien['points'] = 10
```

Puesto que queremos modificar los tres primeros aliens, recorremos con un bucle el slice que incluya sólo a los tres primeros aliens. Todos los aliens son verdes ahora pero no queremos que siempre sea ese el caso, por lo que escribimos una sentencia if para asegurarnos de que sólo estamos modificando los aliens verdes. Si el alien es verde, cambiamos el color a 'yellow', la velocidad a 'medium' y el valor de puntuación a 10, como se muestra en la siguiente salida:

```
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
{'color': 'green', 'points': 5, 'speed': 'slow'}  
{'color': 'green', 'points': 5, 'speed': 'slow'}  
...  
Número total de aliens: 30  
  
Process finished with exit code 0
```

Puedes expandir este bucle añadiendo un bloque `elif` que vuelva rojos a los aliens amarillo, la velocidad rápida y que valen 15 puntos cada uno. Sin mostrar todo el programa de nuevo, este bucle se parecería a esto:

```
for alien in aliens[:3]:  
    if alien['color'] == 'green':  
        alien['color'] = 'yellow'  
        alien['speed'] = 'medium'  
        alien['points'] = 10  
    elif alien['color'] == 'yellow':  
        alien['color'] = 'red'  
        alien['speed'] = 'fast'  
        alien['points'] = 15
```

Es común almacenar un número de diccionarios en una lista cuando cada diccionario contiene muchos tipos de información acerca de un objeto. Por ejemplo, podrías crear un diccionario para cada usuario de un sitio web como hicimos en `users.py` y almacenar diccionarios individuales en una lista llamada `users`. Todos los diccionarios de la lista podrían tener una estructura idéntica por lo que podrías recorrer la lista con un bucle y trabajar con cada objeto del diccionario de la misma manera.

Un lista en un diccionario.

En lugar de poner un diccionario dentro de una lista, a veces es más práctico poner una lista dentro de un diccionario. Por ejemplo, consideremos cómo podrías describir una pizza que alguien ha pedido. Si sólo usaras una lista, todo lo que realmente podrías

almacenar sería una lista de los ingredientes de la pizza. Con un diccionario, una lista de ingredientes puede ser sólo uno de los aspectos de la pizza que estás describiendo.

En el siguiente ejemplo, se almacenan dos tipos de información para cada pizza: un tipo de corteza y una lista de ingredientes. La lista de ingredientes es un valor asociado con la clave 'toppings'. Para usar los elementos de la lista proporcionaremos el nombre del diccionario y la clave 'toppings' como lo haríamos con cualquier diccionario. En lugar de devolver un valor simple, obtendremos una lista de ingredientes:

```
pizza = {'corteza': 'delgada',
          'ingredientes': ['champiñones', 'extra de queso']}

# Resume el pedido
print(f"Has pedido una pizza de corteza - {pizza['corteza']}")

for topping in pizza['ingredientes']:
    print("\t" + topping)
```

La siguiente salida resume la pizza que vamos a cocinar:

```
Has pedido una pizza de corteza - delgada con los siguientes ingredientes:
  champiñones
  extra de queso

Process finished with exit code 0
```

Puedes anidar una lista dentro de un diccionario siempre que quieras asociar más de un valor a una sola clave de diccionario. En el anterior ejemplo de los lenguajes de programación favoritos, si quisieramos almacenar las respuesta de cada persona en una lista la gente podría elegir más de un lenguaje favorito. Cuando recorremos un diccionario, el valor asociado con cada persona podría ser una lista de lenguajes en lugar de un sólo lenguaje. Dentro del bucle for del diccionario otro bucle for para recorrer la lista de lenguajes asociados a cada persona:

```

favorite_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

for name, languages in favorite_languages.items():
    print(f"\n{name.title()}'s favorite languages are:")
    for language in languages:
        print(f"\t{language.title()}")

```

Como puedes ver, el valor asociado a cada nombre es ahora una lista. Observa que algunas personas tienen un lenguaje de programación favorito y otras tienen varios. Cuando recorremos el diccionario utilizamos el nombre de variable *languages* para capturar cada valor del diccionario puesto que sabemos que cada valor será una lista. Dentro del bucle principal del diccionario usamos otro bucle *for* para recorrer la lista de lenguajes favoritos de cada persona. Ahora cada persona puede enumerar tantos lenguajes favoritos como desee:

```

Jen's favorite languages are:
    Python
    Ruby

Sarah's favorite languages are:
    C

Edward's favorite languages are:
    Ruby
    Go

Phil's favorite languages are:
    Python
    Haskell

Process finished with exit code 0

```

Para redefinir este programa más allá incluso, podrías incluir una sentencia *if* al principio del bucle *for* del diccionario para ver si cada persona tiene más de un lenguaje favorito atendiendo al valor de

`len(languages)`. Si lo tienen, la salida quedará igual, pero si sólo tienen uno se puede cambiar la redacción para que muestre la frase en singular: “*Sarah’s favorite language is C.*”

```
favorite_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

for name, languages in favorite_languages.items():
    if len(languages) > 1:
        print(f"\n{name.title()}'s favorite languages are:")
    else:
        print(f"\n{name.title()}'s favorite language are:")
    for language in languages:
        print(f"\t{language.title()}")
```

NOTA: No deberías anidar las listas y diccionarios demasiado profundo. Si lo haces o estás trabajando con el código de otra persona niveles significativos de anidamiento, lo más probable es

```
Jen's favorite languages are:
    Python
    Ruby
Sarah's favorite language are:
    C
Edward's favorite languages are:
    Ruby
    Go
Phil's favorite languages are:
    Python
    Haskell
```

```
Process finished with exit code 0
```

que exista una manera más sencilla de resolver el problema.

Un diccionario en un diccionario.

Puedes anidar un diccionario dentro de otro diccionario, pero tu código puede volverse complicado rápidamente cuando lo hagas. Por ejemplo, si tienes mucho usuarios para un sitio web, cada uno con un nombre de usuario distinto, puedes utilizar los nombres de usuario como claves. Luego puedes almacenar información de cada usuario utilizando un diccionario como valor asociado con su

nombre de usuario. En el siguiente listado, almacenamos tres piezas de información de cada usuario: su nombre, apellido y ubicación. Accederemos a esta información recorriendo los nombres de usuario y el diccionario de información asociado con cada uno de ellos:

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
    },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
    },
}

for username, user_info in users.items():
    print(f"\nUsername: {username}")
    full_name = f"{user_info['first']} {user_info['last']}"
    location = user_info['location']

    print(f"\tFull name: {full_name.title()}")
    print(f"\tLocation: {location.title()}")
```

Primero definimos un diccionario llamado *users* con dos claves: una para cada nombre de usuario '*aeinstein*' y '*mcurie*'. El valor asociado con cada clave es un diccionario que incluye el nombre, apellido y ubicación de cada usuario. Al recorrer el diccionario, Python asigna cada clave a la variable *username* y el diccionario asociado con cada nombre de usuario es asignado a la variable *user_info*. Una vez dentro del bucle principal del diccionario, imprimimos el nombre de usuario.

A continuación empezamos accediendo al diccionario interno. La variable *user_info* que contiene el diccionario con la información del usuario, tiene tres claves: '*first*', '*second*' y '*location*'. Utilizaremos cada clave para generar un formato limpio del nombre completo y la ubicación de cada persona y luego imprimiremos un resumen de todo lo que conocemos de cada una:

```
Username: aeinstein
Full name: Albert Einstein
Location: Princeton

Username: mcurie
Full name: Marie Curie
Location: Paris

Process finished with exit code 0
```

Observa que la estructura de cada diccionario de usuario es idéntica. Aunque Python no lo requiere, esta estructura facilita el trabajo con diccionarios anidados. Si cada usuario del diccionario tiene diferentes claves, el código dentro del bucle *for* podrías ser más complicado.

EJERCICIOS – TRY IT YOURSELF.

6-7. People: Comienza con el programa que escribiste en el ejercicio 6-1. Haz dos nuevos diccionarios representando a diferentes personas y almacena los tres diccionarios en una lista llamada *people*. Recorre tu lista de personas. A medida que recorras la lista, imprime todo lo que sepas de cada persona.

6-8. Pets: Haz varios diccionarios donde cada uno represente a una mascota diferente. En cada uno incluye el tipo de animal y el nombre del propietario. Almacena esos diccionarios en una lista llamada *pets*. A continuación, mientras recorres la lista imprime todo lo que sepas de cada mascota.

6-9. Favorite Places. Haz un diccionario llamado *favorite_places*. Piensa tres nombres para usar como claves en el diccionario y almacena de uno a tres lugares favoritos para cada persona. Para hacer este ejercicio un poco más interesante, pregunta a algunos amigos el nombre de unos cuantos de sus lugares favoritos. Recorre el diccionario e imprime el nombre de cada persona y sus lugares favoritos.

6-10. Favorite Numbers: Modifica el programa del ejercicio 6-2 para que cada persona pueda tener más de un número favorito. Luego imprime el nombre de cada persona con sus números favoritos.

6-11. Cities: Haz un diccionario llamado *cities*. Usa el nombre de tres ciudades como claves de tu diccionario. Crea un diccionario de información de cada ciudad e incluye el país donde está esa ciudad, su población aproximada y un dato de la ciudad. Las claves para cada diccionario de ciudades podría ser algo como *country*, *population* and *fact*. Imprime el nombre de cada ciudad y toda la información que hayas almacenado sobre ella.

6-12. Extensions: Ahora estamos trabajando con ejemplos lo suficientemente complejos como para que se puedan ampliar de muchas maneras. Utiliza un programa de ejemplo de este capítulo y amplíalo añadiendo nuevas claves y valores, cambiando el contexto del programa y mejorando el formato de salida.

CAPÍTULO 7. ENTRADAS DE USUARIO Y BUCLES WHILE.

Muchos programas están escritos para resolver un problema del usuario final. Para hacerlo, necesitas normalmente algo de información del usuario, por ejemplo, digamos que alguien quiere saber si tiene la edad suficiente para votar. Si escribes un programa que responda a esta pregunta, necesitarás saber la edad del usuario antes de dar una respuesta. El programa tendrá que pedirle al usuario que introduzca o ingrese su edad. Una vez que el programa tiene esta entrada puede compararla con la edad de voto para comprobar si el usuario tiene la edad suficiente para votar y luego informar del resultado.

Cómo trabaja la función `input()`.

La función `input()` detiene tu programa a espera al usuario para que introduzca el siguiente texto. Una vez que Python recibe la entrada del usuario, asigna esa entrada a una variable que ha sido declarada para que trabajes con ella.

Por ejemplo, el siguiente programa le pide al usuario que introduzca algún texto, después el mensaje será mostrado de vuelta en la pantalla del usuario.

```
message = input("Dime algo y lo repetiré para ti.\n")
print(message)
```

La función **`input()`** tiene un argumento: el *prompt*, o instrucciones, que queremos mostrar al usuario para que sepa qué hacer. En este ejemplo, cuando Python ejecuta la primera línea el usuario puede ver en el *prompt* “*Dime algo y lo repetiré para ti:*”. El programa espera hasta que el usuario introduce su respuesta y continúa después de pulsar *ENTER*. La respuesta se asigna a la variable *message* y a continuación *print(message)* lo muestra la entrada en pantalla al usuario.

```
Dime algo y lo repetiré para tí: Gracias por hacer de loro  
Gracias por hacer de loro  
  
Process finished with exit code 0
```

NOTA: Sublime Text y otros editores no ejecutan programas que soliciten la entrada al usuario. Puedes utilizar estos editores para escribir programas, pero necesitarás ejecutarlos en una terminal.

Escribir mensajes claros.

Cada vez que utilizas la función **`input()`**, podrías incluir mensaje limpio, *easy-to follow*, que diga al usuario qué tipo de información estás buscando exactamente. Cualquier frase que diga al usuario que entrada podría funcionar. Por ejemplo:

```
name = input("Por favor, introduce tu nombre: ")  
print(f"Hola, {name}!")
```

Añade un espacio al final de tus mensajes (después de los dos puntos en el ejemplo anterior) para separar el mensaje de la respuesta del usuario y ponerle más fácil a tu usuario dónde tiene que introducir su texto:

```
Por favor, introduce tu nombre: JuanJe  
Hola, JuanJe!  
  
Process finished with exit code 0
```

A veces querrás escribir un mensaje más largo que una sola línea. Por ejemplo, puedes querer decirle al usuario por qué estás pidiéndola determinada entrada de datos. Puedes asignar tu mensaje a una variable y pasar esa variable a la función **`input()`**. Esto te permite construir tu mensaje de varias líneas y después escribir una sentencia **`input()`** limpia:

```
prompt = "Si me dices quién eres podremos personalizar los mensaje que veas."
prompt += "\n¿Cuál es tu nombre?: "

name = input(prompt)
print(f"Hola, {name}!")
```

Este ejemplo muestra una manera de construir una cadena multi-línea. La primera línea asigna la primera parte del mensaje a la variable *prompt*. En la segunda línea, el operador `+=` coge la cadena que ha sido asignada a *prompt* y la añade a la nueva cadena al final.

El mensaje abarca ahora dos líneas, de nuevo con un espacio después de la pregunta que marca para una mayor claridad.

```
Si me dices quién eres podremos personalizar los mensaje que veas.
¿Cuál es tu nombre?: JuanJe
Hola, JuanJe!

Process finished with exit code 0
```

Utilizar `int()` para aceptar entradas numéricas.

Cuando usas la función **`input()`**, Python interpreta cualquier cadena que introduzca el usuario. Considera la siguiente sesión de interpretación que pregunta por la edad al usuario:

```
age = input("¿Cuál es tu edad?: ")
print(age)
```

```
¿Cuál es tu edad?: 21
21

Process finished with exit code 0
```

El usuario introduce el número 21, pero cuando Python preguntamos a Python por el valor de la edad devolverá '21', la cadena que representa el valor numérico introducido. Sabemos que Python ha interpretado la entrada como una cadena porque el número está encerrado entre llaves. Si todo lo que necesitas es imprimir la entrada, esto funcionará. Pero si intentas usar la entrada como un numero obtendrás un error:

```
age = input("¿Cuál es tu edad?: ")
print(age >= 18)
```

```
¿Cuál es tu edad?: 21
Traceback (most recent call last):
  File "/home/usuario/PycharmProjects/pythonCrashCourse/capitulo_7_user_input_and_while/age.py", line 6, in <module>
    print(age >= 18)
TypeError: '>=' not supported between instances of 'str' and 'int'

Process finished with exit code 1
```

Cuando intentas utilizar la entrada para hacer una comparación numérica, Python genera un error porque no se pueden comparar cadenas con enteros: la cadena '21' asignada a 'age' no se puede comparar con el valor numérico 18.

Podemos resolver este problema utilizando la función **int()** que le dice a Python que trate la entrada como un valor numérico. La función **int()** convierte una cadena representando a un número en una representación numérica, como se muestra a continuación:

```
age = input('¿Cuál es tu edad?: ')
age = int(age)
print(age >= 18)
```

```
¿Cuál es tu edad?: 21
True

Process finished with exit code 0
```

En este ejemplo, cuando introducimos 21 en el mensaje, Python interpreta el número como una cadena, pero el valor es convertido a representación numérica con `int()`. Ahora Python puede ejecutar el test condicional: comparar la edad (que representa el valor numérico 21) y 18 para ver si 'age' es mayor o igual que 18. Esta prueba se evalúa a **True**.

¿Cómo se usa la función `int()` en un programa real? Considera un programa que determine si una persona es lo suficientemente alta para subir a una montaña rusa:

```
height = input('¿Cuál es tu altura, en pulgadas?: ')
height = int(height)

if height >= 48:
    print("\nEres lo suficientemente alto para subir!")
else:
    print("\nPodrás montar cuando seas un poco más mayor.")
```

El programa puede comparar `height` con 48 porque `height = int(height)` convierte el valor introducido a representación numérica antes de hacer la comparación. Si el número introducido es igual o mayor a 48, decimos que el usuario es lo suficientemente alto:

```
¿Cuál es tu altura, en pulgadas?: 71

Eres lo suficientemente alto para subir!

Process finished with exit code 0
```

Cuando uses entradas numéricas para hacer cálculos y comparaciones asegúrate de convertir primero la entrada a representación numérica.

El operador módulo %.

Una herramienta muy útil para trabajar con valores numéricos es el operador módulo (%), que divide un número entre otro y devuelve el resto.

El operador módulo no te dice cuántas veces un número encaja en otro, sólo te dice cuál es su resto.

Cuando un número es divisible por otro, el resto es 0, por lo que el operador módulo siempre devolverá 0. Puedes utilizar esta circunstancia para determinar si un número es par o impar.

```
number = input("Introduce un número y te diré si es par o impar: ")
number = int(number)

if number % 2 == 0:
    print(f"\nEl número {number} es par.")
else:
    print(f"\nEl número {number} es impar.")
```

Los pares siempre son divisibles entre dos por lo que si el módulo de un número es cero (`if number % 2 == 0`) el número es par. De otra manera es impar.

```
Introduce un número y te diré si es par o impar: 42

El número 42 es par.

Process finished with exit code 0
```

EJERCICIOS – TRY IT YOURSELF.

7-1. Rental Car: Escribe un programa que pregunte al usuario qué tipo de coche quiere. Imprime un mensaje sobre el coche como “Déjame ver si puedo encontrarte un Subaru.”

7-2. Restaurant Seating: Escribe un programa que pregunte al usuario cuántas personas serán en el grupo de la cena. Si la respuesta es más de 8, imprime un mensaje diciendo que tendrán que esperar una mesa libre, en caso contrario infórmale que hay mesa disponible.

7-3. Múltiples of Ten: Pregunta al usuario por un número y dile si el número es múltiplo de 10 o no.

Introducción a los bucles while.

El bucle for toma una colección de elementos y ejecuta un bloque de código para cada uno de los elementos de la colección. Por el contrario, el bucle while se ejecuta siempre y cuando o mientras que, una determinada condición sea verdadera.

El bucle while en acción.

Puedes usar el bucle while para contar a través de una serie de números. Por ejemplo, el siguiente bucle while cuenta del 1 al 5:

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

En la primera línea empezamos contando desde 1 asignando a *current_number* el valor 1. El bucle while entonces está configurado para ejecutarse mientras que el valor de *current_number* sea menor o igual que 5. El código dentro del bucle imprime el valor de *current_number* y luego le añade 1 con '+= 1' (el operador += es un atajo para indicar *current_number* = *current_number* + 1).

Python repite el bucle mientras que la condición *current_number* <= 5 es verdadera. Puesto que 1 es menor que 5, Python imprime 1 y luego añade 1, haciendo que el número actual sea 2. Puesto que 2 es menor que 5, Python imprime 2 y añade de nuevo 1 haciendo que el número actual sea 3 y así sucesivamente. Una vez que el valor de *current_number* es mayor que 5, el bucle se detiene y el programa finaliza:

```
1
2
3
4
5

Process finished with exit code 0
```

Lo más probable es que los programas que uses a diario contengan bucles while. Por ejemplo, un juego necesita un bucle while para seguir ejecutándose tanto tiempo como quieras estar jugando por lo que puede dejar de hacerlo en el momento en que le pidas que se detenga. Los programas no serían divertidos de usar si se detuvieran antes de que les dijéramos que hagan o se mantuviieran funcionando incluso después de que queramos detenerlos por lo que los bucles while son muy útiles.

Permitir al usuario elegir cuándo acabar.

Podemos hacer que el programa *parrot.py* se ejecute tanto tiempo como el usuario quiera poniendo la mayor parte del programa dentro de un bucle while. Definiremos un valor de detención y luego mantendremos el programa ejecutándose mientras que el usuario no introduzca dicho valor de detención.

```
prompt = "\nDime algo y lo repetiré para tí: "
prompt += "\nIntroduce 'quit' para finalizar el programa: "

message = ""

while message != 'quit':
    message = input(prompt)
    print(message)
```

Primero definimos un mensaje que dice al usuario sus dos opciones: introducir un mensaje o introducir ‘quit’ para finalizar el programa. Luego configuramos una variable *message* para hacer un seguimiento de los valores que introduce el usuario. Definimos *message* como una cadena vacía “” por lo que Python tienen algo que comprobar la primera vez que alcanza la línea *while*. La primera vez que el programa se ejecuta y Python alcanza la sentencia *while*, necesita comparar el valor de *message* con ‘quit’ pero el usuario no ha introducido nada todavía. Si Python no tuviera nada con qué comparar no podría continuar ejecutando el programa. Para resolver este problema nos aseguramos de proporcionar un valor inicial a *message*. Aunque sólo es una cadena vacía, para Python tendrá sentido y le permitirá realizar la comprobación que hace que el

bucle `while` funcione. El bucle `while` se ejecuta mientras que el valor de `message` no sea '`quit`'.

La primera vez que recorremos el bucle, `message` sólo es una cadena vacía, por lo que Python entra en el bucle. En `message = input(prompt)`, Python muestra el mensaje por pantalla y espera a que el usuario introduzca su entrada. Lo que sea que el usuario introduzca será asignado a `message` e impreso. Mientras que el usuario no introduzca '`quit`', el mensaje se mostrará de nuevo y Python esperará más entradas. Cuando el usuario finalmente introduzca '`quit`', Python detendrá la ejecución del bucle `while` y finalizará el programa:

```
Dime algo y lo repetiré para tí:  
Introduce 'quit' para finalizar el programa. Hello everyone!  
Hello everyone!  
  
Dime algo y lo repetiré para tí:  
Introduce 'quit' para finalizar el programa. Hello again  
Hello again  
  
Dime algo y lo repetiré para tí:  
Introduce 'quit' para finalizar el programa. quit  
quit  
  
Process finished with exit code 0
```

Este programa funciona bien salvo porque imprime la palabra '`quit`' como si fuera un mensaje más que introduce el usuario para mostrarlo y no queremos que lo haga ya que es el valor que finaliza el bucle. Una simple comprobación `if` lo soluciona:

```
prompt = "\nDime algo y lo repetiré para tí: "  
prompt += "\nIntroduce 'quit' para finalizar el programa."  
  
message = ""  
  
while message != 'quit':  
    message = input(prompt)  
  
    if message != 'quit':  
        print(message)
```

Ahora el programa hace una comprobación rápida antes de mostrar el mensaje por pantalla y sólo lo imprime si no coincide con ‘*quit*’.

Utilizar un *flag*.

En el ejemplo anterior teníamos un programa que realizaba ciertas tareas mientras que la condición dada fuera verdadera. Pero ¿qué sucedería en programas más complicados en los que diferentes eventos pudiera provocar que el programa se detuviera?

Por ejemplo, en un juego, cantidad de eventos diferentes podrían finalizarlo. Cuando el jugador se queda sin barcos, su tiempo se agota o las ciudades que se supone que debe proteger están todas destruidas, el juego debería terminar. Necesita finalizar si sucede cualquiera de esos eventos. Si son muchos los eventos que pueden hacer que el programa se detenga, intentar verificar todas esas condiciones en un sólo bucle while se puede volver complicado y dificultoso.

Para un programa que puede ejecutarse sólo mientras que las condiciones sean verdaderas, puedes definir una variable que determine si todo el programa está activo o no. Esta variable, llamada *flag*(bandera), actúa como una señal para el programa. Podemos escribir nuestros programas para que se ejecuten mientras que *flag* está configurado a *True* y detenerlos cuando cualquiera de los muchos eventos establezcan el valor de *flag* a *False*. Como resultado, todas nuestras sentencias while necesitan comprobar una sola condición: si el *flag* está actualmente a *True* o no. Entonces, todas nuestras otras comprobaciones (para ver si ha ocurrido un evento que haya configurado el *flag* a *False*) se pueden organizar limpiamente en el resto del programa.

Añadamos un *flag* a *parrot.py* de la sección anterior. Este *flag*, que llamaremos *active*, (piensa que puedes llamarlo como quieras) monitoreará si el programa puede seguir ejecutándose o no:

```
prompt = "\nDime algo y lo repetiré para tí: "
prompt += "\nIntroduce 'quit' para finalizar el programa. "

active = True

while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Hemos configurado la variable `active` a `True` por lo que el programa comienza con un estado activo. Al hacerlo, la instrucción `while` es más sencilla porque no se realiza ninguna comparación en la propia instrucción `while`; la lógica es cuidadosa en otras partes del programa. Mientras la variable `active` siga siendo `True` el bucle continuará ejecutándose.

En la sentencia `if`, dentro del bucle `while`, comprobamos el valor de `message` una vez que el usuario ha introducido su entrada. Si el usuario introduce '`quit`' configuraremos '`quit`' a `False` y el bucle se detiene. Si el usuario introduce cualquier otra cosa diferente a '`quit`', imprimiremos su entrada como un mensaje.

Este programa tiene la misma salida que el ejemplo anterior donde hemos colocado la comprobación condicional directamente en la sentencia `while`. Pero ahora tenemos un flag que nos indica si todo el programa está en estado activo, sería más sencillo añadir más test (como sentencias `elif`) para eventos que podrían causar que `active` se volviera `False`. Esto es útil en programas complicados como juegos en los que puede haber muchos eventos que podrían, cada uno, detener el programa. Cuando uno de esos eventos cause que el flag se vuelva `False`, el juego principal se saldrá del bucle, se podría mostrar un mensaje de *Game Over* y el jugador podría tener la opción de jugar de nuevo.

Utilizar `break` para salir del bucle.

Para salir de un bucle `while` inmediatamente sin ejecutar ningún código restante en el bucle, independientemente de los resultados de cualquier prueba condicional, utiliza la instrucción **`break`**. La

instrucción `break` direcciona el flujo de tu programa; puedes utilizarla para controlar qué líneas de código son ejecutadas y cuáles no, por lo que el programa ejecutará el código que tú quieras cuando tú quieras.

Por ejemplo, considera un programa que pregunte al usuario sobre los lugares que ha visitado. Podemos detener el bucle `while` en este programa llamando a `break` tan pronto como el usuario introduzca el valor '`quit`':

```
prompt = "\nPor favor, introduce el nombre de la ciudad que has visitado: "
prompt += "\n(Introduce 'quit' cuando hayas acabado.) "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print(f"Me gustaría ir a {city.title()}")
```

Un bucle que comienza con `while a True` siempre se ejecutará a menos que alcance una instrucción `break`. El bucle en este programa continúa pidiendo al usuario que introduzca los nombres de las ciudades en las que ha estado hasta que teclee '`quit`'. Cuando lo hace se ejecuta la instrucción `break` provocando que Python se salga del bucle.

```
Por favor, introduce el nombre de la ciudad que has visitado:
(Introduce 'quit' cuando hayas acabado.) New York
Me gustaría ir a New York

Por favor, introduce el nombre de la ciudad que has visitado:
(Introduce 'quit' cuando hayas acabado.) San Francisco
Me gustaría ir a San Francisco

Por favor, introduce el nombre de la ciudad que has visitado:
(Introduce 'quit' cuando hayas acabado.) quit

Process finished with exit code 0
```

NOTA: Puedes utilizar la instrucción `break` en cualquier bucle de Python. Por ejemplo, podrías usar `break` para salir de un bucle `for` que esté recorriendo una lista o un diccionario.

Usar `continue` en un bucle.

En lugar de salir de un bucle por completo sin ejecutar el resto del código, puedes usar la instrucción `continue` para volver al principio del bucle en función del resultado de un test condicional. Por ejemplo, considera un bucle que cuente los números del 1 al 10 pero sólo imprima los impares de ese rango:

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue

    print(current_number)
```

Primero inicializamos `current_number` a 0. Puesto que es menor de 10, Python entra en el bucle. Una vez dentro, incrementa su valor a 1, por lo que `current_number` ahora es 1. A continuación la sentencia `if` comprueba el módulo de `current_number` y 2. Si el módulo es cero la instrucción `continue` le dice a Python que ignore el resto del bucle vuelva al principio. Si el número actual no es divisible entre dos el resto del bucle se ejecuta y Python imprime al número actual:

```
1
3
5
7
9

Process finished with exit code 0
```

Evitar bucles infinitos.

Cualquier bucle while necesita una forma de detenerse porque si no se ejecutaría para siempre. Por ejemplo, este bucle podría contar de 1 a 5:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Pero si accidentalmente omitimos la línea `x += 1` (como se muestra a continuación), el bucle se ejecutará para siempre.

```
x = 1
while x <= 5:
    print(x)
```

Esto sucede porque el valor de `x` que comienza en 1, nunca cambiaría por lo que la condición del while siempre sería True.

Todos los programadores escriben un bucle infinito de vez en cuando, especialmente cuando los bucles de los programas tienen sutiles condiciones de salida. Si tu programa se atasca en un bucle infinito, presiona `CTRL + C` o sencillamente cierra la ventana del terminal en la que estés mostrando tu programa.

Para evitar escribir bucles infinitos, prueba cada bucle `while` y asegúrate de que el bucle se detiene cuando se espera que lo haga. Si quieres que tu programa finalice cuando el usuario introduzca cierto valor, ejecuta el programa e introduce ese valor. Si el programa no finaliza, examina la forma en la que tu programa debería hacer que el bucle se cierre. Asegúrate de que al menos una parte del programa haga que la condición del bucle sea falsa o provoca una para llegar a una instrucción `break`.

NOTA: Sublime Text y otros editores tienen embedida una ventana de salida. Esto puede dificultar detener un bucle infinito y podrías tener que cerrar el editor para finalizar el bucle. Intenta clicar en el área de salida del editor antes de presionar `CTRL + C` y tal vez puedas cancelar el bucle infinito.

EJERCICIOS - TRY IT YOURSELF.

7-4. Pizza Toppings: Escribe un bucle que solicite al usuario proporcionar una serie de ingredientes para pizza hasta que introduzca el valor ‘quit’. A medida que introduce cada ingrediente, imprime un mensaje que diga que se agregará ese ingrediente a su pizza.

7-5. Movie Tickets: Un cine tiene diferentes precios de entrada dependiendo de la edad de las personas. Si una persona es menor de 3 años, la entrada es gratis; si tiene entre 3 y 12 cuesta 10 euros; y para los mayores de 12, 15 euros. Escribe un bucle que pregunte la edad al usuario y luego infórmale del coste de su entrada.

7-6. Three Exists: Escribe diferentes versiones de los ejercicios 7-4 o 7-5, para que hagan lo siguiente al menos una vez:

- Utiliza un test condicional en la instrucción while para detener el bucle.
- Utiliza una variable activa para controlar cuánto tiempo se ejecuta el bucle.
- Utiliza una sentencia break para salir del bucle cuando el usuario introduzca el valor ‘quit’.

7-6. Infinity: Escribe un bucle que no termine nunca y ejecútalo. (Para finalizarlo pulsa CTRL + C o cierra la ventana de ejecución del bucle.)

Utilizar un bucle while con listas y diccionarios.

Hasta ahora, hemos estado trabajando sólo con una parte de información de usuario a la vez. Recibimos la entrada del usuario y luego imprimimos la salida o respondemos a ella. La próxima vez que recorramos el bucle while recibiremos otro valor de entrada y responde a eso. Pero para mantener un seguimiento de muchos partes de información de usuarios necesitaremos utilizar listas y diccionarios con nuestros bucles while.

Un bucle for es efectivo para recorrer una lista, pero no deberías modificar una lista dentro de un bucle for porque Python tendrá

problemas para realizar un seguimiento de los elementos de la lista. Para modificar una lista a medida que avances en ella, utiliza un bucle while. Utilizar un bucle while con listas y diccionarios te permite coleccionar, almacenar y organizar muchas de entradas para examinar e informar más tarde.

Mover elementos de una lista a otra.

Considera una lista de nuevos usuarios registrados pero que aún sin verificar para un sitio web. Después de verificar esos usuarios, ¿cómo podemos moverlos a una lista separada de usuarios confirmados? Una forma podría ser usar un bucle while para extraer usuarios de la lista de usuarios no confirmados a medida que los verificamos y después añadirlo a una lista separada de usuarios confirmados. Así es como se vería este código:

```
# Start with users that need to be verified,
# and an empty list to hold confirmed users.

unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verify each user until there are no more unconfirmed users.
# Move each verified user into the list of confirmed users.
while unconfirmed_users:
    current_user = unconfirmed_users.pop()

    print(f'Verifying user: {current_user.title()}')
    confirmed_users.append(current_user)

# Display all confirmed users.
print('\nThe following users have been confirmed:')
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

Comenzamos con una lista de usuarios sin confirmar (Alice, Brian y Candace) y una lista vacía para completarla con usuarios confirmados. El bucle while se ejecuta mientras que la lista de *unconfirmed_users* no esté vacía. Dentro de este bucle, la función *pop()* elimina los usuarios no verificados una vez desde el final desde el final de *unconfirmed_users*. Aquí, debido a que Candace es

la última de la lista `unconfirmed_users`, su nombre será el primero en ser eliminado, asignado a `current_users` y añadido a la lista de `confirmed_users`. Después Brian luego Alice.

Simulamos la confirmación de cada usuario imprimiendo un mensaje de verificación y añadiéndolos luego a la lista de usuario confirmados. A medida que la lista de usuarios no confirmados se reduce, la lista de usuarios confirmados crece. Cuando la lista de usuarios no confirmados esté vacía el bucle se detiene y se imprime la lista de usuarios confirmados.

```
'>>> 
      Verfiying user: Candace
      Verfiying user: Brian
      Verfiying user: Alice

      The following users have been confirmed:
      Candace
      Brian
      Alice

      Process finished with exit code 0
```

Eliminar todas las instancias de un valor específico de una lista.

En el capítulo 3 usábamos `remove()` para eliminar un valor específico de una lista. La función `remove()` nos fue bien porque el valor que estábamos buscando aparecía sólo una vez en la lista. Pero ¿qué ocurre si queremos eliminar todas las instancias de ese valor de una lista?.

Digamos que tienes una lista de mascotas con el valor `'cat'` repetido varias veces. Para eliminar todas las instancias de ese valor, puedes ejecutar un bucle `while` hasta que `'cat'` no aparezca más en la lista, como se muestra a continuación:

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

Comenzamos con una lista que contiene múltiples instancias de 'cat'. Después de imprimir la lista, Python introduce el bucle while porque encuentra el valor en la lista al menos una vez. Una vez dentro del bucle, Python elimina la primera instancia de 'cat', vuelve a la línea del while y luego vuelve a entrar en el bucle cuando encuentra que 'cat' todavía está en la lista. Elimina cada instancia de 'cat' hasta que el valor no vuelve a aparecer en la lista, momento en el que Python sale del bucle e imprime la lista nuevamente:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']

Process finished with exit code 0
```

Complejando un diccionario con entradas de usuario.

Puedes solicitar tantas entradas como necesites en cada paso a través del bucle while. Hagamos un programa de sondeo en el que a todo aquel que pase por el bucle se le solicite el nombre y la respuesta del solicitante. Almacenaremos los datos que hemos recopilado en un diccionario puesto que queremos vincular cada respuesta con un usuario en particular:

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True

while polling_active:
    # Prompt for the person's name and response
    name = input("\n¿Cuál es tu nombre?: ")
    response = input("¿Qué montaña te gustaría escalar algún día?: ")

    # Store response in the dictionary.
    responses[name] = response

    # Find out if anyone else is going to take the poll.
    repeat = input("¿Te gustaría dejar que otra persona responda (si/no)?: ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
for name, response in responses.items():
    print(f"{name} would you like to climb {response}?")
```

El programa primero define un diccionario vacío (`responses`) y configura un flag (`polling_active`) para indicar que la encuesta está activa. Mientras `polling_active` esté a `True`, Python ejecutará el código dentro del bucle.

En el bucle, se le pide al usuario que introduzca su nombre y una montaña que le gustaría escalar. Esa información es almacenada en el diccionario `responses` y se le pregunta al usuario si desea o no mantener la encuesta en ejecución.

Si introduce ‘si’, el programa entra en el bucle **`while`** de nuevo. Si introduce ‘no’ el flag `polling_active` se configura a `False`, el bucle **`while`** detiene su ejecución, y el bloque de código final muestra en pantalla los resultados de la encuesta.

Si ejecutas este programa e introduces respuestas de ejemplo, podrías obtener una salida como esta:

```
¿Cuál es tu nombre?: Eric
¿Qué montaña te gustaría escalar algún día?: Denali
¿Te gustaría dejar que otra persona responda (si/no)?: si

¿Cuál es tu nombre?: Lynn
¿Qué montaña te gustaría escalar algún día?: Devil's Thumb
¿Te gustaría dejar que otra persona responda (si/no)?: no

--- Poll Results ---
Eric would you like to climb Denali
Lynn would you like to climb Devil's Thumb

Process finished with exit code 0
```

EJERCICIOS – TRY IT YOURSELF .

7-8. Deli: Haz una lista llamada `sandwich_orders` y rellénala con nombres de varios sandwiches. Luego haz una lista vacía llamada `finished_sandwiches`. Recorre la lista de pedidos de sandwich e imprime un mensaje para cada pedido, como *He hecho tu sandwich de atún*. Cada sandwich hecho muévelo a la lista de sandwiches terminados. Después de hacer todos los sandwiches, imprime un mensaje que enumere cada sandwich que se hizo.

7-9. No Pastrami: Utiliza la lista `sandwich_orders` del ejercicio 7-8, asegúrate de que el sandwich ‘pastrami’ aparece en la lista al menos tres veces. Añade código cerca del principio de tu programa para imprimir un mensaje diciendo que la tienda de delicatessen se ha quedado sin pastrami y luego utiliza un bucle `while` para eliminar todas las coincidencias de ‘`sandwich_orders`’. Asegúrate de que los sandwiches de pastrami no se añadan a `finished_sandwiches`.

7-10. Dream Vocation: Escribe un programa que encueste a usuarios sobre sus vacaciones de ensueño. Escribe un mensaje similar a *Si pudieras visitar un lugar en el mundo, ¿dónde te gustaría ir?* Incluye un bloque de código que imprima los resultados de la encuesta.

CAPÍTULO 8. FUNCIONES.

En este capítulo aprenderás a escribir funciones, que llamarán a bloques de código diseñados para hacer un trabajo específico. Cuando quieras realizar una tarea en particular que has definido en una función, llamas a la función responsable de esa tarea. Si necesitas utilizar esa función varias veces a lo largo de tu programa, no vas a necesitar escribir de nuevo todo el código para la misma tarea una y otra vez; tan sólo tienes que llamar a la función dedicada a realizar esa tarea y la llamada le dice a Python que ejecute el código dentro de esa función. Encontrarás que el uso de las funciones hará que tus programas sean más fáciles de escribir, leer, probar y arreglar.

En este capítulo también aprenderás formas de pasar información a las funciones, cómo escribir ciertas funciones que cuyo trabajo principal es mostrar información y otras funciones diseñadas para procesar datos y devolver un valor o conjunto de valores. Por último aprenderás a almacenar funciones en archivos separados llamados módulos que te ayudarán a organizar los archivos de tu programa principal.

Definir una función.

Esta es una función sencilla llamada `greet_user()` que imprime un saludo:

```
def greet_user():
    """Muestra un sencillo saludo."""
    print('Hola')
    
greet_user()
```

Este ejemplo muestra la estructura más sencilla de una función. La línea 1 utiliza la palabra clave `def` para informar a Python que estás definiendo una función. Esta es la definición de función que le dice a Python el nombre de la función y, si procede, qué tipo de información necesita la función para hacer su trabajo. Los paréntesis contienen esa información. En este caso, el nombre de la función es `greet_user()` y no necesita información para hacer su trabajo, por lo que los paréntesis están vacíos. (En cualquier caso, los paréntesis son obligatorios.) Finalmente, la definición termina con dos puntos.

Todas las líneas indentadas a continuación de `def greet_user():` forman el cuerpo de la función. El texto en la línea 2 es llamado *docstring*, que describe lo que hace la función. Los docstrings se encierran entre triples comillas dobles o simples, que Python busca cuando se genera documentación para las funciones en tus programas.

La línea 3 es la única línea en el cuerpo de esta función, por lo que `greet_user()` sólo hace una tarea: `print('Hola')`.

Cuando quieras usar esta función, llámala. Una llamada a la función le dice a Python que ejecute el código de la función. Para llamar a la función, escribe el nombre de la función seguido de cualquier información necesaria dentro de los paréntesis, como se muestra en la última línea. Puesto que no necesita información adicional, la llamada a la función es tan simple como introducir `greet_user()`. Como se esperaba, imprime Hola:

```
Hola
Process finished with exit code 0
```

Pasar información a la función.

Modificar ligeramente la función `greet_user()` no sólo puede decirle al usuario Hola, sino que también puede saludarlo por su nombre. Para que la función haga esto introduce `username` en los paréntesis de la definición de la función en `def greet_user()`. Añadiendo `username` aquí permite que la función acepte cualquier valor de `username` que le especifiquemos. La función espera que asignemos un valor a `username` cada vez que la llamemos.. Cuando llamas a `greet_user()` puedes pasar un nombre como 'jesse' dentro de los paréntesis.

```
def greet_user(username):
    """Muestra en pantalla un saludo sencillo."""
    print(f"Hello, {username.title()}")

greet_user('jesse')
```

Al introducir `greet_user('jesse')` se llama a `greet_user()` y se da a la función la información que necesita para ejecutar la llamada a `print()`. La función admite el nombre que le hemos pasado y muestra por pantalla el saludo para ese nombre:

```
Hola, Jesse

Process finished with exit code 0
```

De la misma manera al introducir `greet_user('sarah')` llama a `greet_user()`, pasa 'sarah' e imprime 'Hello, Sarah.'. Puedes llamar a esta función siempre que quieras y pasarle el nombre que quieras para generar una salida distinta cada vez.

Argumentos y parámetros.

En la función anterior, definimos que `greet_user()` requiriera un valor para la variable `username`. Una vez llamada la función y dada la información (el nombre de una persona), imprime el saludo correcto.

La variable `username` en la definición de `greet_user()` es un ejemplo de parámetro, parte de la información que la función

necesita para trabajar. El valor 'jesse' in `greet_user('jesse')` es un ejemplo de argumento. Un argumento es la parte de información pasada en la llamada de una función a otra función. Cuando llamamos a la función, colocamos el valor con el que queremos que funcione la función entre paréntesis. En este caso el argumento 'jesse' se ha pasado a la función `greet_user()` y el valor se ha asignado al parámetro `username`.

NOTA: La gente a veces habla de argumentos y parámetros indistintamente. No te sorprendas si ves variables en **la definición de una función** referidas como argumentos o variables en **la llamada a una función** referidas como parámetros.

Ejercicios - Try it yourself.

8-1. Message: Escribe una función llamada `display_message()` que imprima un mensaje diciendo a cada uno qué está aprendiendo en este capítulo. Llama a la función y asegúrate de que el mensaje se muestra correctamente.

8-2. Favorite Book: Escribe una función llamada `favorite_book()` que acepte un parámetro, `title`. La función podría imprimir un mensaje como 'Uno de mis libros favoritos es Alicia en el país de las maravillas'. Llama a la función y asegúrate de incluir el título del libro como argumento en la llamada a la función.

Pasar argumentos.

Debido a que la definición de una función puede tener múltiples parámetros, la llamada a la función podría necesitar múltiples argumentos. Puedes pasar argumentos a tus funciones de muchas maneras. Puedes usar argumentos posicionales, que necesitan estar en el mismo orden en que fueron escritos los parámetros; argumentos de palabras clave, donde cada argumento consiste en el nombre de una variable y un valor; y listas y diccionarios de valores. Veamos cada uno de ellos por orden.

Argumentos posicionales.

Cuando llamas a una función, Python debe hacer coincidir cada argumento de la llamada a la función con un parámetro en la definición de la función. La forma más sencilla de hacerlo se basa en el orden en que se proporcionan los argumentos. Los valores coincidentes de esta manera se llaman argumentos posicionales.

Para ver cómo funcionan, consideremos una función que muestra por pantalla información acerca de mascotas. La función nos dice qué tipo de animal es cada mascota y su nombre, como se muestra a continuación:

```
def describe_pet(animal_type, pet_name):
    """Muestra información acerca de una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"Mi {animal_type} se llama {pet_name.title()}.")

describe_pet('hamster', 'harry')
```

La definición muestra que esta función necesita un tipo y un nombre de animal. Cuando llamamos a `describe_pet()`, necesitamos proporcionar un tipo de animal y un nombre, en ese mismo orden. Por ejemplo, en la llamada a la función, el argumento '`hamster`' se asigna al parámetro `animal_type` y el argumento '`harry`' al parámetro `pet_name`. En el cuerpo de la función, esos dos parámetros se utilizan para mostrar información sobre la mascota que se está describiendo.

La salida describe un hámster llamado Harry:

```
Tengo un hamster.
Mi hamster se llama Harry.

Process finished with exit code 0
```

Múltiple llamadas a función.

Puedes llamar a una función tantas veces como lo necesites. Describir una mascota diferente requiere tan sólo una llamada más para `describe_pet()`.

```
def describe_pet(animal_type, pet_name):
    """Muestra información acerca de una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"Mi {animal_type} se llama {pet_name.title()}.")\n\n
describe_pet('hamster', 'harry')
describe_pet('perro', 'nuble')
```

En la segunda llamada , pasamos los argumentos ‘perro’ y ‘**nuble**’ a `describe_pet()`. Al igual que con los argumentos utilizados anteriormente, Python hace coincidir ‘perro’ con el parámetro ‘`animal_type`’ y ‘**nuble**’ con el parámetro ‘`pet_name`’. Como anteriormente, la función hace su trabajo e imprime valores para un perro llamado **Nuble**. Ahora tenemos un hámster llamado Harry y un perro llamado **Nuble**.

```
Tengo un hamster.
Mi hamster se llama Harry.

Tengo un perro.
Mi perro se llama Nuble.

Process finished with exit code 0
```

Llamar a una función múltiples veces es una forma muy eficiente de trabajar. El código describiendo a una mascota se escribe una sola vez en la función. Luego, en cualquier momento que quieras describir a una nueva mascota, puedes llamar a la función con la nueva información de la mascota. Incluso si el código se expandiera a diez líneas, todavía podrías describir una nueva mascota en una sola línea llamando de nuevo a la función.

Puedes utilizar cualquier argumento posicional que necesites en tus funciones. Python trabaja a través de los argumentos que le proporcionemos cuando llamamos a la función y hace coincidir cada uno con el correspondiente parámetro en la definición de la función.

El orden importa en los argumentos posicionales.

Puedes obtener resultados inesperados si mezclas el orden de los argumentos en la llamada a la función si usas argumentos posicionales:

```
def describe_pet(animal_type, pet_name):
    """Muestra información acerca de una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"Mi {animal_type} se llama {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

En esta llamada a la función enumeramos primero el nombre y el tipo de animal en segundo lugar. Puesto que el argumento ‘harry’ está ordenado el primero en esta ocasión, ese valor es asignado al parámetro `animal_type`. De la misma manera, ‘hamster’ es asignada a ‘`pet_name`’. Ahora tenemos un ‘harry’ llamado ‘hamster’:

```
Tengo un harry.
Mi harry se llama Hamster.

Process finished with exit code 0
```

Si obtienes resultados divertidos como este, asegúrate de que el orden de los argumentos en tu función coincide con el orden de los parámetros en la definición de la función.

Argumentos de palabras clave.

Un argumento de palabra clave, es un par nombre-valor que pasas a la función. Asocias directamente el nombre y el valor en los argumentos, así que cuando pasas el argumento a la función, no hay confusión (no terminarás con un `harry` llamado `hamster`). Los argumentos de palabras clave te liberan de tener que preocuparte de ordenar correctamente los argumentos en la llamada a la función y aclaran el rol de cada valor en la llamada a la función.

Reescribamos `pets.py` utilizando argumentos de palabras clave para llamar a `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Muestra información acerca de una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"Mi {animal_type} se llama {pet_name.title()}.")  
💡
describe_pet(animal_type='hamster', pet_name='harry')
```

La función `describe_pet()` no ha cambiado. Pero cuando llamamos a la función decimos explícitamente a Python qué parámetro debe emparejar con cada argumento. Cuando Python lea la llamada a la función sabrá asignar el argumento ‘hamster’ al parámetro `animal_type` y el argumento ‘harry’ a `pet_name`. La salida correcta muestra que tenemos un hamster llamado Harry.

El orden de los argumentos de palabras clave no importa porque Python sabe dónde tiene que ir cada valor. Las dos siguientes llamadas a la función son equivalentes:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

NOTA: Cuando uses argumentos de palabras clave, asegúrate de utilizar el nombre exacto de los parámetros en la definición de la función.

Valores por defecto.

Cuando escribimos una función puedes definir un valor por defecto para cada parámetro. Si un argumento para un parámetro se proporciona en la llamada a la función, Python usa el valor del argumento. Si no, utiliza el valor por defecto del parámetro. Así que cuando defines un valor por defecto para un parámetro, puedes excluir el argumento correspondiente que normalmente se escribiría en la llamada a la función. Utilizar valores por defecto puede simplificar tus llamadas a la función y aclarar las formas en que tus funciones se utilizan normalmente.

Por ejemplo, si observas que la mayoría de las llamadas a la función `describe_pet()` se utilizan para describir perros, puedes establecer el valor predeterminado de `animal_type()` en ‘perro’. Ahora cualquier llamada a `describe_pet()` para un perro, puede omitir esa información:

```
def describe_pet(pet_name, animal_type='perro'):
    """Muestra información acerca de una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"Mi {animal_type} se llama {pet_name.title()}.")  
  
describe_pet(pet_name='nuble')
```

Cambiamos la definición de `describe_pet()` para incluir un valor por defecto, ‘perro’, para `animal_type`. Ahora cuando la función es llamada sin especificar `animal_type`, Python sabe que tiene que usar el valor ‘perro’ para ese parámetro:

```
Tengo un perro.  
Mi perro se llama Nuble.  
  
Process finished with exit code 0
```

Observa que el orden de los parámetros en la definición de la función ha cambiado. Debido a que el valor por defecto no es necesario para especificar un tipo de animal como un argumento, el único argumento que queda en la llamada a la función es el nombre de la mascota. Python todavía interpreta esto como un argumento posicional, por lo que si la función es llamada sólo con el nombre de la mascota, ese argumento coincidirá con el primer parámetro ordenado en la definición de la función. Esta es la razón por la que el primer parámetro debe ser `pet_name`.

La forma más sencilla para usar ahora esta función es proporcionar sólo el nombre del perro en la llamada a la función.

```
describe_pet(pet_name='nuble')
```

Esta llamada a la función tendría el mismo resultado que en el ejemplo anterior. El único argumento proporcionado es ‘nuble’ por lo que coincide con el primer parámetro en la definición de *pet_name*. Puesto que no se proporciona un argumento para *animal_type*, Python utiliza el valor por defecto ‘perro’.

Para describir a un animal que no sea un perro podrías usar la llamada a la función así:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Debido a que se proporciona un argumento explícito para *animal_type*, Python ignora el valor por defecto del parámetro.

NOTA: Cuando uses valores por defecto, cualquier parámetro con un valor por defecto debe aparecer después de todos los parámetros que no tienen valores predeterminados.

Llamadas a funciones equivalentes.

Debido a que los argumentos posicionales, los de palabras clave y los valores predeterminados se pueden usar juntos, a menudo tendrás varias formas equivalentes de llamar a una función. Considera la siguiente definición para *describe_pet()* con un valor por defecto proporcionado:

```
def describe_pet(pet_name, animal_type='perro'):
```

Con esta definición siempre necesitas proporcionar un argumento para *pet_name*, y este valor se puede proporcionar usando al formato posicional o de palabra clave. Si el animal descrito no es un perro, se debe incluir un argumento en la llamada a *animal_type*, y este argumento también puede ser especificado utilizando el formato posicional o de palabra clave.

Todas las llamadas siguientes podrían funcionar para esta función:

```
# Un perro llamado Willie.  
describe_pet('willie')  
describe_pet(pet_name='willie')  
  
# Un hamster llamado Harry.  
describe_pet('harry', 'hámster')  
describe_pet(pet_name='harry', animal_type='hámster')  
describe_pet(animal_type='hámster', pet_name='harry')
```

Cada una de estas llamadas a la función tendría el mismo resultado que los anteriores ejemplos.

NOTA: Realmente no importa qué estilo de llamada utilices. Siempre que tus llamadas a la función produzcan el mismo resultado, utiliza sencillamente el estilo que te resulte más fácil de entender.

Evitar errores de argumento.

Cuando empiezas a usar funciones, no te sorprendas si encuentras errores de argumentos no coincidentes. Los argumentos no coincidentes se producen cuando se proporciona a una función más o menos argumentos de los que necesita para hacer su trabajo. Por ejemplo, esto es lo que sucedería si intentamos llamar a `describe_pet()` sin argumentos:

```
def describe_pet(animal_type, pet_name):  
    """Muestra información acerca de una mascota."""|  
    print(f"\nTengo un {animal_type}.")  
    print(f"Mi {animal_type} se llama {pet_name.title()}.")  
  
describe_pet()
```

Python reconoce que falta información en la llamada a la función y el rastreo nos dice que:

```
Traceback (most recent call last):
  File "/home/usuario/PycharmProjects/pythonCrashCourse/capitulo_8_funciones/pets_5.py", line 6, in <module>
    describe_pet()
TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'

Process finished with exit code 1
```

El seguimiento nos dice la localización del problema, permitiéndonos revisar y ver si algo ha salido mal en la nuestra llamada a la función. Se escribe la llamada a la función no válida para que la veamos. El seguimiento nos dice que la llamada ha perdido dos argumentos e informa los nombres de los argumentos perdidos. Si esta función estuviera en un archivo separado, probablemente podríamos reescribir la llamada correctamente sin tener que abrir ese fichero y leer el código de la función.

Python es útil porque lee el código de la función por nosotros y nos dice los nombres de los argumentos que necesitamos proporcionar. Esta es otra motivación para dar a tus variables y funciones nombres descriptivos. Si lo haces, los mensajes de error de Python serán más útiles para ti y para cualquiera que pueda usar tu código.

Si proporcionas demasiados argumentos, deberías obtener un seguimiento similar que pueda ayudarte a hacer coincidir correctamente la llamada a la función con la definición de la función.

EJERCICIOS – TRY IT YOURSELF.

8-3. T-Shirt: Escribe una función llamada *make_shirt()* que acepte una talla y el texto de un mensaje que debe ser impreso en la camiseta. La función podría imprimir una frase resumiendo la talla de la camiseta y el mensaje impreso en ella.

Llama a la función usando argumentos posicionales para hacer la camiseta. Llama a la función una segunda vez utilizando argumentos de palabras clave.

8-4. Large Shirts: Modifica la función *make_shirt()* para que las camisetas sean large por defecto con un mensaje en el que se lea *I love Python*. Haz una talla *large* y otra *medium* con el mensaje por

defecto y una camiseta de cualquier talla con un mensaje diferente.

8-5. Cities: Escribe una función llamada `describe_city()` que acepte el nombre de una ciudad y su país. La función debe imprimir una frase simple como *Reykjavik está en Islandia*. Da un valor por defecto al parámetro país. Llama a tu función con tres ciudades diferentes y al menos una de las cuales no esté en el país por defecto.

Valores de retorno.

Una función no siempre tiene que mostrar una salida directamente. En su lugar, puede procesar algunos datos y devolver un valor o un conjunto de valores. El valor devuelto por la función se llamada valor de retorno. La instrucción `return` toma un valor de dentro de una función y lo envía de vuelta a la línea que ha llamado a esa función. Los valores de retorno te permiten mover gran parte del trabajo pesado de tu programa a tus funciones, lo que puede simplificar el cuerpo de tu programa.

Devolver un valor simple.

Echemos un vistazo a una función que toma un nombre y un apellido y devuelve un nombre completo con un formato ordenado.

```
def get_formatted_name(first_name, last_name):
    """ Devuelve un nombre completo con un formato ordenado. """
    full_name = f'{first_name} {last_name}'
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

La definición de `get_formatted_names()` toma como parámetros un nombre y un apellido. Combina esos dos valores, añade un espacio entre ellos y asigna el resultado a `full_name`. El valor `full_name` es convertido a title case, y luego es devuelto a la llamada de la línea `return`.

Cuando llamas a una función con valores de retorno, debes proporcionar una variable a la que se le pueda asignar el valor devuelto. En este caso, el valor devuelto es asignado a la variable *musician*. La salida muestra un nombre con un formato ordenado compuesto por las partes del nombre de una persona.

```
Jimi Hendrix  
Process finished with exit code 0
```

Esto puede parecer mucho trabajo para obtener un nombre bien formateado cuando podríamos haber escrito `print('Jimi Hendrix')`. Pero cuando consideres trabajar con un programa grande que necesita almacenar muchos nombres y apellidos por separado, las funciones como `get_formatted_name()` se vuelven muy útiles. Almacena nombres y apellidos por separado y a continuación llamas a esta función siempre que deseas mostrar un nombre completo.

Hacer que un argumento sea opcional.

A veces tiene sentido hacer que un argumento sea opcional para que las personas que usan la función puedan optar por proporcionar información adicional sólo si quisieran hacerlo. Puede utilizar valores por defecto para hacer argumentos opcionales.

Por ejemplo, digamos que queremos expandir `get_formatted_names()` para manejar también los segundos nombres. Un primer intento de incluir segundos nombres se podría parecer a esto:

```
def get_formatted_name(first_name, middle_name, last_name):  
    """ Devuelve un nombre completo con un formato ordenado. """  
    full_name = f"{first_name} {middle_name} {last_name}"  
    return full_name.title()  
  
musician = get_formatted_name('jonh', 'lee', 'hooker')  
print(musician)
```

Esta función trabaja cuando le damos un primer y segundo nombre y los apellidos. La función toma las tres partes del nombre y luego construye una cadena con ellos. La función añade espacios donde se

necesitan y convierte todo el nombre con las primeras letras a mayúsculas.

```
Jimi Hendrix  
Process finished with exit code 0
```

Pero no siempre se necesitan los segundos nombres y esta función, tal y como está escrita, no funcionaría si intentaras llamarla sólo con un nombre y un apellido. Para hacer que el segundo nombre sea opcional podemos dar al argumento *middle_name* un valor por defecto vacío e ignorarlo a menos que el usuario proporcione un valor. Para hacer que *get_formatted_names()* funcione sin un segundo nombre, configuraremos el valor por defecto de *middle_name* a una cadena vacía y lo desplazaremos al final de la lista de parámetros:

```
def get_formatted_name(first_name, last_name, middel_name=' '):  
    """ Devuelve un nombre completo con un formato ordenado. """  
    if middel_name:  
        full_name = f"{first_name} {middel_name} {last_name}"  
    else:  
        full_name = f"{first_name} {last_name}"  
    return full_name.title()  
  
musician = get_formatted_name('jimi', 'hendrix')  
print(musician)  
  
musician = get_formatted_name('jonh', 'lee', 'hooker')  
print(musician)
```

En este ejemplo, el nombre se construye de tres posibles partes. Puesto que siempre hay un nombre y un apellido, esos parámetros se ordenan primero en la definición de la función. El segundo nombre es opcional, por lo que se coloca el último en la definición y su valor por defecto es una cadena vacía.

En el cuerpo de la función comprobamos si se ha proporcionado un segundo nombre. Python interpreta las cadenas no vacías como True por lo que *middle_name* se evalúa a True si hay un argumento para el segundo nombre en la llamada a la función. Si se proporciona un segundo nombre, el primer y segundo nombre y el apellido se

combinan para formar el nombre completo. Después se cambia este nombre para que las primeras letras de cada nombre sean mayúsculas y se devuelve a la línea que llama a la función donde se asigna a la variable *musician* y se imprime. Si no se proporciona una segundo nombre, la cadena vacía hace que el test if falle y se ejecute el bloque else. El nombre completo se hace sólo con nombre y apellido, el nombre formateado se devuelve para llamar a la línea donde es asignado a *musician* y se imprime.

Llamar a esta función con un nombre y un apellido se fácil. Sin embargo, si estuviéramos usando un segundo nombre, tenemos que asegurarnos de que ese segundo nombre sea el último argumento pasado para que Python haga coincidir correctamente a los argumentos posicionales.

Esta versión modificada de nuestra función, trabaja sólo con primer nombre y apellido e igualmente lo hace con personas que tengan también un segundo nombre:

```
Jimi Hendrix
Process finished with exit code 0
```

Los valores opcionales permiten que las funciones manejen una amplia gama de casos de uso mientras que permite que las llamas a funciones sigan siendo lo más simples posible.

Devolver un diccionario.

Un función puede devolver cualquier tipo de valor que necesites, incluyendo estructuras de datos más complejas como listas y diccionarios. Por ejemplo, la siguiente función toma pares de una nombre y devuelve un diccionario que representa a una persona:

```
def build_person(first_name, last_name):
    """ Devuelve un diccionario de información sobre una persona. """
    person = {'first': first_name, 'last': last_name}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

La función `build_person()` toma un nombre y un apellido y pone esos valores dentro de un diccionario. El valor de `first_name` se almacena con la clave '`first`', y el valor de `last_name` se almacena con la clave '`last`'. El diccionario completo representando a una persona se devuelve con `return`. El valor devuelto se imprime con las dos partes originales de la información textual ahora almacenadas en un diccionario.

```
{'first': 'jimi', 'last': 'hendrix'}  
Process finished with exit code 0
```

Esta función toma una simple información textual y la coloca en una estructura de datos más significativa que te permite trabajar con la información más allá de una simple impresión. Las cadenas '`jimi`' y '`hendrix`' están ahora etiquetadas como primer nombre y apellido. Puedes extender fácilmente esta función para que acepte valores opcionales como un segundo nombre, una edad, una ocupación o cualquier otra información que quieras almacenar acerca de una persona. Por ejemplo, el siguiente cambio también te permite almacenar la edad de una persona:

```
def build_person(first_name, last_name, age=None):  
    """ Devuelve un diccionario con información acerca de una persona. """  
    person = {'first': first_name, 'last': last_name}  
    if age:  
        person['age'] = age  
    return person  
  
musician = build_person('jimi', 'hendrix', age=27)  
print(musician)
```

Añadimos el nuevo parámetro opcional `age` a la definición de la función y asignamos al parámetro el valor especial `None`, que es utilizado cuando la variable no tiene ningún valor asignado. Puedes pensar en `None` como un marcador de valor. En una prueba condicional, `None` se evalúa a `False`. Si la llamada a la función incluye un valor para `age`, ese valor es almacenado en el diccionario. Esta función siempre almacena el nombre de una persona, pero puede ser modificada para almacenar cualquier otra información que quieras acerca de una persona.

Usar una función con un bucle while.

Puedes usar funciones con todas las estructuras que has aprendido hasta ahora. Por ejemplo, usemos la función `get_formatted_name()` con un bucle `while` para saludar más formalmente a los usuarios. Aquí tenemos un primer intento de saludar a las personas usando su nombre y su apellido:

```
def get_formatted_name(first_name, last_name):
    """ Devuelve un nombre completo bien formateado. """
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Esto es un bucle infinito!
while True:
    print("\nPor favor, introduce tu nombre: ")
    f_name = input('First name: ')
    l_name = input('Last name: ')

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHola, {formatted_name}!")
```

Para este ejemplo, hemos utilizado una versión simple de `get_formatted_name()` que no incluye segundos nombres. El bucle **while**, pide al usuario que introduzca su nombre y nosotros solicitamos su nombre y apellido por separado.

Pero hay un problema con este bucle **while**: No hemos definido una condición de terminación. ¿Dónde ponemos esa condición cuando preguntamos por una serie de entradas? Queremos que el usuario pueda salir tan fácil como sea posible, por lo que cada mensaje debería ofrecer una forma de salir. La sentencia **break** ofrece una forma directa de salir del bucle en cualquier momento:

```

def get_formatted_name(first_name, last_name):
    """ Devuelve un nombre completo bien formateado. """
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Esto es un bucle infinito!
while True:
    print("\nPor favor, introduce tu nombre: ")
    print('(Introduce "q" para salir)')

    f_name = input('First name: ')
    if f_name == 'q':
        break

    l_name = input('Last name: ')
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHola, {formatted_name}!")

```

Añadimos un mensaje que informa al usuario cómo salir y luego salimos del bucle si el usuario introduce el valor de salida en cualquier momento. Ahora el programa continuará saludando personas hasta que alguien introduzca ‘q’ para su nombre:

```

Por favor, introduce tu nombre:
(Introduce "q" para salir)
First name: eric
Last name: matthes

Hola, Eric Matthes!

Por favor, introduce tu nombre:
(Introduce "q" para salir)
First name: q

Process finished with exit code 0

```

EJERCICIOS - TRY IT YOURSELF.

8-6. City Names: Escribe una función llamada `city_country()` que tome el nombre de una ciudad y de su país. La función devolverá una cadena formateada como esta:

“Santiago, Chile”

Llama tu función con al menos tres pares ciudad-país, e imprime los valores devueltos.

8-7. Album. Escribe una función llamada `make_album()` que construya un diccionario que describa un álbum de música. La función debe tomar un nombre de un artista, el título de un álbum y tiene que devolver un diccionario que contenga esas dos partes de información. Utiliza la función para hacer tres diccionarios que representen diferentes álbumes. Imprime cada valor devuelto para mostrar que los diccionarios almacenan correctamente la información del álbum.

Utiliza `None` para añadir un parámetro para que `make_album()` te permita almacenar el número de canciones en un álbum. Si la línea de llamada incluye un valor para el número de canciones, agrega ese valor al diccionario del álbum. Haz al menos una nueva llamada a la función que incluya el número de canciones en un álbum.

8-8. User Albums. Comienza con el programa del ejercicio 8-7. Escribe un bucle `while` que permita a los usuarios introducir el álbum de un artista y el título. Una vez que tienes esa información, llama a `make_album()` con la entrada del usuario e imprime el diccionario que se ha creado. Asegúrate de incluir un valor `quit` en el bucle `while`.

Pasar una lista.

Con frecuencia encontrarás muy útil pasar una lista a una función, ya sea una lista de nombres, números u objetos más complejos como diccionarios. Cuando pasas una lista a una función, la función obtiene acceso directo al contenido de la lista. Usemos funciones para hacer que el trabajo con listas sea más eficiente.

Digamos que tenemos una lista de usuarios y queremos imprimir un saludo a cada uno. El siguiente ejemplo envía una lista de nombres

a una función llamada `greet_users()`, que saluda a cada persona de la lista individualmente:

```
def greet_users(names):
    """ Imprime un simple saludo a cada usuario de la lista. """
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['hanna', 'ty', 'margot']
greet_users(usernames)
```

Definimos `greet_users()` que espera una lista de nombres que asigna al parámetro `nombres`. La función recorre la lista recibida e imprime un saludo a cada usuario. Definimos la lista de usuarios `usernames`, y luego la pasamos a `greet_users()` en nuestra llamada a la función:

```
Hello, Hanna!
Hello, Ty!
Hello, Margot!

Process finished with exit code 0
```

Esta es la salida que esperábamos. Cada usuario ve un saludo personalizado y puedes llamar a la función cada vez que quieras saludar a un grupo específico de usuarios.

Modificar una lista en una función.

Cuando pasas una lista a una función, la función puede modificar la lista. Cualquier cambio que hagas en la lista dentro del cuerpo de la función será permanente, permitiéndote trabajar eficientemente incluso cuando estés trabajando con una gran cantidad de datos.

Considera una compañía que crea modelos de impresión en 3D de diseños que envía a los usuarios. El diseño que necesita para imprimir está almacenado en una lista y antes de ser impreso lo

mueven a una lista separada. El siguiente código hace eso sin utilizar funciones:

```
# Comienza con algunos diseños que necesitan ser impresos
unprinted_desings = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

# Simula la impresión de cada diseño hasta que no quede ninguno.
# Mueve cada diseño a completed_models después de imprimirlo.
while unprinted_desings:
    current_desing = unprinted_desings.pop()
    print(f"Printing model: {current_desing}")
    completed_models.append(current_desing)

# Muestra todos los modelos completados
print("\nLos siguientes modelos han sido impresos: ")
for completed_model in completed_models:
    print(completed_model)
```

Este programa empieza con una lista de diseños que necesitan ser impresos y una lista vacía llamada *completed_models* para mover cada diseño después de ser impreso. Mientras los diseños permanezcan en *unprinted_desings*, el bucle while simula la impresión de cada diseño eliminando un diseño del final de la lista, almacenándolo en *current_design* y mostrando por pantalla un mensaje indicando que el diseño actual ha sido impreso. Luego añade el diseño a la lista de modelos completados. Cuando el bucle termina, se muestra por pantalla la lista de diseños que se ha impreso:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

Los siguientes modelos han sido impresos:
dodecahedron
robot pendant
phone case

Process finished with exit code 0
```

Podemos reorganizar este código escribiendo dos funciones, cada una de las cuales realiza una función específica. La mayor parte del código no cambiará; solamente lo estamos haciendo más cuidadosamente estructurado. La primera función se encargará de imprimir el diseño y la segunda resumirá las impresiones que se han realizado:

```
def print_models(unprinted_designs, completed_models):
    """ Simula la impresión de cada diseño hasta que no quede ninguno.
        Mueve cada diseño a completed_models después de imprimirlo. """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """ Muestra todos los modelos que se ha impreso. """
    print("\nLos siguientes modelos han sido impresos: ")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Definimos la función `print_models()` con dos parámetros: una lista de diseños que necesita ser impresa y una lista de modelos completados. Dando esas dos listas, la función simula imprimir cada diseño vaciando la lista de diseños no impresos y llenando la lista de modelos completados. Luego definimos la función `show_completed_models()` con un parámetro: la lista de modelos completados. Dando esta lista, `show_completed_models()` muestra por pantalla cada modelo que ha sido impreso.

Este programa tiene la misma salida que la versión sin funciones, pero el código está mucho mejor organizado. El código que hace la mayoría del trabajo se ha movido a dos funciones separadas, lo que hace que la parte principal del programa sea más fácil de

entender. Mira el cuerpo del programa para ver lo fácil que es entender lo que está haciendo este programa:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Configuramos una lista de diseños no impresos y una lista que contendrá los modelos completados. Luego, puesto que ya hemos definido nuestras dos funciones, todo lo que tenemos que hacer es llamarlas y pasárselas los argumentos correctos. Llamamos a *print_models()* y pasamos las dos listas que necesitamos; como se esperaba, *print_models()* simula imprimir los diseños. Luego llamamos a *show_completed_models()* y pasamos la lista de modelos completados para que pueda informar de los modelos que se han impreso. Los nombres de las funciones descriptivas permiten a otros leer el código y entenderlo incluso sin comentarios.

Este programa es más fácil de extender y mantener que la versión sin funciones. Si necesitamos imprimir más diseños después, simplemente llamamos de nuevo a *print_models()*. Si observamos que el código de impresión necesita ser modificado, podemos cambiar el código una vez y nuestros cambios tendrán lugar en todos los lugares donde se llama a la función. Esta fórmula es más eficiente que tener que actualizar código de forma separada en todas las partes del programa.

Este ejemplo también demuestra la idea de que cada función puede hacer un trabajo específico. La primera función imprime cada diseño y la segunda muestra los modelos completados. Esto es más beneficioso que usar una función que haga ambos trabajos. Si estás escribiendo una función y observas que la función hace demasiadas tareas, intenta dividir el código en dos funciones. Recuerda que siempre puedes llamar a una función desde otra función, lo que te ayudará al dividir una tarea compleja en una serie de pasos.

Impedir que una función modifique una lista.

A veces querremos evitar que una función modifique una lista. Por ejemplo, digamos que empezamos con una lista de diseños no

impresos y escribimos una función para moverlos a una lista de modelos completados, como en el ejemplo anterior. Puedes decidir que, aunque hayas impreso todos los diseños, quieras conservar la lista original de diseños no impresos para tus registros.

Pero puesto que has sacado todos los nombres de `unprinted_designs`, la lista está ahora vacía y esta lista vacía es la única versión que tenemos de la lista; la original se marchó. En este caso, podemos solucionar el problema pasando a la función una copia de la lista, no la original. Los cambios que la función haga a la lista, afectarán sólo a la copia, dejando intacta la lista original.

Puedes enviar una copia de la lista a la función de esta manera:

```
function_name(list_name[:])
```

La notación `slice [:]` hace una copia de la lista para enviarla a la función. Si no queremos vaciar la lista de diseños no impresos en `printing_models.py`, podemos llamar a `print_models()` así:

```
print_models(unprinted_designs[:], completed_models)
```

La función `print_models()` puede hacer este trabajo porque todavía recibe los nombres de todos los diseño no impresos. Pero esta vez utiliza una copia de la lista original de diseños no impresos, no la lista `unprinted_desgins` actual. La lista `completed_models`, se llenará con los nombres de modelos impresos como hicimos anteriormente, pero la lista original de diseños no impresos no se verá afectada por la función.

Aunque puedes conservar el contenido de una lista pasando una copia de ella a tus funciones, puedes pasar la lista original a tus funciones, a menos que tengas una razón específica para pasarle una copia. Es más eficiente que una función trabaje con una lista existente para evitar usar el tiempo y la memoria necesarios para hacer una copia separada, especialmente cuando se trabaja con listas grandes.

EJERCICIOS - TRY IT YOURSELF.

8-9. Messages: Haz una lista que contenga una serie de mensajes de texto cortos. Pasa la lista a una función llamada `show_messages()`, que imprima cada mensaje de texto.

8-10. Sending Messages: Empieza con una copia de tu programa del ejercicio 8-9. Escribe una función llamada `send_messages()` que imprima cada mensaje de texto y mueva cada mensaje a una nueva lista llamada `sent_messages` a medida que se imprime. Después de llamar a la función, imprime ambas listas para asegurarte de que los mensajes se hayan movido correctamente.

8-11. Archived Messages: Empieza con tu trabajo del ejercicio 8-10. Llama a la función `send_messages()` con una copia de la lista de mensajes. Después de llamar a la función, imprime ambas listas para mostrar que la lista original ha conservado sus mensajes.

Pasar un número arbitrario de argumentos.

A veces no sabremos de antemano cuántos argumentos necesita aceptar una función. Afortunadamente, Python permite a una función colecciónar un número arbitrario de argumentos desde la instrucción de llamada.

Por ejemplo, considera una función que fabrique una pizza. Necesitará aceptar un número de ingredientes, pero no puedes saber de antemano cuántos ingredientes querrá una persona. La función del siguiente ejemplo tiene un parámetro, `*toppings`, pero este parámetro, pero este parámetro colecciona tantos argumentos como le proporcione la línea de la llamada:

```
new*
def make_pizza(*toppings):
    """Imprime la lista de ingredientes que ha sido solicitada."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

El asterisco en el parámetro llamado `*toppings`, le dice a Python que fabrique una lista vacía llamada `toppings` y empaquete los valores que reciba dentro de la tupla. La llamada a `print()` en el cuerpo de una función, muestra que Python puede manejar una llamada a la función con un valor y otra con tres valores. Trata las diferentes llamadas de manera similar. Observa que Python empaqueta los argumentos en una tupla, incluso si la función sólo recibe un valor:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')

Process finished with exit code 0
```

Ahora podemos reemplazar la llamada a `print()` con un bucle que recorra la lista de ingredientes y describa la pizza que se está pidiendo:

```
def make_pizza(*toppings):
    """ Resumir la pizza que estamos a punto de hacer."""
    print("\nHaciendo una pizza con los siguientes ingredientes: ")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

La función responde de forma adecuada, reciba uno o tres valores:

```
Haciendo una pizza con los siguientes ingredientes:
- pepperoni

Haciendo una pizza con los siguientes ingredientes:
- mushrooms
- green peppers
- extra cheese

Process finished with exit code 0
```

Esta sintaxis funciona independientemente de cuántos argumentos reciba la función.

Mezclando argumentos posicionales y arbitrarios.

Si quieres que una función acepte varios tipos diferentes de argumentos, el parámetro que acepta un número arbitrario de argumentos se debe colocar el último en la definición de la función. Python hace coincidir primero los argumentos posicionales y de palabras clave y luego recopila los argumentos restantes en el final del parámetro.

Por ejemplo, si la función necesita obtener un tamaño para la pizza, ese parámetro debe ir antes del parámetro `*toppings`:

```
def make_pizza(size, *toppings):
    """ Resume la pizza que estamos a punto de hacer."""
    print(f"\nHaciendo una pizza de tamaño {size}-inch con los siguientes ingredientes: ")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

En la definición de la función, Python asigna el primer valor que recibe al parámetro `size`. Todos los otros valores que van después son almacenados en la tupla `toppings`. La llamada a la función incluye un argumento para el tamaño primero, seguido de tantos `toppings` como se necesiten.

Ahora cada pizza tiene un tamaño y un número de ingredientes, y cada parte de la información se imprime en su lugar apropiado, mostrando primero el tamaño y después los ingredientes:

```
Haciendo una pizza de tamaño 16-inch con los siguientes ingredientes:
- pepperoni

Haciendo una pizza de tamaño 12-inch con los siguientes ingredientes:
- mushrooms
- green peppers
- extra cheese

Process finished with exit code 0
```

NOTA: A menudo verás el nombre genérico del parámetro `*args`, que recopila argumentos posicionales arbitrarios como este.

Utilizando argumentos de palabras clave arbitrarios.

A veces querrás aceptar un número arbitrario de argumentos, pero no sabrás de antemano qué tipo de información se pasará a la función. En este caso puedes escribir funciones que acepten tantos pares clave-valor como proporcione la instrucción de llamada. Un ejemplo puede ser la creación de perfiles de usuario: sabes que obtendrás información del usuario, pero no sabes qué tipo de información vas a recibir. La función `build_profile()` del siguiente ejemplo, siempre toma un primer nombre y un apellido, pero también acepta un número arbitrario de argumentos de palabras clave:

```
new*
def build_profile(first, last, **user_info):
    """ Construye un diccionario que contiene todo lo que sabemos de un usuario. """
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')

print(user_profile)
```

La definición de `build_profile()` espera un nombre y un apellido y luego permite al usuario pasar tantos pares nombre-valor como quiera. El doble asterisco antes del valor `**user_info` hace que Python cree un diccionario vacío llamado `user_info` y empaquete cualquier par nombre-valor que reciba en ese diccionario.

En el cuerpo de `build_profile()`, añadimos el nombre y el apellido al diccionario `user_info` porque siempre recibiremos esas dos partes de información del usuario y no se han colocado en el diccionario todavía. Luego devolvemos el diccionario `user_info` a la línea de llamada de la función.

Llamamos a `build_profile()`, pasando el nombre '`albert`', el apellido '`einstein`' y los dos pares clave-valor

location='princeton' y field='physics'. Asignamos el perfil devuelto a `user_profile` e imprimimos `user_profile`:

```
{'location': 'princeton', 'field': 'physics', 'first_name': 'albert', 'last_name': 'einstein'}  
Process finished with exit code 0
```

El diccionario devuelto contiene el nombre y el apellido del usuario y, en este caso, también la ubicación y el campo de estudio. La función trabajará sin importar cuántos pares clave-valor se proporcionen en la llamada a la función.

Puedes mezclar posicionales, palabras clave y valores arbitrarios de muchas formas diferentes cuando esribas tus propias funciones. Es de utilidad saber existen todos esos tipos de argumentos porque los verás con frecuencia cuando empieces a leer código de otras personas. Se necesita práctica para aprender a usar los diferentes tipos correctamente y saber cuándo usar cada uno. Por ahora, recuerda usar el enfoque más simple que haga el trabajo. A medida que progreses aprenderás a usar el enfoque de manera más eficiente cada vez.

NOTA: Con frecuencia verás el parámetro llamado `**kwargs` usado para colecciones de argumentos de palabras clave sin especificar.

EJERCICIOS – TRY IT YOURSELF.

8-12. Sandwiches: Escribe una función que acepte una lista de elementos que una persona quiera en un sándwich. La función debe tener un parámetro que coleccione tantos elementos como proporcione la llamada a la función y debería imprimir un resumen del sándwich que se ha pedido. Llama a la función tres veces utilizando diferentes números de argumentos cada vez.

8-13. User Profile: Empieza con una copia de `user_profile.py` del ejemplo anterior. Construye un perfil de ti mismo, llamado `build_profile()`, utilizando tu nombre, apellidos y otros tres pares clave-valor que te describan.

8-14. Cars: Escribe una función que almacene información sobre un coche en un diccionario. La función siempre debe recibir un

fabricante y un nombre de modelo de coche. Debe aceptar un número arbitrario de palabras clave. Llama a la función con la información requerida con la información requerida y otros dos pares clave-valor como color o una función opcional. Tu función podría trabajar con una llamada como esta:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Imprime el diccionario devuelto para asegurarte de que toda la información se ha almacenado correctamente.

Almacenar las funciones en módulos.

Una ventaja de las funciones es la forma en que separan los bloques de código de tu programa principal. Usando nombres descriptivos en tus funciones, tu programa principal será mucho más fácil de seguir. Puedes ir un paso más allá almacenando tus funciones en archivos separados llamados módulos y luego importar esos módulos a tu programa principal. Una instrucción **import** le indica a Python que haga que el código de un módulo esté disponible en el archivo de programa que se está ejecutando actualmente.

Almacenar tus funciones en archivos separados, te permite ocultar los detalles del código de tu programa y centrarte en su lógica de nivel superior. También te permite reutilizar funciones en diferentes programas. Cuando almacenas funciones en archivos separados puedes compartir esos archivos con otros programadores sin tener que compartir todo el programa. Saber cómo importar funciones también te permite usar librerías de funciones que otros programadores hayan escrito.

Existen muchas formas de importar un módulo, y lo veremos brevemente.

Importar un módulo completo.

Para empezar a importar funciones, primero necesitamos crear un módulo. Un módulo es un fichero terminado en .py que contiene el código que quieras importar a tu programa. Hagamos un módulo que contenga la función `make_pizza()`. Para hacer este módulo eliminaremos todo lo del archivo `pizza.py` salvo la función `make_pizza()`:

```
def make_pizza(size, *toppings):
    """ Resumir la pizza que estamos a punto de hacer."""
    print(f"\nHaciendo una pizza de tamaño {size}-inch con los siguientes ingredientes: ")
    for topping in toppings:
        print(f"- {topping}")
```

Ahora, hagamos un archivo separado llamado `making_pizzas.py` en el mismo directorio que `pizza.py`. Este fichero importa el módulo que acabamos de crear y hace dos llamadas a `make_pizza()`:

```
import pizza_2

pizza_2.make_pizza(16, 'pepperoni')
pizza_2.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Cuando Python lee este archivo, la línea `import pizza` le dice a Python que abra el archivo `pizza.py` y copie todas las funciones desde él a este programa. Ahora mismo no ves que el código esté siendo copiado entre los archivos porque Python copia el código entre bastidores justo antes de ejecutar el programa. Todo lo que necesitas saber es que cualquier función definida en `pizza.py` estará disponible en `making_pizzas.py`.

Para llamar a una función desde un módulo importado, introduce el nombre del módulo importado, `pizza`, seguido del nombre de la función, `make_pizza()`, separados por un punto. Este programa produce la misma salida que el programa original que no importaba ningún módulo:

```
Haciendo una pizza de tamaño 16-inch con los siguientes ingredientes:  
- pepperoni  
  
Haciendo una pizza de tamaño 12-inch con los siguientes ingredientes:  
- mushrooms  
- green peppers  
- extra cheese  
  
Process finished with exit code 0
```

Este primer enfoque para importar, en el que simplemente escribes **import** seguido del nombre del módulo, hace que cada función del módulo esté disponible en tu programa. Si usas este tipo de sentencia **import** para importar un nombre de módulo completo llamado *module_name.py*, cada función de este módulo estará disponible a través de la siguiente sintaxis:

```
module_name.function_name()
```

Importar funciones específicas.

También puedes importar una función concreta desde un módulo. Aquí está la sintaxis general para este enfoque:

```
from module_name import function_name
```

Puedes importar tantas funciones como quieras de un módulo separando el nombre de cada función con una coma:

```
from module_name import function_0, function_1, function_2
```

El ejemplo *making_pizzas.py* se podría parecer a esto si quisieramos importar sólo la función que vayamos a usar:

```
from pizza_2 import make_pizza  
  
make_pizza(16, 'pepperoni')  
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Con esta sintaxis no necesitas usar la notación de punto cuando llames a la función. Debido a que estamos importando explícitamente la `make_pizza()` en la sentencia `import`, podemos llamarla por el nombre cuando usemos la función.

Asignar un Alias a una función.

Si el nombre de la función que estás importando puede crear conflicto con un nombre existente en tu programa o si el nombre de la función es muy largo, puedes utilizar uno corto, un alias que es como un nombre parecido a un apodo para la función.

Aquí le damos a la función `make_pizza()` un alias `mp()`, para importar `make_pizza` como `mp`. La palabra clave `as` renombra la función utilizando el alias que tú le proporciones:

```
from pizza_2 import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

La instrucción `import` que se muestra aquí, cambia el nombre de la función `make_pizza()` a `mp()` en este programa. Cada vez que queramos llamar a `make_pizza()` podemos escribir simplemente `mp()` en su lugar y Python ejecutará el código de `make_pizza()` evitando cualquier confusión con otra función `make_pizza` que puedas haber escrito en este archivo de programa.

La sintaxis general para proporcionar un alias es:

```
from module_name import function_name as fn
```

Utilizar as para dar un alias a un módulo.

También puedes proporcionar un alias para el nombre de un módulo. Dando un alias corto a un módulo como `p` para `pizza`, te permite llamar a las funciones de los módulos más rápidamente. La llamada `p.make_pizza()` es más concisa que la llamada `pizza.make_pizza()`:

```
import pizza_2 as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Al módulo pizza se le ha dado el alias *p* en la instrucción **import**, pero todos las funciones del módulo conservan sus nombres originales.

Llamar al as funciones escribiendo *p.make_pizza()* no sólo es más conciso que escribir *pizza.make_pizza()* sino que también redirige tu atención desde el nombre del módulo y te permite centrarte en los nombres descriptivos de las funciones. Estos nombres de función, que claramente te dicen lo que hace cada una, son más importantes para la legibilidad de tu código que utilizar el nombre del módulo completo.

La sintaxis general para este enfoque es:

```
import module_name as mn
```

Importar todas las funciones de un módulo.

Puedes decirle a Python que importe cada función de un módulo utilizando el operador asterisco:

```
from pizza_2 import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

El asterisco en la sentencia **import** le dice a Python todas las funciones del módulo pizza a este archivo de programa. Puesto que cada función es importada puedes llamar a cada función por su nombre sin usar la notación punto. Sin embargo, es mejor no usar este enfoque cuando trabajes con módulos más grandes que no hayas escrito tú: si el módulo tienen un nombre de función que coincide con un nombre existente de tu proyecto, puedes obtener resultados inesperados. Python puede ver varias funciones o variables con el mismo nombre y en lugar de importar todas las funciones por separado, las sobrescribirá.

El mejor enfoque es importar la función o funciones que quieras o importar todo el módulo y usar la notación punto. Esto conduce a un código fácil de leer y de entender. Se incluye esta sección para reconocer sentencias **import** como las siguientes cuando las veas en el código de otras personas:

```
from module_name import *
```

Diseñar funciones.

Debes tener en cuenta algunos detalles cuando diseñas funciones. Las funciones deben tener nombres descriptivos y esos nombres deben usar letras minúsculas y guiones bajos. Los nombres descriptivos te ayudan a ti y a otros a comprender que está intentando hacer tu código. Los nombres de módulos deben usar también estas convenciones.

Cada función debe tener un comentario que explique de forma concisa qué hace la función. Este comentario debe aparecer inmediatamente después de la definición de la función y usar el formato comillas. En una función bien documentada, otros programadores pueden usar la función leyendo sólo la descripción entre las comillas. Se debe poder confiar en que el código funciona como se describe en los comentarios y siempre que se conozcan el nombre de la función, los argumentos que necesita y el tipo de valor que devuelve, debes ser capaz de usarlo en tus programas.

Si especificas un valor predeterminado para un parámetro, no se debe utilizar ningún espacio a cada lado del signo igual.

```
def function_name(parameter_0, parameter_1='default value')
```

La misma convención se debe usar para los argumentos de palabras clave en las llamadas a la función:

```
function_name(value_0, parameter_1='value')
```

El [PEP 8](#) recomienda que limites las líneas de código a 79 caracteres para que cada línea sea visible en una ventana del editor de código a un tamaño razonable. Si un conjunto de

parámetros provoca que la definición de una función sea mayor de 79 caracteres, presiona **ENTER** después de abrir paréntesis en la línea de la definición. En la siguiente línea pulsa **TAB** dos veces para separar la lista de argumentos del cuerpo de la función que sólo tendrá un nivel de indentado.

La mayoría de los editores alinean automáticamente cualquier línea adicional de parámetros para que coincida con el indentado que se ha establecido en la primera línea:

```
def function_name(  
    parameter_0, parameter_1, parameter_2,  
    parameter_3, parameter_4, parameter_5):  
    function body...
```

Si tu programa o módulo tiene más de una función, puedes separar cada una con dos líneas en blanco para que sea más fácil ver dónde termina una función y comienza la siguiente.

Todas las instrucciones **import** deben ser escritas al principio del fichero. La única excepción es si usas comentarios al principio de tu fichero para describir el programa en general.

EJERCICIOS – TRY IT YOURSELF.

8-15. Printing Models: Pon las funciones del ejemplo *printing_models.py* en un archivo separado llamado *printing_functions.py*. Escribe una sentencia **import** en la parte superior de *printing_models.py*, y modifica el archivo para utilizar las funciones importadas.

8-16. Imports: Usa un programa que hayas escrito que contenga una función, almacena la función en un archivo separado. Importa la función al archivo de tu programa principal y llama a la función utilizando cada uno de estos enfoques:

```
import module_name  
from module_name import function_name  
from module_name import function_name as fn  
import module_name as mn  
from module_name import *
```

8-17. Styling Functions: Elige tres programas cualesquier escritos en este capítulo y asegúrate de que sigue las guías de estilo descritas en esta sección.

Resumen .

En este capítulo hemos aprendido cómo escribir funciones y pasar argumentos para que tus funciones tengan acceso a la información que necesitan para hacer su trabajo. Hemos aprendido cómo usar argumentos posicionales y de palabras clave y cómo aceptar un número arbitrario de argumentos. Hemos visto funciones que muestran salidas por pantalla y funciones que devuelven valores. Hemos aprendido cómo usar funciones con listas, diccionarios, sentencias *if* y bucles *while*. También hemos visto cómo almacenar funciones en archivos separados llamados módulos para que los archivos de tus programas sean más simples y fáciles de entender. Por último hemos aprendido a diseñar funciones para que nuestros programas continúen estando bien estructurados y sean tan sencillos como sea posible tanto para nosotros como para otras personas que los lean.

Uno de nuestros objetivos como programadoras y programadores debe ser escribir código simple que haga lo que queremos que haga y funciones que ayuden a hacerlo. Te permitirán escribir bloques de código y olvidarte de ellos una vez que sepas que funcionan. Cuando sabes que una función hace su trabajo correctamente, puedes confiar en que continuará funcionando y que pasará a tu próxima tarea de codificación.

Las funciones te permiten escribir código una vez y reutilizar ese código tantas veces como quieras. Cuando necesites ejecutar el código de una función tendrás que escribir una llamada de una línea y la función hará su trabajo. Cuando necesites modificar el comportamiento de una función, sólo tendrás que modificar un bloque de código y tus cambios tendrán efecto en todos los lugares en los que hayas llamado a esa función.

Utilizar funciones hace que tus programas sean más fáciles de leer y un buen nombre de función resume lo que hace cada parte de un programa. Leer una serie de llamadas a funciones te da una idea mucho más rápida de lo que hace un programa que leer una serie larga de bloques de código.

Las funciones también hacen que tu código sea más fácil de probar y depurar. Cuando la mayor parte del trabajo de tu programa se realiza mediante un conjunto de funciones, cada una de las cuales tiene un trabajo específico, es mucho más fácil probar y mantener el código que has escrito. Puedes escribir un programa por independiente que llame a cada función y probar si cada una de ellas funciona en todas las situaciones que te puedas encontrar. Al hacer esto, puedes estar seguro de que tus funciones trabajarán correctamente cada vez que las llames.

En el capítulo 9 aprenderás a escribir clases. Las clases combinan funciones y datos en un paquete ordenado que se puede utilizar de manera flexible y eficiente.

CAPÍTULO 9. CLASES.

La programación orientada a objetos es uno de los enfoques más efectivos para escribir software. En la programación orientada a objetos escribes clases que representan situaciones y cosas del mundo real, y creas objetos basados en esas clases. Cuando escribes una clase, defines el comportamiento general que puede tener toda una categoría de objetos. Cuando creas objetos individuales desde la clase, cada objeto es dotado automáticamente con el comportamiento general y a continuación se le da cada uno el rasgo único que quieras. Te sorprenderás lo bien que se pueden modelar las situaciones del mundo real con la programación orientada a objetos.

Hacer un objeto desde una clase se llama instanciar y tu trabajas con instancias de una clase. En este capítulo escribirás clases y crearás instancias de esas clases. Especificarás el tipo de información que será almacenada en instancias y definirás las acciones que pueden tomar con esas instancias. También escribirás clases que extiendan la funcionalidad de clases ya existentes, por lo que clases similares pueden compartir código de manera eficiente. Almacenarás clases en módulos e importarás clases escritas por otros programadores en tus propios archivos de programa.

Entender la programación orientada a objetos te ayudará a ver el mundo como lo hace un programador. Te ayudará a conocer realmente tu código, no sólo lo que esté sucediendo línea a línea, sino también los grandes conceptos que hay tras ello. Conocer la lógica detrás de las clases te entrenará a pensar con lógica para que puedas escribir programas que aborden de manera efectiva casi cualquier problema que encuentres.

Las clases también hacen la vida más fácil a ti y a otros programadores con los que trabajes a medida que asumes desafíos cada vez más complejos. Cuando tú y otros programadores escribáis código basado en el mismo tipo de lógica, podrás comprender el trabajo de cada uno. Tus programas tendrán sentido para muchos colaboradores, permitiendo que todos logren más.

Crear y usar una clase.

Puedes modelar casi cualquier cosa usando clases. Empecemos escribiendo una simple clase, Dog, que represente un perro, no uno en particular, sino cualquier perro. ¿Qué sabemos acerca de los perros como mascotas? Bien, todos ellos tienen un nombre y una edad. También sabemos que muchos perros se sientan y se dan la vuelta. Estas dos partes de información (nombre y edad) y estos dos comportamientos (sentarse y darse la vuelta) irán en nuestra clase Dog puesto que son muy comunes en la mayoría de los perros. Después de escribir nuestra clase, la usaremos para hacer instancias individuales, cada una de las cuales representa a un perro específico.

Creando la clase perro.

Cada instancia creada desde la clase Dog almacenará un nombre y una edad y daremos a cada perro la habilidad de sit() (sentarse) y roll_over() (darse la vuelta):

```
new*
class Dog:
    """Un intento sencillo de modelar un perro."""

    new*
    def __init__(self, name, age):
        """Inicializa los atributos nombre y edad."""
        self.name = name
        self.age = age

    new*
    def sit(self):
        """Simula a un perro sentándose respondiendo a un comando."""
        print(f"{self.name} is now sitting.")

    new*
    def roll_over(self):
        """Simula dar la vuelta respondiendo a un comando."""
        print(f"{self.name} rolled -over.")
```

Hay mucho que observar aquí, pero no te preocupes. Verás esta estructura a través de este capítulo y tendrás mucho tiempo para acostumbrarte a ella. Definimos la clase llamada *Dog*. Por convención, los nombres de las clases se escriben con mayúscula en Python. No hay paréntesis en la definición de la clase porque estamos creando la clase desde cero. A continuación escribimos un *docstring* describiendo lo que hace la clase.

El método __init__().

Una función que parte de una clase, es un método. Todo lo que has aprendido acerca de las funciones también se aplica a los métodos; prácticamente, la única diferencia que hay por ahora es la forma en que llamaremos a los métodos. El método __init__() es un método especial que Python ejecuta automáticamente cada vez que creamos una instancia de la clase perro. Este método tiene dos guiones bajos iniciales y dos guiones bajos finales, una convención que evita que los nombres de los métodos por defecto de Python entren en conflicto con los nombres de tus métodos. Asegúrate de usar dos guiones bajos a cada lado de __init__(). Si sólo usas uno a cada lado, el método no será llamado automáticamente cuando uses tu

clase, lo que puede dar lugar a errores que serán difíciles de identificar.

Definimos el método `__init__()` para tener tres parámetros: `self`, `name` y `age`. El parámetro `self` es necesario en la definición del método y debe aparecer el primero antes que los demás parámetros. Debe ser incluido en la definición porque cuando Python llame a este método más tarde (para crear una instancia de `Dog`), la llamada al método pasará automáticamente el argumento `self`. Cada llamada al método asociada con una instancia pasa automáticamente `self`, que es una referencia a la propia instancia. Proporciona a la instancia individual acceso a los atributos y métodos de la clase. Cuando creamos una instancia de `Dog`, Python llama al método `__init__()` desde la clase `Dog`. Pasaremos un nombre y una edad a `Dog` como argumentos; `self` se pasa automáticamente, por lo que no necesitamos pasarlo. Siempre que queramos crear una instancia de la clase `Dog`, proporcionaremos valores sólo para los dos últimos parámetros, `name` y `age`.

Las dos variables definidas dentro del método `__init__()` tienen cada una el prefijo `self`. Cualquier variable con el prefijo `self` estará disponible para todos los métodos de la clase y también podremos acceder a esas variables a través de cualquier instancia creada a partir de la clase. La línea `self.name = name` toma el valor asociado con el parámetro `name` y lo asigna a la variable `name` que luego se adjunta a la instancia que se está creando. El mismo proceso ocurre con `self.age = age`. Las variables que son accesibles desde instancias como estas son llamadas atributos.

La clase `Dog` tiene otros dos métodos definidos: `sit()` y `roll_over()`. Puesto que estos dos métodos no necesitan información adicional para ejecutarse, sólo los definiremos para que tengan un parámetro, `self`. Las instancias que creamos más tarde tendrán acceso a estos métodos. En otras palabras, podrán sentarse y darse la vuelta. Por ahora `sit()` y `roll_over()` no hacen mucho más. Simplemente imprimen un mensaje diciendo que el perro se ha sentado o que el perro se ha dado la vuelta. Pero el concepto se puede extender a situaciones realistas: si esta clase fuera parte de un juego real, estos métodos contendrían código para hacer que un perro animado se sentase o se diese la vuelta.

Creación de una instancia a partir de una clase.

Piensa en una clase como en un conjunto de instrucciones sobre cómo crear una instancia. La clase *Dog* es un conjunto de instrucciones que le dicen a Python cómo crear instancias individuales que representen perros específicos.

Creemos una instancia representando a un perro en concreto:

```
my_dog = Dog('Nuble', 6)

print(f"Mi perro se llama {my_dog.name}.")
print(f"Y tiene {my_dog.age} años.")
```

Primero le decimos a Python que cree un perro que se llame '*Nuble*' y que tenga 6 años. Cuando Python lee esta línea, llama al método `init_()` de la clase *Dog* con los argumentos '*Nuble*' y 6. El método `init_()` crea una instancia representando este perro en particular y configura los atributos *name* y *age* utilizando los valores proporcionados. A continuación Python devuelve una instancia representando este perro. Asignamos esa instancia a la variable *my_dog*. La convención de nombres resulta de gran ayuda aquí: generalmente podemos asumir que un nombre en mayúscula como *Dog*, hace referencia a una clase y un nombre en minúscula como *my_dog* hace referencia a una sola instancia creada a partir de esa clase.

Acceso a atributos.

Para acceder a los atributos de una instancia usaremos la notación punto. En el anterior ejemplo accedemos al valor del atributo *name* de *my_dog* escribiendo: `my_dog.name`

La notación punto se utiliza con frecuencia en Python. Esta sintaxis muestra cómo Python encuentra el valor de un atributo. Aquí Python mira la instancia *my_dog* y luego encuentra el nombre del atributo asociado con *my_dog*. Este es el mismo atributo al que se hace referencia como *self.name* en la clase *Dog*. A continuación usamos el mismo enfoque para trabajar con el atributo *age*.

La salida es un resumen de lo que sabemos acerca de *my_dog*:

```
Mi perro se llama Nuble.  
Y tiene 6 años.  
  
Process finished with exit code 0
```

Llamadas a métodos.

Después de crear una instancia de la clase *Dog*, podemos usar la notación punto para llamar a cualquier método de la clase *Dog*. Hagamos que nuestro perro se siente y se de la vuelta:

```
my_dog.sit()  
my_dog.roll_over()
```

Para llamar al método damos el nombre de la instancia (en este caso *my_dog*) y el método al que quieras llamar separado por un punto. Cuando Python lee *my_dog.sit()*, busca el método *sit()* en la clase *Dog* y ejecuta el código. Python interpreta la línea *my_dog.roll_over()* de la misma manera.

Ahora Nuble hace lo que le hemos dicho que haga:

```
Nuble is now sitting.  
Nuble rolled -over.
```

Esta sintaxis es bastante útil. Cuando los atributos y métodos han recibido nombres descriptivos como *name*, *age*, *sit()* y *roll_over()*, podemos inferir fácilmente qué se supone que debe hacer un bloque de código, incluso uno que nunca hemos visto antes.

Crear múltiples instancias.

Puedes crear tantas instancias de una clase como necesites. Creemos una segunda clase llamada *your_dog*:

```
my_dog = Dog('Nuble', 6)
your_dog = Dog('Willie', 7)

print(f"Mi perro se llama {my_dog.name}.")
print(f"Mi perro tiene {my_dog.age} años.")
my_dog.sit()

print(f"Tu perro se llama {your_dog.name}.")
print(f"Tu perro tiene {your_dog.age} años.")
your_dog.sit()
```

En este ejemplo creamos un perro llamado Nuble y un perro llamado Willie. Cada perro es una instancia separada con sus propios atributos, capaz del mismo conjunto de acciones:

```
Mi perro se llama Nuble.
Mi perro tiene 6 años.
Nuble is now sitting.

Tu perro se llama Willie.
Tu perro tiene 7 años.
Willie is now sitting.

Process finished with exit code 0
```

Incluso si usamos el mismo nombre y la misma edad para el segundo perro, Python creará una instancia separada de la clase *Dog*. Puedes crear tantas instancias de una clase como necesites, siempre y cuando des a cada instancia un nombre de variable único u ocupen un único lugar en una lista o diccionario.

EJERCICIOS - TRY IT YOURSELF.

9-1. *Restaurant*: Crea una clase llamada *Restaurant*. El método `__init__()` para *Restaurant* debe almacenar dos atributos: `restaurant_name` y `cuisine_type`. Crea un método llamado `describe_restaurant()` que imprima esas dos partes de información y un método llamado `open_restaurant()` que imprima un mensaje indicando que el restaurante está abierto.

Crea una instancia llamada `restaurant` de tu clase. Imprime los dos atributos individualmente y luego llama a ambos métodos.

9-2. *Three restaurants*: Comienza tu clase desde el ejercicio 9-1. Crea tres instancias diferentes de la clase y llama a `describe_restaurant()` para cada instancia.

9-3. *Users*: Crea una clase llamada *User*. Crea dos atributos llamados `first_name` y `last_name` y a continuación, crea otros atributos que normalmente se almacenen en un perfil de usuario. Crea un método llamado `describe_user()` que imprima un resumen de la información del usuario. Crea otro método llamado `greet_user()` que imprima un saludo personalizado para el usuario.

Crea varias instancias representando diferentes usuarios y llama a ambos métodos para cada usuario.

Trabajar con clases e instancias.

Puedes usar clases para representar muchas situaciones del mundo real. Una vez que escribes una clase, dedicarás mucho tiempo trabajando con instancias creadas de esa clase. Una de las primeras tareas que querrás hacer es modificar los atributos asociados con una instancia en particular. Puedes modificar los atributos de una instancia directamente o escribir métodos que actualicen los atributos de formas específicas.

La clase Car.

Escribamos una nueva clase representando a un coche. Nuestra clase almacenará información acerca del tipo de coche con el que estamos trabajando y tendrá un método que resuma esa información.

```

new*
class Car:
    """un simple intento de representar un coche."""
    new*
    def __init__(self, make, model, year):
        """Inicializa los atributos para describir un coche."""
        self.make = make
        self.model = model
        self.year = year

    new*
    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo con formato ordenado."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

```

Primero definimos el método `__init__()` con el parámetro `self` en primer lugar, tal y como hicimos antes con nuestra clase `Dog`. También proporcionamos otros tres parámetros: `make`, `model` y `year`. El método `__init__()` toma estos parámetros y los asigna a los atributos que serán asociados a las instancias creadas a partir de esta clase. Cuando creamos una nueva instancia de `Car`, necesitaremos especificar un fabricante (`make`), un modelo (`model`) y un año (`year`) para nuestra instancia.

A continuación definimos un método llamado `get_descriptive_name()` que pondrá el fabricante, año y modelo en una sola cadena que describirá perfectamente al automóvil. Esto nos ahorrará tener que imprimir el valor de cada atributo individualmente. Para trabajar con los valores de los atributos en este método usamos `self.make`, `self.model` y `self.year`. Por último creamos una instancia de la clase `Car` y la asignamos a la variable `my_new_car`. Luego llamamos a `get_descriptive_name()` para mostrar que tipo de coche tenemos:

```

2019 Audi A4
Process finished with exit code 0

```

Para hacer la clase más interesante, agreguemos un atributo que cambie con el tiempo. Añadiremos un atributo que almacene el kilometraje total del automóvil.

Establecer un valor por defecto para un atributo.

Cuando se crea una instancia, los atributos se pueden definir sin pasarlos como argumentos. Estos atributos se pueden definir en el método `__init__()` donde se les asigna un valor por defecto.

Añadamos un atributo llamado `odometer_reading` que inicie siempre con el valor de 0. También añadimos un método `read_odometer()` que nos ayude a leer cada cuentakilómetros del coche.

```
▲ JuanJesus*
class Car:
    """un simple intento de representar un coche."""
    ▲ JuanJesus*
    def __init__(self, make, model, year):
        """Inicializa los atributos para describir un coche."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    ▲ JuanJesus*
    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo con formato ordenado."""
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()

    new*
    def read_odometer(self):
        """Imprime una frase que muestra el kilometraje del coche."""
        print(f'Este coche tiene {self.odometer_reading} kilómetros.')

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

En esta ocasión, cuando Python llama al método `__init__()` para crear una nueva instancia, almacena los valores `make`, `model` y `year`

como atributos, igual que hizo en el ejemplo anterior. A continuación, Python crea un nuevo atributo llamado *odometer_reading* y establece su valor inicial a 0. También tenemos un método llamado *read_odometer()* que hace más fácil leer el kilometraje de un coche.

Nuestro coche comienza con un kilometraje de 0:

```
2019 Audi A4
Este coche tiene 0 kilómetros.

Process finished with exit code 0
```

No se venden muchos coches con exactamente 0 kilómetros en el cuenta kilómetros, por lo que se necesita una forma de cambiar el valor de este atributo.

Modificar valores de atributos.

Puedes cambiar los valores de los atributos de tres maneras : puedes cambiar el valor directamente con una instancia, establecer el valor a través de un método o incrementar el valor (agregar una cantidad determinada) a través de un método. Echemos un vistazo a cada uno de estos enfoques.

Modificar el valor de un atributo directamente.

La manera más sencilla de modificar el valor de un atributo es acceder al atributo a través de una instancia. Aquí establecemos la lectura del kilometraje directamente a 23:

```
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

Utilizamos la notación punto para acceder al atributo *odometer_reading* del coche y establecer su valor directamente. Esta sentencia le dice a Python que tome la instancia *my_new_car*,

encuentre el atributo `odometer_reading` asociado a él y establezca el valor del atributo a 23:

```
2019 Audi A4
Este coche tiene 23 kilómetros.

Process finished with exit code 0
```

A veces querrás acceder directamente a los atributos de esta manera, pero otras veces querrás escribir un método que actualice el valor por ti.

Modificar el valor de un atributo a través de un método.

Puede ser de gran ayuda tener métodos que actualicen ciertos atributos por ti. En lugar de acceder al atributo directamente, se pasa el nuevo valor a un método que controla la actualización internamente.

Este ejemplo muestra un método llamado `update_odometer()`:

```
new*
def update_odometer(self, mileage):
    """Establece la lectura del cuentakilómetros al valor dado."""
    self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)
my_new_car.read_odometer()
```

La única modificación para coche es el añadido del método `update_odometer()`. Este método toma un valor de kilometraje y lo asigna a `self.odometer_reading`. Llamamos a `update_odometer()` y le damos 23 como argumento (correspondiente al parámetro kilometraje en la definición del método). Esto establece la lectura del cuentakilómetros a 23 y `read_odometer()` imprime esta lectura.

```
2019 Audi A4
Este coche tiene 23 kilómetros.

Process finished with exit code 0
```

Podemos ampliar el método `update_odometer()` para que haga tareas adicionales cada vez que la lectura del cuentakilómetros sea modificada. Añadamos un poco de lógica para asegurarnos de que nadie invierte la lectura del cuentakilómetros.

```
new*
def update_odometer(self, mileage):
    """Establece la lectura del cuentakilómetros al valor dado.
       Rechaza el cambio si se intenta hacer retroceder el cuentakilómetros."""
    if mileage <= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("no puedes retroceder el cuentakilómetros, hijueputa!")
```

Ahora, `update_odometer()` comprueba que la nueva lectura tiene sentido antes de modificar el atributo. Si el nuevo kilometraje, `mileage`, es mayor que o igual al ya existente, `self.odometer_reading`, puedes actualizar el cuentakilómetros a la nueva lectura. Si el nuevo kilometraje es menor al kilometraje existente, obtendrás un aviso de que no puedes hacer retroceder el cuentakilómetros.

Incrementar el valor de un atributo a través de un método.

A veces querrás incrementar el valor de un atributo en una determinada cantidad en lugar de establecer un valor completamente nuevo. Digamos que compramos un coche usado y le ponemos 160,93 kilómetros entre el momento en que lo compramos y el momento en que lo registramos. Aquí tenemos un método que nos permite pasar esta cantidad de incremento y añadir ese valor a la lectura del cuentakilómetros:

```
9     def increment_odometer(self, miles):
10         """Añade una cantidad dada a la lectura del cuentakilómetros."""
11         self.odometer_reading += miles
12
13
14 my_used_car = Car('Subaru', 'outback', 2015)
15 print(my_used_car.get_descriptive_name())
16
17 my_used_car.update_odometer(23_500)
18 my_used_car.read_odometer()
19
20 my_used_car.increment_odometer(160.93)
21 my_used_car.read_odometer()
```

El nuevo método `increment_odometer()` toma un número de millas y añade ese valor a `self.odometer_reading`. Creamos un coche usado `my_used_car`. Establecemos el cuentakilómetros a 23,500 llamando a `update_odometer()` y pasamos esos 23,500. Llamamos a `increment_odometer()` y pasamos 160,93 para añadir los 160,93 kilómetros que hemos conducido entre la compra del coche y su registro:

```
2015 Subaru Outback
Este coche tiene 23500 kilómetros.
Este coche tiene 23660.93 kilómetros.

Process finished with exit code 0
```

Puedes modificar fácilmente este método para rechazar incrementos negativos para que nadie use esta función para revertir un cuentakilómetros.

NOTA: Puedes utilizar métodos como estos para controlar cómo los usuarios de tu programa actualizan valores como la lectura de un cuentakilómetros, pero alguien con acceso al programa puede establecer la lectura del cuentakilómetros a cualquier valor accediendo directamente al atributo. Un sistema de seguridad presta especial atención a los detalles, además de a tareas de revisión básicas como las mostradas anteriormente.

EJERCICIOS – TRY IT YOURSELF.

9-4. Number Served: Comienza con tu programa del ejercicio 9-1. Añade un atributo llamado `number_served` con un valor predeterminado de 0. Crea una instancia de esta clase llamada `restaurant`. Imprime el número de clientes a los que ha servido el restaurante y luego cambia este valor e imprímelo de nuevo.

Añade un método llamado `set_number_served()` que te permita establecer el número de clientes que han sido servidos. Llama a este método con un nuevo número e imprime el valor otra vez.

Añade un método llamado `increment_number_served()` que te permita incrementar el número de clientes que han sido atendidos. Llama a este método con el número que quieras que pueda representar cuántos clientes fueron atendidos en, digamos, un día de trabajo.

9-5. Login Attempts: Añade un atributo llamado `login_attempts` a tu clase `User` del ejercicio 9-3. Escribe un método llamado `increment_login_attempts()` que incremente el valor de `login_attempts` en 1. Escribe otro método llamado `reset_login_attempts()` que resetee el valor de `login_attempts` a 0. Crea una instancia de la clase `User` y llama a `increment_login_attempts()` varias veces. Imprime el valor de `login_attempts` para asegurarte de que se incrementó adecuadamente, y luego llama a `reset_login_attempts()`. Imprime `login_attempts` de nuevo para asegurarte de que se reseteó a 0.

Herencia.

No siempre tienes que empezar de cero al escribir una clase. Si la clase que estás escribiendo es una versión especializada de otra clase que hayas escrito, puedes usar herencia. Cuando una clase hereda de otra, toma los atributos y métodos de la primera clase. La clase original se llama clase padre, y la nueva clase es la clase hija. La clase hija puede heredar cualquiera o todos los métodos de la clase padre, pero también es libre de definir nuevos atributos y métodos propios.

El método `__init__()` para una clase hija.

Cuando estás escribiendo una nueva clase basada en una clase existente, con frecuencia querrás llamar al método `__init__()` desde la clase padre. Esto inicializará cualquier atributo que se definió en el método `__init__()` de la clase padre y hará que estén disponibles en la clase hija.

Como ejemplo, modelemos un coche eléctrico. Un coche eléctrico es sólo un tipo específico de coche, por lo que podemos basar nuestra nueva clase `ElectricCar` en la clase `Car` que escribimos anteriormente. Entonces sólo tendremos que escribir código para los atributos y el comportamiento específico de los coches eléctricos.

Comencemos haciendo una versión simple de la clase `ElectricCar` que hace todo lo que hace la clase `Car`:

```
new*
class ElectricCar(Car):
    """Representa aspectos de un coche específico para vehículos eléctricos."""

    new*
    def __init__(self, make, model, year):
        """Inicializa los atributos de la clase padre."""
        super().__init__(make, model, year)

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
```

Comenzamos con la clase `Car`. Cuando creas una clase hija, la clase padre debe formar parte del fichero actual y debe aparecer antes de la clase hija en el fichero. A continuación definimos la clase hija, `ElectricCar`. Se debe incluir el nombre de la clase padre en los paréntesis en la definición de la clase hija. El método `__init__()` toma la información requerida para crear una instancia de `Car`.

La función `super()` es una función especial que te permite llamar un método de la clase padre. Esta línea le dice a Python que llame al método `__init__()` de la clase padre que da a una instancia de `ElectricCar` todos los atributos definidos en ese método. El nombre

proviene de una convención de llamar a la clase padre superclase y a la clase hija, subclase.

Comprobamos si la herencia está funcionando adecuadamente intentando crear un coche eléctrico con el mismo tipo de información que proporcionamos cuando hicimos un coche normal. A continuación creamos una instancia de *ElectricCar* y la asignamos a *my_tesla*. Esta línea llama al método `__init__()` definido en *ElectricCar*, que a su vez le dice a Python que llame al método `__init__()` definido en la clase padre, *Car*. Proporcionamos los argumentos 'tesla', 'model s' y 2019.

A parte de `__init__()` no hay atributos o métodos todavía que sean específicos de un coche eléctrico. En este momento, nos estamos asegurando de que el coche eléctrico tenga los comportamientos de *Car* adecuados:

```
2019 Tesla Model S

Process finished with exit code 0
```

La instancia de *ElectricCar* funciona justo como una instancia de *Car*, por lo que ahora podemos empezar a definir atributos y métodos específicos para coches eléctricos.

Definir atributos y métodos para la clase hija.

Una vez que tienes una clase hija que hereda de una clase padre, puedes añadir nuevos atributos y métodos necesarios para diferenciar la clase hija de la clase padre.

Añadamos un atributo específico para los coches eléctricos (una batería, por ejemplo) y un método para informar de este atributo. Almacenaremos el tamaño de la batería y escribiremos un método que imprima la descripción de la batería:

```
new*
class ElectricCar(Car):
    """Representa aspectos de un coche específico para vehículos eléctricos."""

    new*
    def __init__(self, make, model, year):
        """Inicializa los atributos de la clase padre.
           A continuación inicializa los atributos específicos de un coche eléctrico."""
        super().__init__(make, model, year)
        self.battery_size = 75

    new*
    def describe_battery(self):
        """Imprime una frase describiendo la batería."""
        print(f"Este coche tienen una batería de {self.battery_size}-KWh.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

Añadimos un nuevo atributo `self.battery_size` y establecemos el valor inicial a, digamos, 75. Este atributo se asociará con todas las instancias creadas a partir de la clase `ElectricCar`, pero no se asociará con ninguna instancia de `Car`. También añadimos un método llamado `describe_battery()` que imprime la información acerca de la batería. Cuando llamamos a este método, obtenemos una descripción que es claramente específica de un coche eléctrico:

```
2019 Tesla Model S
Este coche tienen una bateria de 75-KWh.

Process finished with exit code 0
```

No hay límites para todo cuánto se puede especializar la clase `ElectricCar`. Puedes añadir tantos atributos y métodos como necesites para modelar un coche eléctrico con el grado de precisión que necesites. Un atributo o método que podría pertenecer a cualquier coche en lugar de a uno que sea específico de un coche eléctrico, debe añadirse a la clase `Car` en lugar de a la clase `ElectricCar`. Entonces, cualquiera que utilice la clase `Car`, tendrá también disponibles esas funcionalidades y la clase `ElectricCar` sólo contendrá el código para la información y el comportamiento específico de los coches eléctricos.

Reemplazar métodos de la clase padre.

Puedes reemplazar cualquier método de la clase padre que no se ajuste a lo que estás intentando modelar con la clase hija. Para hacerlo, define un método en la clase hija con el mismo nombre que el método que quieras reemplazar en la clase padre. Python ignorará el método de la clase padre y sólo prestará atención al método de la clase hija.

Digamos que la clase `Car` tiene un método llamado `fill_gas_tank()`. Este método no tiene sentido para un vehículo eléctrico, por lo que puedes querer reemplazar este método. Aquí tienes la forma de hacerlo:

```
new*
def fill_gas_tank(self):
    """Los coches eléctricos no necesitan un tanque de gasolina."""
    print("Este coche no necesita un tanque de gasolina.")
```

Ahora, si alguien intenta llamar a `fill_gas_tank()` con un coche eléctrico, Python ignorará el método `fill_gas_tank()` de `Car` y ejecutará este código en su lugar. Cuando usas herencia, puedes hacer que las clases hijas conserven lo que necesiten y reemplacen cualquier cosa que no necesite de la clase padre.

Instancias como atributos.

Cuando modelamos algo del mundo real en código, puedes encontrar que estás añadiendo más y más detalles a la clase. Encontrarás que tienes una creciente lista de métodos y atributos y que tus archivos se están haciendo muy grandes. En estas situaciones puedes reconocer que parte de una clase se puede escribir como una clase separada. Puedes dividir tu clase grande en clases más pequeñas que funcionen juntas.

Por ejemplo, si continuamos añadiendo detalles a la clase `ElectricCar`, podemos advertir que estamos añadiendo demasiados atributos y métodos específicos para la batería del coche. Cuando veamos que esto está sucediendo, podemos parar y mover esos atributos y métodos a una clase separada llamada `Battery`. Entonces podemos usar una instancia de `Battery` como si fuera un atributo de la clase `ElectricCar`:

```
class Battery:
    """A simple attempt to model a battery for an electric car."""
    new*
    def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size
    new*
    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

new*
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    new*
    def __init__(self, make, model, year):
        """Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car."""
        super().__init__(make, model, year)
        self.battery = Battery()
```

Primero definimos una nueva clase llamada *Battery* que no hereda de ninguna otra clase. El método `__init__()` tiene un parámetro, *battery_size*, además de *self*. Este es un parámetro opcional que establece el tamaño de la batería a 75 si no se proporciona ningún valor. El método *describe_battery()* se ha movido a esta clase también.

En la clase *ElectricCar*, añadimos un atributo llamado *self.battery*. Esta línea le dice a Python que cree una nueva instancia de *Battery* (con un tamaño de 75 por defecto, puesto que especificamos ningún valor) y asigna esa instancia al atributo *self.battery*. Esto ocurre cada vez que llamamos al método `__init__()`. Cualquier instancia de *ElectricCar* tendrá ahora una instancia de *Battery* creada automáticamente.

Creamos un coche eléctrico y lo asignamos a la variable *my_tesla*. Cuando queremos describir la batería, necesitamos trabajarla a través del atributo *battery* del coche:

```
my_tesla.battery.describe_battery()
```

Esta línea le dice a Python que mire la instancia *my_tesla*, encuentre su atributo *battery* y llame al método *describe_battery()* que está asociado con la instancia *Battery* almacenada en el atributo.

La salida es idéntica a lo que vimos previamente:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Esto parece mucho trabajo extra, pero ahora podemos describir la batería con tanto detalle como queramos sin saturar la clase *ElectricCar*. Vamos a añadir otro método a *Battery* que informe del alcance del coche basado en el tamaño de la batería:

```

class Battery:
    """A simple attempt to model a battery for an electric car."""
    new *
    def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size
    new *
    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    new *
    def get_range(self):
        """Imprime una frase acerca del alcance que proporciona esta batería."""
        if self.battery_size == 75:
            range = 418.43
        elif self.battery_size == 100:
            range = 506.94

        print(f"Este coche puede circular unos {range} kilómetros con una carga completa.")

```

```

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()

```

El nuevo método `get_range()` realiza un simple análisis. Si la capacidad de la batería es 75 kWh, `get_range()` establece el alcance es 418,43 kilómetros, y si la capacidad es de 100 kWh, lo establece a 506,94 kilómetros. A continuación informa del valor. Cuando queramos usar este método tenemos que volver a llamarlo a través del atributo `battery` del coche.

La salida nos dice el alcance del coche en función del tamaño de su batería:

```

2019 Tesla Model S
This car has a 75-kWh battery.
Este coche puede circular unos 418.43 kilómetros con una carga completa.

Process finished with exit code 0

```

Modelando objetos del mundo real.

A medida que comiences a modelar cosas más complicadas como los coches eléctricos, lucharás con preguntas interesantes. ¿Es el alcance de un coche eléctrico una propiedad de la batería o del coche? Si sólo describimos un coche probablemente esté bien mantener la asociación del método `get_range()` con la clase `Battery`. Pero si estamos describiendo toda una línea de fabricantes de coches, probablemente queramos mover `get_range()` a la clase `ElectricCar`.

El método `get_range()` aún verificaría el tamaño de la batería antes de determinar el alcance, pero informaría un alcance específico para el tipo de coche con el que está asociado. Alternativamente, podemos mantener la asociación del método `get_range()` con la batería, pero pasando un parámetro como `car_model`. El método `get_range()` informaría de un alcance basado en el tamaño de la batería y el modelo del coche.

Esto te lleva a un punto interesante en tu crecimiento como programador. Cuando luches con preguntas como esta, estarás pensando a un alto nivel de lógica en lugar de en un nivel centrado en la sintaxis. Estarás pensando, no sobre Python sino sobre cómo representar en código el mundo real. Cuando llegues a este punto te darás cuenta de que a menudo no hay enfoques correctos o incorrectos para modelar situaciones del mundo real. Algunos enfoques son más eficientes que otros, pero se necesita práctica para encontrar las representaciones más eficientes. Si tu código funciona como quieras, ¡lo estarás haciendo bien! No te desanimes si descubres que estás destrozando tus clases y reescribiéndolas varias veces usando diferentes enfoques. En la búsqueda de escribir código preciso y eficiente, todos pasan por este proceso.

EJERCICIOS - TRY IT YOURSELF.

9-6. Ice Cream Stand: Un puesto de helados es un tipo específico de restaurante. Escribe una clase llamada `IceCreamStand` que herede de la clase `Restaurant` que escribiste en el ejercicio 9-1 o en el 9-4. Cualquiera de las versiones de la clase funcionará, sólo elige la que más te guste. Añade un atributo llamado `flavours` que almacene una lista de sabores de helados. Escribe un método que

muestre esos sabores. Crea una instancia de *IceCreamStand* y llama a este método.

9-7. Admin: Un administrador es un tipo especial de usuario. Escribe una clase llamada *Admin* que herede de la clase *User* que escribiste en el ejercicio 9-3 o 9-5. Añade un atributo, *privileges*, que almacene una lista de cadenas como “puede añadir un post”, “puede eliminar un post”, “puede banear a un usuario”, etc. Escribe un método llamado *show_privileges()* que liste el conjunto de privilegios del administrador. Crea una instancia de *Admin* y llama a tu método.

9-8. Privileges: Escribe una clase separada *Privileges*. La clase debe tener un atributo, *privileges*, que almacene una lista de cadenas como la descrita en el ejercicio anterior. Mueve el método *show_privileges()* a este clase. Crea una instancia como un atributo en la clase *Admin*. Crea una nueva instancia de *Admin* y usa tu método para mostrar sus privilegios.

9-9. Battery Upgrade: Usa la versión final de *ElectricCar* de esta sección. Añade un método a la clase *Battery* llamado *upgrade_battery()*. Este método podrá comprobar el tamaño de la batería y establecer la capacidad a 100 si no lo está ya. Crea un coche eléctrico con el tamaño de batería por defecto, llama al método *get_range()* una vez luego llama al método *get_range()* una segunda vez después de actualizar la batería. Debes ver un incremento en el alcance del coche.

Importar clases.

A medida que añades más funcionalidad a tus clases, tus ficheros pueden volverse más largos, incluso cuando uses la propiedad de la herencia. De acuerdo con la filosofía de Python, querrás mantener tus archivos lo más ordenados posible. Para ayudarte, Python te permite almacenar tus clases en módulos y luego importar las clases que necesites a tu programa principal.

Importación de una sola clase.

Creemos un módulo que contenga sólo la clase *Car*. Esto plantea un sutil problema de nomenclatura: ya tenemos un archivo llamado *car.py* en este capítulo, pero este módulo debe llamarse *car.py* porque contiene el código que representa a un coche. Resolveremos este problema de nomenclatura almacenando la clase *Car* en un módulo llamado *car.py*, reemplazando el archivo *car.py* que estábamos usando anteriormente. A partir de ahora, cualquier programa que utilice este módulo necesitará un nombre de archivo más específico, como *my_car.py*. Aquí se muestra *car.py* solamente con el código de la clase *Car*.

```
class Car:
    """un simple intento de representar un coche."""
    __author__ = "JuanJesuses"

    def __init__(self, make, model, year):
        """Inicializa los atributos para describir un coche."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    __author__ = "JuanJesuses"
    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo con formato ordenado."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    __author__ = "JuanJesuses"
    def read_odometer(self):
        """Imprime una frase que muestra el kilometraje del coche."""
        print(f"Este coche tiene {self.odometer_reading} kilómetros.")

    new*
    def update_odometer(self, mileage):
        """Establece la lectura del cuentakilómetros al valor dado.
        Rechaza el cambio si se intenta hacer retroceder el cuentakilómetros."""
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("no puedes retroceder el cuentakilómetros, hijueputa!")

    new*
    def increment_odometer(self, miles):
        """Añade una cantidad dada a la lectura del cuentakilómetros."""
        self.odometer_reading += miles
```

En primer lugar incluimos un docstring de nivel de módulo que describe brevemente el contenido de este módulo. Debes escribir un docstring para cada módulo que crees.

Ahora haremos un archivo separado llamado `my_car.py`. Este archivo importará la clase `Car` y luego creará una instancia a partir de esa clase:

```
from car import Car

my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

La sentencia `import` le dice a Python que abra el módulo `car` e importe la clase `Car`. Ahora podemos usar la clase `Car` como si estuviera definida en este archivo. La salida es la misma que vimos anteriormente:

```
2019 Audi A4
Este coche tiene 23 kilómetros.

Process finished with exit code 0
```

Importar clases es una forma efectiva de programar. Imagínate cuánto tiempo tardaría el archivo de programa si se incluyera toda la clase `Car`. En su lugar, mueves la clase a un módulo e importas el módulo que aún contiene toda la misma funcionalidad, pero mantiene su archivo de programa principal limpio y fácil de leer. También almacenarás la mayoría de la lógica en archivos separados; una vez que tus clases funcionan como quieras que lo hagan, puedes dejar solos esos archivos y centrarte en la lógica de alto nivel de tu programa principal.

Almacenamiento de varias clases en un módulo.

Puedes almacenar tantas clases como necesites en un sólo módulo, aunque cada clase de ese módulo debe estar relacionada de alguna

manera. Las clases *Battery* y *ElectricCar* ayudan a representar coches, así que añadámoslas al módulo *car.py*.

Ahora podemos hacer un nuevo archivo llamado *my_electric_car.py*, importar la clase *ElectricCar* y hacer un coche eléctrico:

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Esto tiene la misma salida que vimos anteriormente, aunque la mayor parte de la lógica está escondida en un módulo:

```
2019 Tesla Model S
This car has a 100-kWh battery.
Este coche puede circular unos 506.94 kilómetros con una carga completa.

Process finished with exit code 0
```

Importar múltiples clases desde un módulo.

Puedes importar tantas clases como necesites a tu archivo de programa. Si queremos hacer un coche tradicional y un coche eléctrico en el mismo archivo, necesitaremos importar ambas clases, *Car* y *ElectricCar*:

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

Puedes importar múltiple clases desde un módulo separando cada clase con una coma. Una vez que has importado las clases

necesarias, eres libre de hacer tantas instancias de cada clase como necesites.

En este ejemplo hacemos un Volkswagen Beetle tradicional y un coche eléctrico Tesla Roadster:

```
2019 Volkswagen Beetle
2019 Tesla Roadster

Process finished with exit code 0
```

Importación de un módulo completo.

También puedes importar un módulo completo y acceder a las clases usando la notación punto. Este enfoque es simple y resulta un código que es fácil de leer. Puesto que cada llamada que crea una instancia de clase incluye el nombre de un módulo, no tendrás conflictos de nomenclatura con los nombres utilizados en el archivo actual.

Aquí se muestra cómo importar la módulo car completo y luego crear un coche tradicional y un coche eléctrico:

```
import car

my_beetle = car.Car('volkswagen', 'beetle', 2019)
print(my_beetle.get_descriptive_name())

my_tesla = car.ElectricCar('tesla', 'roadster', 2019)
print(my_tesla.get_descriptive_name())
```

Importamos todo el módulo *car*. A continuación accedemos a las clases que necesitemos a través de la sintaxis *module_name.ClassName*. De nuevo creamos un Volkswagen Beetle y un Tesla Roadster.

Importar todas las clases de un módulo.

Puedes importar todas las clases de un módulo utilizando la siguiente sintaxis:

```
from module_name import *
```

Este método no se recomienda por dos razones. Primero, resulta útil poder leer las instrucciones import en la parte superior de una archivo y tener una idea clara de qué clases utiliza un programa. Con este enfoque no está muy claro qué clases del módulo estamos usando. También puede dar lugar a confusión con los nombres de archivo. Si accidentalmente importas una clase con el mismo nombre de cualquier otra clase o módulo de tu programa, puedes crear errores difíciles de diagnosticar. Esto se muestra porque, aunque no es un enfoque recomendado, es probable que lo veas en el código de otras personas en algún momento.

Si necesitas importar muchas clases desde un módulo, es mejor que importes todo el módulo y utilices la sintaxis `module_name.ClassName`.

No verás todas las clases en la parte superior del fichero, pero verás claramente dónde se ha utilizado el módulo en el programa. También podrás evitar los posibles conflictos de nomenclatura que puedan surgir al importar todas las clases de un módulo.

Importación de un módulo en un módulo.

A veces querrás distribuir tus clases en varios módulos para evitar que cualquier archivo crezca demasiado y evitar almacenar clases no relacionadas en el mismo módulo. Cuando almacenas tus clases en muchos módulos puedes encontrar que una clase en un módulo dependa de otra clase en otro módulo. Cuando esto ocurra, puedes importar la clase requerida en el primer módulo.

Por ejemplo, almacenemos la clase `Car` en un módulo y las clases `ElectricCar` y `Battery` en un módulo separado. Haremos un nuevo módulo llamado `electric_car.py` - sustituyendo al archivo `electric_car` creado anteriormente - y sólo copiaremos las clases `Battery` y `ElectricCar` en este archivo:

```
from car import Car

class Battery:
    --snip--

class ElectricCar(Car):
    --snip--
```

La clase *ElectricCar* necesita acceso a su clase padre *Car*, por lo que importamos *Car* directamente al módulo *electric_car.py*. Si olvidamos esta línea, Python generará un error cuando intentemos importar el módulo *electric_car*. También necesitamos actualizar el módulo *car* para que sólo contenga la clase *Car*:

```
"""A class that can be used to represent a car."""
```

```
class Car:  
    --snip--
```

Ahora podemos importar cada módulo de forma separada y crear cualquier tipo de coche que necesitemos:

```
from car import Car  
from electric_car import ElectricCar  
  
my_beetle = Car('volkswagen', 'beetle', 2019)  
print(my_beetle.get_descriptive_name())  
  
my_tesla = ElectricCar('tesla', 'roadster', 2019)  
print(my_tesla.get_descriptive_name())
```

Importamos *Car* desde su módulo y *ElectricCar* desde el suyo. A continuación creamos un coche tradicional y uno eléctrico. Ambos tipos de coches son creados correctamente:

```
2019 Volkswagen Beetle  
2019 Tesla Roadster  
  
Process finished with exit code 0
```

Uso de Alias.

Como viste en el capítulo 8, los alias pueden ser de mucha ayuda cuando usas módulos para organizar el código de tus proyectos. También puedes usar alias cuando importes clases.

Como ejemplo, considera un programa en el que deseas hacer un motón de coches eléctricos. Puede llegar a ser tedioso escribir (y

leer) *ElectricCar* una y otra vez. Puedes dar a *ElectricCar* un alias en la sentencia de importación:

```
from electric_car import ElectricCar as EC
```

Ahora puedes usar este alias siempre que quieras hacer un coche eléctrico:

```
my_tesla = EC('tesla', 'roadster', 2019)
```

Encontrar tu propio flujo de trabajo.

Como puedes ver, Python ofrece muchas opciones sobre cómo estructurar el código en un proyecto grande. Es importante conocer todas estas posibilidades para que puedas determinar las mejores formas de organizar tus proyectos, así como comprender los proyectos de otras personas.

Cuando estés empezando, mantén tu estructura de código de forma simple. Intenta hacerlo todo en un archivo y mueve tus clases a módulos separados una vez que todo esté funcionando. Si te gusta cómo interactúan los módulos y los archivos, prueba almacenar las clases en módulos al iniciar un proyecto. Encuentra un enfoque que te permita escribir código que funcione y empieza desde allí.

EJERCICIOS – TRY IT YOURSELF

9-10. Imported Restaurant: Utilizando tu última clase *Restaurant*, almacénala en un módulo. Haz un archivo separado que importe *Restaurant*. Haz una instancia *Restaurant* y llama a un método de *Restaurant* que muestre que la sentencia import está funcionando adecuadamente.

9-11. Imported Admin: Comienza tu trabajo desde el ejercicio 9-8. Almacena las clases *User*, *Privileges* y *Admin* en un módulo. Crea un archivo separado, haz una instancia de *Admin* y llama a

`show_privileges()` para mostrar que todo está funcionando correctamente.

9-12. Multiple Modules: Almacena la clase `User` en un módulo, y almacena las clases `Privileges` y `Admin` en un módulo separado. En un archivo separado, crea una instancia de `Admin` y llama a `show_privileges()` para mostrar que todo sigue funcionando correctamente.

La librería estándar de Python.

La librería estándar de Python es un conjunto de módulos incluidos con cada instalación de Python. Ahora que tienes un conocimiento básico de cómo trabajan las funciones y las clases, puedes empezar a usar módulos como estos que otros programadores han escrito. Puedes usar cualquier función o clase de la librería estándar incluyendo una simple sentencia `import` al principio de tu archivo. Echemos un vistazo a un módulo, `random`, que puede ser muy útil para modelar muchas situaciones del mundo real.

Una función interesante del módulo `random2` (*Python3.10*) es `randint()`. Esta función toma dos enteros como argumentos y devuelve un entero seleccionado aleatoriamente entre esos números (también incluidos).

Así es como se genera un número `random` entre 1 y 6:

```
from random2 import randint

print(randint(1, 6))
```

Otra función útil es `choice()`. Esta función toma una lista o tupla y devuelve un elemento elegido aleatoriamente:

```
from random2 import choice

players = ['charles', 'martina', 'michael', 'florence', 'eli']
first_up = choice(players)
print(first_up)
```

El modulo `random2` no se debe utilizar cuando estés codificando aplicaciones relacionadas con la seguridad, pero es lo

suficientemente bueno para mucho proyectos divertidos e interesantes.

NOTA: También puedes descargar módulos de fuentes externas. Verás un número de esos ejemplos en la Parte II, donde necesitaremos módulos externos para completar cada proyecto.

EJERCICIOS – TRY IT YOURSELF.

9-13. Dice: Haz una clase Dice(dado) con un atributo llamado sides, que tenga un valor por defecto de 6. Escribe un método llamado roll_dice() que imprima un número aleatorio entre 1 y el número de caras que tiene el dado. Haz un dado de 6 caras y hazlo rodar 6 veces.

Haz un dado de 10 caras y otro de 20 caras. Haz rodar cada uno 10 veces.

9-14. Lottery: Haz una lista o una tupla que contenga una serie de 10 números y cinco letras. Aleatoriamente, selecciona 4 números o letras de la lista e imprime un mensaje diciendo que cualquier boleto que coincida con esos cuatro números o letras ganará un premio.

9-15. Lottery Analysis: Puedes usar un bucle para comprobar lo difícil que puede ser ganar el tipo de lotería que acabas de modelar. Haz una lista o una tupla llamada my_ticket. Escribe un bucle que siga tirando números hasta que gane tu boleto. Imprime un mensaje que informe cuantas veces ha tenido que ejecutarse el bucle para darte un boleto ganador.

9-16. Python Module of the Week: Un excelente recurso para explorar la librería estándar de Python es un sitio llamado Python Module of the Week. Ve a <https://pymotw.com> y mira la tabla de contenidos. Encuentra un módulo que te interese y lee acerca de él, quizá empiece con el módulo random2.

Clases de estilo.

Vale la pena aclarar algunos problemas de estilo relacionados con las clases, especialmente a medida que tus programas se vuelven más complicados.

Los nombres de las clases pueden ser escritos en *CamelCase*. Para hacerlo, se ponen en mayúsculas la primera letra de cada palabra del nombre, y no se usan guiones bajos. Los nombres de instancia y módulos deben escribirse en minúsculas con guiones bajos entre las palabras.

Cada clase debe tener un *docstring* inmediatamente después de la definición de la clase. El *docstring* debe ser una breve descripción de lo que hace la clase, y debe seguir las mismas convenciones de formato que utilizaste para escribir los *docstrings* en las funciones. Cada módulo debe tener también un *docstring* definiendo para qué se puede usar las clases de un módulo.

Puedes usar líneas en blanco para organizar el código pero no lo hagas en exceso. Dentro de una clase puedes usar una línea en blanco entre métodos y dentro de un módulo puedes usar dos líneas en blanco para separar clases.

Si necesitas importar un módulo de la librería estándar y un módulo que hayas escrito, sitúa la sentencia `import` del módulo de la librería estándar primero. A continuación añade una línea en blanco y la instrucción `import` para el módulo que hayas escrito. En programas con múltiples sentencias `import`, esta convención hace más sencillo ver de dónde vienen los diferentes módulos usados en el programa.

Resumen.

En este capítulo hemos aprendido cómo escribir clases. Hemos aprendido cómo almacenar información en una clase utilizando atributos y cómo escribir métodos que den a tus clases el comportamiento que necesitan. Hemos aprendido a escribir métodos `__init__()` que crean instancias de tus clases con los atributos que quieras exactamente. Hemos visto cómo modificar los atributos de una instancia directamente a través de los métodos. Hemos aprendido que la herencia puede simplificar la creación de clases relacionadas entre sí y hemos aprendido a usar instancias de una

clase como atributos en otra clase para mantener cada clase de forma simple.

Hemos visto cómo almacenar clases en módulos e importar las clases que necesitas en los archivos en los que serán utilizadas para poder mantener tus proyectos organizados. Hemos empezado a conocer la librería estándar de Python y hemos visto un ejemplo basado en el módulo *random*. Finalmente hemos aprendido las clases de estilo utilizando las convenciones de Python.

En el capítulo 10, aprenderás a trabajar con archivos para que puedas guardar el trabajo que has hecho en un programa y lo que has permitido que hagan los usuarios. También aprenderás acerca de las excepciones y un clase especial de Python diseñada para ayudarte a responder a los errores cuando surjan.

Capítulo 10. Ficheros y Excepciones.

Ahora que has dominado las habilidades básicas que necesitas para escribir programas que sean fáciles de usar, es hora de pensar en hacer que tus programas sean aún más relevantes y usables. En este capítulo aprenderás a trabajar con ficheros para que tus programas pueden analizar rápidamente grandes cantidades de datos.

Aprenderás a capturar errores para que tus programas no rompan cuando se encuentren situaciones inesperadas. Aprenderás acerca de las excepciones, que son objetos especiales que crea Python para gestionar errores que surgen mientras un programa se está ejecutando. También aprenderás acerca del módulo *json*, que te permite salvar datos de usuario para que no se pierdan cuando tu programa se haya detenido.

Aprender a trabajar con ficheros y salvar datos, hará que tus programas sean más fáciles de utilizar para la gente. Los usuarios pueden elegir qué datos introducir y cuándo hacerlo. Los gente podrá ejecutar tu programa, realizar alguna tarea y luego cerrar el programa y continuar más tarde donde lo dejaron. Aprender a manejar las excepciones te ayudará a lidiar con situaciones en las que los archivos no existen y lidiar con otros problemas que pueden hacer que tus programas se bloqueen. Esto hará tus programas más robustos cuando encuentren datos incorrectos, ya provengan de errores inocentes o de intentos maliciosos de romper

tus programas. Con las habilidades que aprendas en este capítulo harás tus programas más utilizables, estables y aplicables.

Lectura de un archivo.

Hay una increíble cantidad de datos disponible en los archivos de texto. Los archivos de texto pueden contener datos del clima, de tráfico, socioeconómicos, de literatura y mucho más. Leer desde un fichero es particularmente útil en aplicaciones de análisis de datos, pero también es aplicable a cualquier situación en la que quieras analizar o modificar información almacenada en un fichero. Por ejemplo, puedes escribir un programa que lea el contenido de un archivo de texto y reescriba el archivo con un formato que permita mostrarlo a un navegador.

Cuando quieras trabajar con la información de un fichero de texto, el primer paso es leer el archivo en la memoria. Puedes leer todo el contenido de un fichero o puedes trabajar en el archivo una línea cada vez.

Lectura de un archivo completo.

Para empezar, necesitamos un archivo con unas cuantas líneas de texto en su interior. Empecemos con un fichero que contenga pi a 30 decimales, con 10 decimales por línea:

```
3.1415926535  
8979323846  
2643383279
```

Para intentar el siguiente ejemplo por ti mismo, puedes introducir estas líneas en un editor y salvar el fichero como *pi_digits.txt*, o puedes descargar el archivo de los recursos del libro a través de <https://nostarch.com/pythoncrashcourse2e/>. Guarda el fichero en el mismo directorio donde estás guardando los programas de este capítulo.

Aquí tienes un programa que abre un fichero, lo lee e imprime el contenido del fichero en la pantalla:

```
with open('pi_digits.txt') as file_object:  
    contents = file_object.read()  
    print(contents)
```

La primera línea de este programa tiene mucho que hacer. Empecemos mirando la función `open()`. Para realizar cualquier trabajo con un archivo, incluso si sólo es para imprimir su contenido, primero debes abrir el archivo para acceder a él. La función `open()` necesita un argumento: el nombre del fichero que quieras abrir. Python busca este fichero en el directorio donde se almacena el programa que se está ejecutando actualmente. En este ejemplo, `file_reader.py` se está ejecutando actualmente, por lo que Python busca `pi_digits.txt` en el directorio donde está almacenado `file_reader.py`. La función `open()` devuelve un objeto representando el fichero. Aquí, `open('pi_digits.txt')` devuelve un objeto representando `pi_digits.txt`. Python asigna este objeto a `file_object`, con el que trabajaremos más tarde en el programa. La palabra clave **`with`**, cierra el fichero una vez que no necesitamos acceder más a él.

Observa cómo llamamos a `open()` en este programa pero no a `close()`. Puedes abrir y cerrar este fichero llamando a `open()` y `close()`, pero si un error en el programa permite que se ejecute el método `close()`, el archivo nunca se podrá cerrar. Esto puede parecer trivial, pero los archivos que no se cierran correctamente, pueden causar pérdida de datos o corromperse. Y si llamas a `close()` demasiado pronto en tu programa, te encontrarás trabajando con un archivo cerrado (un archivo al que no puedes acceder), lo que conduce a más errores. No siempre es fácil saber exactamente cuándo debes cerrar un archivo, pero con la estructura mostrada aquí, Python lo resolverá por ti. Todo lo que tienes que hacer es abrir el archivo y trabajar con él de la forma que quieras, confiando en que Python lo cerrará automáticamente cuando el bloque **`with`** finalice su ejecución.

Una vez que tenemos un objeto de archivo que representa `pi_digits.txt`, usamos el método `read()` en la segunda línea de nuestro programa para leer todo el contenido del archivo y

almacenarlo como una cadena larga en `contents`. Cuando imprimimos el valor de `contents`, recuperamos todo el archivo de texto:

```
3.1415926535  
8979323846  
2643383279  
  
Process finished with exit code 0
```

La única diferencia entre esta salida y el archivo original, es la línea en blanco adicional al final de la salida. La línea en blanco aparece porque `read()` devuelve una cadena vacía cuando llega al final del archivo. Esta cadena vacía aparece como una línea en blanco. Si quieres eliminar la línea en blanco adicional, puedes usar `rstrip()` en la llamada a `print()`:

```
with open('pi_digits.txt') as file_object:  
    contents = file_object.read()  
print(contents.rstrip())
```

Recuerde que el método `rstrip()` de Python elimina o tira, cualquier espacio en blanco de la parte derecha de una cadena. Ahora la salida coincide exactamente con el contenido del fichero original:

```
3.1415926535  
8979323846  
2643383279  
  
Process finished with exit code 0
```

Rutas de archivo.

Cuando pasas un simple nombre de archivo como `pi_digits.txt` a la función `open()`, Python busca en el directorio donde está almacenado el archivo que se está ejecutando actualmente (es decir, tu archivo de programa `.py`).

A veces, dependiendo de cómo organices tu trabajo, el archivo que quieras abrir no estará en el mismo directorio que el archivo de

programa. Por ejemplo, puedes almacenar tus archivos de programa en una carpeta llamada *python_work*; dentro de *python_work*, puedes tener otra carpeta llamada *text_files* para distinguir tus archivos de programa de los archivos de texto que estés manipulando. Aunque *text_files* esté en *python_work*, simplemente pasando a *open()* el nombre de un fichero de *text_files*, no funcionará porque Python sólo buscará en *python_work* y se detendrá allí; no va a seguir y buscar en *text_files*. Para que Python abra archivos de un directorio distinto de aquel en el que está almacenado el archivo de programa, es necesario proporcionar una ruta de archivo que indique a Python que busque en una ubicación específica del sistema.

Puesto que *text_files* está dentro de *python_work*, puedes usar una ruta de archivo relativa para abrir un archivo desde *text_files*. Una ruta de archivo relativa le dice a Python que busque una ubicación determinada relativa al directorio en el que se encuentra almacenado el archivo del programa en ejecución. Por ejemplo, se escribiría:

```
with open('text_files/filename.txt') as file_object:
```

Esta línea le dice a Python que busque el archivo .txt deseado en la carpeta *text_files* y asume que *text_files* está ubicado dentro de *python_work* (que lo está).

NOTA: Los sistemas Windows utilizan una barra invertida en lugar de una barra inclinada(/) al mostrar rutas de archivos, pero todavía se pueden utilizar barras diagonales en tu código.

También le puedes decir a Python exactamente dónde está el archivo en tu ordenador, independientemente de dónde esté almacenado el programa que se está ejecutando. Esto es llamado ruta absoluta de archivo. Utiliza una ruta absoluta si una ruta relativa no funciona. Por ejemplo, si pones *text_files* en cualquier otra carpeta distinta de *python_work* - digamos, una carpeta llamada *other_files* - entonces pasando solamente la ruta '*text_files/filename.txt*' no funcionará porque Python sólo buscará en la ubicación dentro de *python_work*. Tendrás que escribir una ruta completa para aclarar dónde quieras que busque Python.

Las rutas absolutas son normalmente más largas que las rutas relativas, por lo que es útil asignarlos a una variable y luego pasar esa variable a `open()`:

```
file_path = '/home/ehmatthes/other_files/text_files/filename.txt'  
with open(file_path) as file_object:
```

Utilizando rutas absolutas, puedes leer ficheros desde cualquier localización de tu sistema. Por ahora lo más fácil es almacenar archivos en el mismo directorio que tus archivos de programa o en una carpeta como `text_files` dentro del directorio que almacene tus archivos de programa.

NOTA: Si intentas utilizar barras invertidas en una ruta de archivo obtendrás un error porque la barra invertida se utiliza para escapar caracteres en cadenas. Por ejemplo, en la ruta: “`C:\path\to\file.txt`”, la secuencia `\t` es interpretada como una tabulación. Si necesitas usar barras invertidas, puedes escapar cada una en la ruta así: “`C:\\\\patch\\\\to\\\\file.txt`”.

Leyendo línea a línea.

Cuando estás leyendo un fichero, con frecuencia querrás examinar cada línea del fichero. Es posible que busques determinada información en el fichero o que quieras modificar el texto del fichero de alguna manera. Por ejemplo, puede que quieras leer un archivo de datos meteorológicos y trabajar con cualquier línea que incluya la palabra `sunny` en la descripción del tiempo de ese día. En un informe de noticias puedes buscar cualquier línea con la etiqueta `<headline>` y reescribir esa línea con un tipo específico de formato.

Puedes usar un bucle `for` en el fichero objeto para examinar cada línea de un archivo de una en una:

```
filename = 'pi_digits.txt'  
  
with open(filename) as file_object:  
    for line in file_object:  
        print(line)
```

Primero asignamos el nombre del fichero desde el que estamos leyendo a la variable `filename`. Esto es una convención muy común cuando trabajas con ficheros. Puesto que la variable no representa el archivo real sino que es sólo una cadena que indica a Python dónde encontrar el archivo, se puede cambiar fácilmente `pi_digits.txt` por el nombre de otro archivo con el que quieras trabajar. Después de llamar a `open()` se asigna a la variable `file_object` un objeto que representa al fichero y su contenido. De nuevo usamos la sintaxis **`with`** para permitir que Python abra y cierre el fichero adecuadamente. Para examinar el contenido del fichero trabajamos con cada línea del fichero recorriendo todo el fichero objeto.

Cuando imprimimos cada línea, encontramos incluso más líneas en blanco:

```
3.1415926535

8979323846

2643383279

Process finished with exit code 0
```

Estas líneas en blanco aparecen porque hay un carácter invisible de nueva línea al final de cada línea en el archivo de texto. La función `print`, añade su propia nueva línea cada vez que la llamamos, por lo que terminamos con dos caracteres de nueva línea al final de cada línea: uno desde el fichero y otra desde `print()`. Utilizar `rstrip()` en cada línea de la llamada a `print()` elimina esas líneas en blanco extras:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Ahora la salida vuelve a coincidir con el contenido del fichero:

```
3.1415926535
8979323846
2643383279

Process finished with exit code 0
```

Hacer una lista de líneas de un fichero.

Cuando usas **with**, el objeto devuelto por `open()` sólo está disponible dentro del bloque **with** que lo contenga. Si quieres conservar el acceso al contenido del archivo fuera del bloque **with**, puedes almacenar las líneas del archivo en una lista dentro del bloque y luego trabajar con esa lista. Puedes procesar partes del archivo de forma inmediata o posponer parte del procesamiento en el programa para más adelante.

El siguiente ejemplo almacena las líneas de `pi_digits.txt` en una lista dentro del bloque **with** y luego imprime las líneas fuera del bloque **with**:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line.rstrip())
```

Primero el método `readlines()` toma cada línea de un fichero y la almacena en una lista. A continuación esta lista se asigna a `lines`, con la que podemos continuar trabajando después de que finalice el bloque **with**. A continuación usamos un sencillo bucle `for` para imprimir cada línea de `lines`. Puesto que cada elemento de `lines` se corresponde con cada línea del fichero, la salida coincide exactamente con el contenido del fichero.

Trabajar con el contenido de un fichero.

Después de leer un archivo en memoria, puedes hacer todo lo que quieras con esos datos, así que vamos a explorar brevemente los dígitos de `pi`. En primer lugar intentaremos construir una única

cadena que contenga todos los dígitos del archivo sin espacios en blanco:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.rstrip()

print(pi_string)
print(len(pi_string))
```

Empezamos abriendo el archivo y almacenando cada línea de dígitos en una lista, tal y como hicimos en el ejemplo anterior. Creamos la variable *pi_string* para contener los dígitos de pi. A continuación creamos un bucle que añada cada línea de dígitos a *pi_string* y elimine el carácter nueva línea de cada línea. Imprimimos esa cadena y también mostramos la longitud de esa cadena:

```
3.1415926535 8979323846 2643383279
36

Process finished with exit code 0
```

La variable *pi_digits* contiene el espacio en blanco que estaba a la izquierda de los dígitos en cada línea, pero podemos evitar esto usando *strip()* en lugar de *rstrip()*.

```
filename = 'pi_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.rstrip()
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

Ahora tenemos una cadena que contiene *pi* con 30 decimales:

```
3.141592653589793238462643383279
32

Process finished with exit code 0
```

NOTE: Cuando Python lee desde un fichero de texto, interpreta todo el texto del fichero como una cadena. Si lees un número y quieras trabajar con ese valor en un contexto numérico, tendrás que convertirlo en un entero usando la función *int()* o convertirlo a float usando la función *float()*.

Archivos grandes: Un millón de dígitos.

Hasta ahora nos hemos centrado en el análisis de un archivo de texto que sólo contiene 3 líneas, pero el código de estos ejemplos funcionaría igual de bien en archivos mucho más grandes. Si comenzamos con un archivo de texto que contenga pi con 1.000.000 de decimales en lugar de 30, podemos crear un cadena sencilla que contenga todos esos dígitos. No necesitamos cambiar nuestro programa en absoluto salvo pasarle un fichero diferente. También vamos a imprimir sólo los primeros 50 decimales, así que no tenemos que ver un millón de dígitos en la terminal:

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.strip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

La salida muestra que, efectivamente, tenemos una cadena que contiene pi hasta 1.000.000 de decimales.

```
3.14159265358979323846264338327950288419716939937510...
1000002

Process finished with exit code 0
```

Python no tienen límites relativos a la cantidad de datos con los que se puede trabajar; puedes trabajar con tantos datos como la memoria de tu equipo pueda manejar.

¿Contiene Pi tu cumpleaños?

Siempre he sentido curiosidad por saber si mi cumpleaños aparece en algún lugar de los dígitos de *Pi*. Usemos el programa que acabamos de escribir para encontrar si el cumpleaños de alguien aparece en cualquier lugar del primer millón de dígitos de *pi*. Podemos hacerlo expresando cada cumpleaños como una cadena de dígitos y ver si esa cadena aparece en algún lugar de *pi_string*:

```

filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.strip()

💡
birthday = input('Introduce tu cumple con el formato mmddyy: ')
if birthday in pi_string:
    print('Your birthday appears in the first million digits of pi!')
else:
    print('Your birthday does not appear in the first million digits of pi.')

```

Comenzamos solicitando el cumpleaños al usuario y luego comprobamos si esa cadena está en `pi_string`. Probémoslo:

```

Introduce tu cumple con el formato mmddyy: 122774
Your birthday does not appear in the first million digits of pi.

Process finished with exit code 0

```

Mi cumpleaños no aparece en los dígitos de pi, pero sí que aparece la fecha de cumpleaños del autor del libro:

```

Introduce tu cumple con el formato mmddyy: 120372
Your birthday appears in the first million digits of pi!

Process finished with exit code 0

```

Una vez que has leído de un fichero puedes analizar su contenido de cualquier forma que puedas imaginar.

EJERCICIOS – TRY IT YOURSELF.

10-1. Learning Python: Abre un fichero en blanco en tu editor de texto y escribe unas pocas líneas resumiendo que has aprendido de Python hasta ahora. Comienza cada línea con la frase *In Python you can...* Guarda el fichero como `learning_python.txt` en el mismo

directorio que los ejercicios de este capítulo. Escribe un programa que lea el fichero e imprima lo que has escrito tres veces. Imprime el contenido una vez leyendo el archivo completo, una vez recorriéndolo con un bucle sobre el objeto de archivo y otra vez almacenando las líneas en una lista y trabajando con ellas fuera del bloque.

10-2. Learning C: Puedes usar el método `replace()` para reemplazar una palabra en una cadena con otra palabra diferente. Aquí tienes un ejemplo rápido que te muestra cómo reemplazar ‘dog’ por ‘cat’ en una sentencia:

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

Lee en cada línea del nuevo fichero que acabas de crear, `learning_python.txt` y cambia la palabra `Python` por el nombre de otro lenguaje como `C`. Imprime por pantalla cada línea modificada.

Escribir en un fichero.

Una de las maneras más sencillas de guardar datos es escribiéndolos en un fichero. Cuando escribes texto en un fichero, la salida estará todavía disponible después de cerrar el terminal que contenga la salida de tu programa. Puedes examinar la salida después de que finalice la ejecución de un programa y también puedes compartir los archivos de salida con otras personas. También puedes escribir programas que vuelvan a leer el texto en la memoria y trabajar con él más tarde.

Escribir en un fichero vacío.

Para escribir texto en un fichero, necesitas llamar a `open()` con un segundo argumento diciéndole a Python lo que quieras escribir en el fichero. Para ver cómo funciona esto, escribamos un mensaje sencillo y almacenémoslo en un fichero en lugar de imprimirllo por pantalla:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

La llamada a `open()` en este ejemplo tiene dos argumentos. El primer argumento sigue siendo el nombre del fichero que queremos abrir. El segundo argumento `'w'`, le dice a Python que queremos abrir el fichero en modo escritura. Puedes abrir un fichero en modo lectura (`'r'`), modo escritura (`'w'`), modo añadir (`'a'`) o un modo que te permita leer y escribir el fichero (`'r+'`). Si omites del argumento del modo, Python abre el fichero en modo sólo-lectura por defecto.

La función `open()` crea automáticamente el archivo que estás escribiendo si no existe aún. Sin embargo, ten cuidado al abrir un fichero en modo escritura (`'w'`) porque si el fichero ya existe, Python eliminará el contenido del fichero antes de devolver el fichero objeto.

A continuación usamos el método `write()` en el fichero objeto para escribir una cadena en el fichero. Este programa no tiene salida por la terminal, pero si abres el fichero `programming.txt` verás esta línea.

```
I love programming.

Process finished with exit code 0
```

Este archivo se comporta como cualquier otro de tu computadora. Puedes abrirlo, escribir texto nuevo en él, copiarlo desde, pegarlo a, y así sucesivamente.

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

Escribir múltiples líneas.

La función `write()` no añade ninguna nueva línea al texto que escribes. Así que si escribes más de una línea sin incluir los caracteres de nueva línea, tu archivo puede no verse de la forma que quieras:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating games.")
```

Si abres `programming.txt` verás las dos líneas juntas:

```
I love programming.I love creating games.

Process finished with exit code 0
```

Incluir nuevas líneas en tu llamada a `write()` hace que cada cadena aparezca en su propia línea:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating games.\n")
```

La salida ahora muestra las líneas separadas:

```
I love programming.
I love creating games.

Process finished with exit code 0
```

También puedes usar espacios, caracteres de tabulación y líneas en blanco para formatear tu salida, tal y como lo has estado haciendo con la salida basada en la terminal.

Añadir a un fichero.

Si quieres añadir contenido a un fichero en lugar de sobrescribir el ya existente, puedes abrir el fichero en modo añadido (*append mode*). Cuando abres un fichero en modo *append*, Python no elimina los contenidos de l fichero antes de devolver el fichero objeto. Cualquier línea que escribas en el fichero se añadirá al final de este. Si el fichero no existe aún, Python creará un fichero vacío para ti.

Modifiquemos *write_message.py* añadiendo algunas nuevas razones por las que amamos programar, al fichero que ya existe '*programming.txt*':

```
filename = 'programming.txt'

# Añadimos cadenas al archivo de texto. No se sobrescribe, se añaden.
with open(filename, 'a') as file_object:
    file_object.write("I also love finding meaning in large datasets.\n")
    file_object.write("I love creating apps that can run in a browser.\n")
```

Usamos el argumento '*a*' para abrir el fichero en modo *append* en lugar de escribir sobre el archivo existente. Después escribimos dos nuevas líneas que son añadidas a *programming.txt*:

```
I love programming.
I love creating games.
I also love finding meaning in large datasets.
I love creating apps that can run in a browser.
```

Terminamos con el contenido original del fichero, seguido de nuevo contenido que acabamos de añadir.

EJERCICIOS - TRY IT YOURSELF.

10-3. Guest: Escribe un programa que pregunte al usuario por su nombre. Cuando responda, escribe su nombre en un fichero llamado *guest.txt*.

10-4. Guest Book: Escribe un bucle que pregunte al usuario por su nombre. Cuando introduzca su nombre, imprime un saludo en la pantalla y añade una línea recordándole su visita en un archivo llamado `guest_book.txt`. Asegúrate de que cada entrada aparece en una nueva línea en el fichero.

10-5. Programming Poll: Escribe un bucle que pregunte a las personas por qué les gusta programar. Cada vez que alguien introduzca una razón, añade esa razón a un fichero que almacene todas las respuestas.

Excepciones.

Python utiliza objetos especiales llamados excepciones para gestionar errores que surgen durante la ejecución de un programa. Cada vez que ocurre un error que hace que Python no sepa qué hacer a continuación, se crea un objeto excepción. Si escribes código que captura la excepción, el programa continuará ejecutándose. Si no capturas la excepción, el programa se detendrá y mostrará un *traceback* que incluya un reporte de la excepción que se ha producido.

Las excepciones son capturadas con bloques `try-except`. Un bloque `try-except` pide a Python hacer algo, pero también le dice a Python qué hacer si surge una excepción. Cuando usas bloques `try-except` tus programas continuarán ejecutándose, incluso si las cosas comienzan a ir mal. En lugar de *tracebacks*, que pueden ser confusos de leer para los usuarios, quienes verán más amigables mensajes de error escritos por ti.

Capturar la excepción de división por cero (`ZeroDivisionError`).

Veamos un sencillo error que hace que Python lance una excepción. Probablemente sepas que es imposible dividir por cero, pero preguntémosle a Python de todas formas:

```
>>> print(5/0)
```

Por supuesto que Python no puede hacerlo, por lo que obtenemos un *traceback*:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

El error mostrado en el traceback, `ZeroDivisionError`, es un objeto excepción. Python crea este tipo de objeto en respuesta a una situación donde no puede hacer lo que le pedimos. Cuando esto sucede, Python detiene el programa y nos dice el tipo de excepción que se ha producido. Podemos usar esta información para modificar nuestro programa. Le diremos a Python que hacer cuando ocurra este tipo de excepción; de esta forma, si vuelve a suceder, estaremos preparados.

Usar bloques `try-except`.

Cuando piensas que puede ocurrir un error, puedes escribir un bloque `try-except` para capturar la excepción que se pueda producir. Le dices a Python que intente ejecutar algún código, y le dices qué hacer si el código deriva en un tipo particular de excepción.

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

```
>>> try:
...     print(5/0)
... except:
...     print("No puedes dividir entre cero.")
...     raise
```

Situamos `print(5/0)`, que es la línea que causa el error, dentro del bloque `try`. Si el código dentro del bloque `try` funciona, Python se salta el bloque `except`. Si el código dentro del bloque `try` produce un error, Python busca un bloque `except` cuyo error coincida con el que ha producido y ejecuta el código contenido en ese bloque.

En este ejemplo, el código del bloque `try` produce un `ZeroDivisionError`, por lo que Python busca un bloque `except` que le diga cómo responder. Entonces Python ejecuta el código de ese bloque y el usuario ve un mensaje de error amigable en lugar de un informe de seguimiento del error o `traceback`:

```
No puedes dividir entre cero.
>>>
```

Si hay más código a continuación del bloque `try-except`, el programa puede continuar ejecutándose porque le dice a Python cómo capturar el error. Veamos un ejemplo en el que la captura de un error puede permitir a un programa continuar ejecutándose.

Uso de excepciones para evitar fallos.

Capturar errores correctamente es especialmente importante cuando el programa tiene más trabajo que hacer después de que ocurra el error. Esto suele ocurrir en programas que solicitan entrada de datos del usuario. Si el programa responde apropiadamente a una entrada incorrecta, puede solicitar más entradas válidas en lugar de fallar.

Vamos a crear un calculadora sencilla que sólo haga divisiones:

```
print("dame dos números y los dividiré.")
print("Introduce 'q' para terminar.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    answer = int(first_number) / int(second_number)
    print(answer)
```

Este programa pide al usuario que introduzca un `first_number` y, si el usuario no introduce '`q`', un `second_number`. A continuación dividimos esos dos números para obtener una respuesta (`answer`). Este programa no hace nada para capturar errores, por lo que si le pedimos que divide por cero, causaremos un fallo:

```
Dame dos números y los dividiré.
Introduce 'q' para terminar.

First number: 5
Second number: 0
Traceback (most recent call last):
  File "/home/usuario/PycharmProjects/pythonCrashCourse/capitulo_10_ficheros_y_excepciones/excepciones/division_calculator.py", line 13, in <module>
    answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Es malo que el programa falle, pero tampoco es una buena idea dejar que los usuarios vean el informe de error ya que los usuarios sin conocimientos técnicos se confundirán y en un entorno malicioso, los atacantes sabrán más de lo que quieras que sepan a partir de un rastreo (*traceback*). Por ejemplo, sabrán el nombre de tu archivo de programa, y verán una parte de tu código que no funciona correctamente. Un atacante experimentado podría utilizar a veces esta información para decidir qué tipo de ataques utilizar contra tu código.

El bloque else.

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

```
print("Dame dos números y los dividiré.")
print("Introduce 'q' para terminar.")

while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("No puedes dividir entre cero, subnormal.")
    else:
        print(answer)
```

Pedimos a Python que intente completar la operación división en el bloque *try*, que sólo incluye el código que podría causar el error. Cualquier código que dependa del éxito del bloque *try*, se añade al bloque *else*. En este caso, si la operación división es exitosa, usaremos el bloque *else* para imprimir el resultado.

El bloque `except` le dice a Python cómo responder cuando se produzca el error de la división por cero (`ZeroDivisionError`). Si el bloque `try` no tiene éxito debido a un error de división por cero, imprimimos un mensaje amigable diciéndole al usuario cómo evitar ese tipo de error. El programa continúa ejecutándose y el usuario nunca ve el registro de errores (`traceback`).

```
Dame dos números y los dividiré.  
Introduce 'q' para terminar.  
  
First number: 5  
Second number: 0  
No puedes dividir entre cero, subnormal.  
  
First number: 5  
Second number: 2  
2.5  
  
First number: q  
  
Process finished with exit code 0
```

El bloque `try-except-else` funciona así: Python intenta ejecutar el código en el bloque `try`. El único código que debe ir en un bloque `try` es el código que pueda provocar una excepción. A veces tendrás código adicional que deba ejecutarse sólo si el bloque `try` ha tenido éxito; este código va en el bloque `else`. El bloque `except` le dice a Python qué hacer en el caso de que se produzca una excepción cuando intente ejecutar el código dentro del bloque `try`. Al anticiparte a las posibles fuentes de error, puedes escribir programas robustos que sigan ejecutándose aunque se encuentren con datos no válidos o no encuentren los recursos que necesitan. Tu código será resistente a errores de usuarios inocentes y ataques maliciosos.

Capturar la excepción `FileNotFoundException`.

Un problema muy común cuando trabajas con ficheros es la gestión de ficheros que faltan o no se encuentran. El archivo que estás buscando puede estar en una ubicación diferente, el nombre del archivo puede estar mal escrito o simplemente puede que el fichero

no existe. Puedes manejar todas estas situaciones directamente con un bloque *try-except*.

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

```
filename = 'alice.txt'

with open(filename, encoding='utf-8') as f:
    contents = f.read()
```

Hay dos cambios aquí. Uno es el uso de la variable '*f*' para representar al objeto fichero, lo que es una convención muy común. EL segundo es el uso del argumento '*encoding*'. Este argumento se necesita cuando la codificación por defecto tu sistema no coincide con la codificación del archivo que se está leyendo.

Python no puede leer de un fichero que no existe, por lo que se produce una excepción:

```
Traceback (most recent call last):
  File "/home/usuario/PycharmProjects/pythonCrashCourse/capitulo_10_ficheros_y_excepciones/excepciones/alice.py", line 5, in <module>
    with open(filename, encoding='utf-8') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'

Process finished with exit code 1
```

La última línea del rastreo informa de un *FileNotFoundException*: esta es la excepción que crea Python cuando no puede encontrar el fichero que está intentando abrir.

En este ejemplo, la función *open()* es la que produce el error, así que para capturarla, el bloque *try* comenzará con la línea que contiene *open()*:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"Lo siento, el fichero {filename} no existe.")
```

En este ejemplo, el código del bloque `try` produce un `FileNotFoundException`, así que Python busca un bloque `except` que coincida con el error. Entonces Python ejecuta el código de ese bloque y el resultado es un mensaje de error amigable en lugar de un informe de errores o rastreo (*traceback*):

```
A ver cómo te lo digo gilipollas, el fichero alice.txt no existe.  
Process finished with exit code 0
```

El programa no tiene nada más que hacer si el fichero no existe, por lo que el código que captura el error no añade más a este programa. Continuemos con este ejemplo y veamos cómo el manejo de excepciones puede ayudar cuando se trabaja con más de un archivo.

Anализar тексто.

Puedes analizar archivos de texto que contengan libros enteros. Muchas obras clásicas de la literatura están disponibles como archivos de texto simples porque son de dominio público. El texto utilizado en esta sección viene del proyecto **Gutenberg** (<http://gutenberg.org/>). El proyecto Gutenberg mantiene una colección de obras literarias que están disponibles para el público y es un gran recurso si estás interesado en trabajar con textos literarios en tus proyectos de programación.

Saquemos el texto de Alicia en el País de las Maravillas e intentemos contar el número de palabras del texto. Usaremos el método de cadenas **`split()`**, que puede formar una lista de palabras de una cadena. Esto es lo que hace **`split()`** con una cadena que contiene sólo el título “Alicia en el País de las Maravillas”:

```
>>> title = "Alice in Wonderland"  
>>> title.split()  
['Alice', 'in', 'Wonderland']  
>>>
```

El método **`split()`** separa una cadena en partes siempre que encuentre un espacio y almacena todas esas partes de la cadena en una lista. El resultado es una lista de palabras de la cadena, aunque también pueden aparecer signos de puntuación con algunas palabras. Para contar el número de palabras en *Alice in*

Wonderland, usaremos **`split()`** sobre todo el texto. Luego contaremos los elementos de la lista para hacernos una idea aproximada del número de palabras del texto:

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"A ver cómo te lo digo gilipollas, el fichero {filename} no existe.")
else:
    # Contamos el número aproximado de palabras del fichero.
    words = contents.split()
    num_words = len(words)
    print(f"Comemierdas, el fichero {filename} tiene más o menos {num_words} palabras. "
          f"Más o menos, joder que lo quieras saber todo.")
```

Movemos el archivo *alice.txt* al directorio correcto por lo que el bloque *try* funcionará esta vez. Cogemos la cadena de *contents*, que ahora contiene todo el texto de *Alice in wonderland* como una gran cadena, y usamos el método **`split()`** para producir una lista de todas las palabras del libro. Cuando usamos *len()* sobre esta lista para saber la longitud, obtenemos una buena aproximación del número de palabras en la cadena original. Imprimimos la frase que informa cuántas palabras se han encontrado en el archivo. Este código se ubica en el bloque *else* debido a que funcionará sólo si el código del bloque *try* se ha ejecutado correctamente. La salida nos dice cuántas palabras hay en *alice.txt*:

```
Comemierdas, el fichero alice.txt tiene más o menos 29590 palabras.
Más o menos, joder que lo quieras saber todo.

Process finished with exit code 0
```

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

Trabajar con múltiples ficheros.

Añadamos más libros para analizar. Pero antes de hacerlo, vamos a mover el grueso de este programa a una función llamada `count_words()`. De este modo, será más fácil realizar el análisis para varios libros:

```
new*
def count_words(filename):
    """Cuenta el número aproximado de palabras en un fichero."""
    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"A ver cómo te lo digo gilipollas, el fichero {filename} no existe.")
    else:
        # Contamos el número aproximado de palabras del fichero.
        words = contents.split()
        num_words = len(words)
        print(f"Comemierdas, el fichero {filename} tiene más o menos {num_words} palabras.\n"
              f"Mas o menos, joder que lo quieras saber todo.")

filename = 'alice.txt'
count_words(filename)
```

La mayoría del código no ha cambiado. Simplemente lo hemos indentado y movido al cuerpo de `count_words()`. Es un buen hábito conservar los comentarios actualizado cuando se modifica un programa, así que hemos cambiado el comentario a un docstring y lo hemos modificado ligeramente.

Ahora podemos escribir un bucle sencillo para contar las palabras en cualquier texto que queramos analizar. Haremos esto almacenando los nombres de los ficheros que queremos analizaren una lista, y luego llamaremos a `count_words()` para cada fichero de la lista. Intentaremos contar las palabras de *Alice in Wonderland*, *Siddartha*, *Moby Dick* y *Little Women* que están todos disponibles de forma pública. Dejaremos *Siddartha* intencionadamente fuera del directorio que contiene `word_count.py` para que podamos ver lo bien que maneja nuestro programa un archivo que falta:

```

new*

def count_words(filename):
    """Cuenta el número aproximado de palabras en un fichero."""
    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        print(f"A ver cómo te lo digo gilipollas, el fichero {filename} no existe.")
    else:
        # Contamos el número aproximado de palabras del fichero.
        words = contents.split()
        num_words = len(words)
        print(f"Comemierdas, el fichero {filename} tiene más o menos {num_words} palabras.\n"
              f"Más o menos, joder que lo quieres saber todo.")

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)

```

El fichero `siddhartha.txt` no tiene efectos sobre el resto de la ejecución del programa:

```

Comemierdas, el fichero alice.txt tiene más o menos 29590 palabras.
Más o menos, joder que lo quieres saber todo.
A ver cómo te lo digo gilipollas, el fichero siddhartha.txt no existe.
Comemierdas, el fichero moby_dick.txt tiene más o menos 215864 palabras.
Más o menos, joder que lo quieres saber todo.
Comemierdas, el fichero little_women.txt tiene más o menos 189141 palabras.
Más o menos, joder que lo quieres saber todo.

Process finished with exit code 0

```

Utilizando el bloque `try-except` en este ejemplo proporcionamos dos avances significativos. Prevenimos que nuestros usuarios vean el rastreo (`traceback`) y dejamos que el programa continúe analizando los textos que es capaz de encontrar. Si no capturamos el `FileNotFoundException` que produce `siddhartha.txt`, el usuario podría ver el rastreo completo y el programa detendría su ejecución después de intentar analizar *Siddhartha*. Nunca analizaría *Moby Dick* o *Little Women*.

Fallar en silencio.

En el ejemplo anterior, informamos a los usuarios que uno de los ficheros no estaba disponible. Pero no necesitas informar de cada excepción que captures. A veces querrás que el programa erre silenciosamente cuando ocurra una excepción y continúe como si no hubiera pasado nada. Para hacer que un programa falle de forma silenciosa, escribe un bloque `try` como de costumbre, pero dile explícitamente a Python que no haga nada en el bloque `except`. Python tiene una sentencia **`pass`** que dice que no haga nada en un bloque:

```
new*
def count_words(filename):
    """Cuenta el número aproximado de palabras en un fichero."""
    try:
        with open(filename, encoding='utf-8') as f:
            contents = f.read()
    except FileNotFoundError:
        pass
    else:
        # Contamos el número aproximado de palabras del fichero.
        words = contents.split()
        num_words = len(words)
        print(f"Comiendo, el fichero {filename} tiene más o menos {num_words} palabras.\n"
              f"Mas o menos, joder que lo quieras saber todo.")

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']
for filename in filenames:
    count_words(filename)
```

La única diferencia entre este listado y el anterior es la sentencia **`pass`**. Ahora, cuando se produce un `FileNotFoundException`, el código del bloque `except` se ejecuta, pero no ocurre nada. No se genera un rastreo y no hay salida en respuesta al error que se ha producido. Los usuarios ven el conteo de palabras para cada archivo que exista, pero no ven ninguna indicación de que no se encontró un archivo:

```
Comecierdas, el fichero alice.txt tiene más o menos 29590 palabras.  
Más o menos, joder que lo quieras saber todo.  
Comecierdas, el fichero moby_dick.txt tiene más o menos 215864 palabras.  
Más o menos, joder que lo quieras saber todo.  
Comecierdas, el fichero little_women.txt tiene más o menos 189141 palabras.  
Más o menos, joder que lo quieras saber todo.  
  
Process finished with exit code 0
```

La sentencia **pass** también actúa como marcador de posición. Es como un recordatorio de que has elegido no hacer nada en este punto específico de la ejecución del programa y que podrías querer hacer algo más tarde. Por ejemplo, en este programa podríamos decidir escribir cualquier nombre de archivo que falte en un archivo llamado *missing_files.txt*. Los usuarios no verían este archivo pero nosotros si podríamos leerlo y tratar cualquier texto que falte.

Decidir qué errores reportar.

¿Cómo saber cuándo informar de un error a tus usuarios y cuando fallar en silencio? Si los usuarios saben qué textos se supone que deben ser analizados, podrían agradecer un mensaje que les informe de por qué no se han analizado algunos textos. Si los usuarios esperan ver algunos resultado, pero no saben qué libros deben analizarse, es posible que no necesiten saber que algunos textos no estaban disponibles. Dar a los usuarios información que no están buscando puede hacer que la usabilidad de tu programa sea menor. Las estructuras de gestión de errores de Python te ofrecen un control preciso sobre cuánto compartir con los usuarios cuando las cosas fallen. Tú decides cuánta información compartes.

El código bien escrito y probado es poco propenso a errores internos como los de sintaxis o los lógicos. Pero cada vez que tu programa dependa de cuestiones externas como la entrada de usuario, la existencia de un fichero o la disponibilidad de una conexión de red, hay una posibilidad de que se produzca una excepción. Una pequeña experiencia que te ayudará a saber dónde incluir los bloques de captura de excepciones en tu programa y cuánta información debes dar a tus usuarios acerca de los errores producidos.

EJERCICIOS – TRY IT YOURSELF.

10-6. Addition: Un problema habitual cuando se solicita la inserción de datos numéricos, ocurre cuando la gente proporciona texto en lugar de números. Cuando intentas convertir la entrada a un *int*, obtendrás un *ValueError*. Escribe un programa que pida dos números. Súmalos e imprime el resultado. Captura el *ValueError* si cualquiera de los valores de entrada no es un número e imprime un mensaje de error amigable. Prueba tu programa introduciendo dos números y luego introduciendo algo de texto en lugar de un número.

10-7. Addition Calculator: empaqueta el código del ejercicio 10-6 en un bucle *while* para que el usuario pueda seguir introduciendo números incluso si comete un error e introduce texto en lugar de un número.

10-8. Cats and Dogs: Haz dos ficheros, *cats.txt* y *dogs.txt*. Almacena al menos tres nombres de gatos en el primer fichero y tres nombres de perro en el segundo. Escribe un programa que intente leer esos ficheros e imprima el contenido por pantalla. Envuelve tu código en un bloque *try-except* para capturar un error de archivo no encontrado (*FileNotFoundException*) e imprime un mensaje amigable si el archivo está perdido. Mueve uno de los archivos a una ubicación diferente en tu equipo y asegúrate de que el código del bloque *except* se ejecuta correctamente.

10-9. Silent Cats and Dogs: modifica el bloque *except* del ejercicio 10-8 para que falle silenciosamente si falta alguno de los archivos.

10-10. Common Words: Visita el proyecto (<https://gutenberg.org/>) y encuentra unos cuantos textos que te gusten para analizar. Descarga los archivos de texto de estas obras o copia el texto en bruto desde tu navegador a un archivo de texto en tu ordenador. Puedes usar el método *count()* para encontrar cuántas veces aparece una frase o una palabra en una cadena. Por ejemplo, el siguiente código cuenta el número de veces que aparece ‘row’ en una cadena:

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

Observa que convirtiendo la cadena a minúscula usando `lower()` captura todas las apariciones de la palabra que estás buscando independientemente de su formato.

Escribe un programa que lea los ficheros que has encontrado en *Project Gutenberg* y determina cuántas veces aparece la palabra ‘the’ en cada texto. Esto será una aproximación puesto que también contará palabras como ‘then’ y ‘there’. Intenta contar ‘the’, con un espacio en la cadena, y observa como se reduce el número de coincidencias.

Almacenamiento de datos.

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

El módulo `json` permite volcar estructuras de datos sencillas de Python en un archivo y cargar los datos de ese archivo la próxima vez que se ejecute el programa. También puedes utilizar `json` para compartir datos entre diferentes programas de Python. Incluso mejor, el formato de datos **JSON** no es específico de Python, por lo que puedes compartir datos que hayas almacenado en el formato **JSON** con gente que trabaje en muchos otros lenguajes de programación. Es un formato muy útil, portable y sencillo de aprender.

NOTA: El formato JSON (*JavaScript Object Notation*) fue desarrollado originalmente para JavaScript, sin embargo desde entonces se ha convertido en un formato muy común usado por muchos lenguajes, incluido Python.

Usando `json.dump()` y `json.load()`

Escribamos un pequeño programa que almacene un conjunto de números y otro programa que lea esos números de nuevo en la memoria. El primer programa usará `json.dump()` para almacenar el conjunto de números y el segundo programa utilizará `json.load()`.

La función `json.dump()` toma dos argumentos: la parte de los datos para almacenar y fichero objeto que se puede usar para almacenar los datos. Aquí está cómo puedes usar `json.dump()` para almacenar una lista de números:

```
import json

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

filename = 'numbers.json'
with open(filename, 'w') as f:
    json.dump(numbers, f)
```

Primero importamos el módulo `json` y luego creamos una lista de números para trabajar con ellos. Escogemos un nombre de archivo para almacenar la lista de números. Es habitual utilizar la extensión de archivo `.json` para indicar que los datos del archivo están almacenados en formato **JSON**. Entonces abrimos el archivo en modo escritura (`'w'`), lo que permite a `json` escribir datos en el archivo. Usamos la función `json.dump()` para almacenar la lista `numbers` en el archivo `numbers.json`.

Este programa no tiene salida, pero si abrimos el archivo `numbers.json` y miramos su contenido veremos que los datos se almacenan en un formato similar al de Python:

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

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

```
import json

filename = 'numbers.json'
with open(filename) as f:
    numbers = json.load(f)

print(numbers)
```

Primero nos aseguramos de leer desde el mismo archivo en el que escribimos. Esta vez, cuando abrimos el archivo, lo abrimos en modo lectura porque Python sólo necesita leer desde el archivo. Después utilizamos la función `json.load()` para cargar la información almacenada en `numbers.json`, y lo asignamos a la variable `numbers`. Finalmente imprimimos la lista de números recuperada y vemos que es la misma lista creada en `number_writer.py`:

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

Process finished with exit code 0
```

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

Guardar y leer datos generados por el usuario.

Salvar datos con `json` es muy útil cuando estás trabajando con datos generados por el usuario, porque si no almacenas la información del usuario de alguna manera, la perderás cuando el programa se detenga. Echemos un vistazo a un ejemplo donde pedimos al usuario que introduzca su nombre la primera vez que arranquen el programa y luego recordar su nombre cuando arranquen el programa de nuevo.

Empecemos almacenando el nombre de usuario:

```
import json

username = input('Introduce tu nombre: ')

filename = 'username.json'
with open(filename, 'w') as f:
    json.dump(username, f)
    print(f"Te recordaremos cuando vuelvas, {username}")
```

Pedimos un nombre de usuario para almacenar. A continuación utilizamos `json.dump()`, pasándole un nombre de usuario y un objeto de archivo para almacenar el nombre de usuario en el archivo. Luego imprimimos un mensaje informando al usuario que hemos almacenado su información:

```
Introduce tu nombre: JuanJe
Te recordaremos cuando vuelvas, JuanJe

Process finished with exit code 0
```

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

```
import json

filename = 'username.json'

with open(filename) as f:
    username = json.load(f)
    print(f"Bienvenido de nuevo, {username}!")
```

Primero usamos `json.load()` para leer la información almacenada en `username.json` y asignarla a la variable `username`. Ahora que hemos recuperado el nombre de usuario, podemos dar de nuevo la bienvenida:

```
Bienvenido de nuevo, JuanJe!  
Process finished with exit code 0
```

Necesitamos combinar esos dos programas en un archivo. Cuando alguien ejecute `remember_me.py`, queremos recuperar su nombre de usuario de la memoria si es posible, por lo tanto, empezaremos con un bloque `try` que intente recuperar el nombre de usuario. Si el archivo `username.json` no existe, haremos que el bloque `except` pida un nombre de usuario y lo almacene en `username.json` para la próxima vez:

```
import json  
  
# Carga username si ha sido almacenado previamente.  
# En otro caso, solicita el nombre de usuario y lo almacena.  
  
filename = 'username.json'  
  
try:  
    with open(filename) as f:  
        username = json.load(f)  
except:  
    username = input("¿Cómo te llamas?: ")  
    with open(filename, 'w') as f:  
        json.dump(username, f)  
        print(f"Te recordaremos la próxima ve que vuelvas, {username}!")  
else:  
    print(f"Bienvenido de nuevo, {username}!")
```

Aquí no hay código nuevo; hemos combinado los bloques de código de los dos últimos ejemplos en un archivo. Primero intentamos abrir el archivo `username.json`. Si existe, volvemos a leer el nombre de usuario de la memoria y volvemos a imprimir un mensaje dándole al bienvenida al usuario en el bloque `else`. Si esta es la primera vez que el usuario ejecuta el programa, `username.json` no existirá ocurrirá un `FileNotFoundException`. Python se moverá al bloque `except` donde pedirá al usuario que introduzca su nombre. A continuación usamos `json.dump()` para almacenar el nombre de usuario e imprimir un saludo.

Cualquiera que sea el bloque que se ejecuta, el resultado es un nombre de usuario y un saludo adecuado. Si es la primera vez que se ejecuta el programa, la salida es esta:

```
Introduce tu nombre: JuanJe
Te recordaremos cuando vuelvas, JuanJe

Process finished with exit code 0
```

De otra manera:

```
Bienvenido de nuevo, JuanJe!

Process finished with exit code 0
```

Esta es la salida que verás si el programa ya se ha ejecutado al menos una vez.

Refactorizar.

A menudo llegarás a un punto en que tu código funcionará, pero reconocerás que podrías mejorar el código dividiéndolo en una serie de funciones que tienen tareas específicas. Este proceso es llamado refactorización. La refactorización hace tu código más limpio, más comprensible y más fácil de extender.

Podemos refactorizar *remember_me.py* trasladando la mayor parte de su lógica a una o más funciones. El objetivo de *remember_me.py* es saludar al usuario, así que vamos a mover todo nuestro código existente en una función llamada *greet_user()*:

```
import json

new*
def greet_user():
    """Saluda al usuario por su nombre."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        username = input("¿Cómo te llamas?: ")
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"Te recordaremos cuando vuelvas, {username}!")
    else:
        print(f"Bienvenido de nuevo, {username}!")

greet_user()
```

Puesto que ahora estamos utilizando una función, actualizaremos los comentarios con un docstring que refleje cómo funciona el programa actualmente. Este archivo es un poco más limpio, pero la función `greet_user()` hace más que sólo saludar al usuario – también recupera un nombre de usuario almacenado si existe y solicita un nuevo nombre de usuario si no existe.

Vamos a refactorizar `greet_user()` para que no haga tantas tareas diferentes.

Empezaremos moviendo el código que recupera un nombre de usuario almacenado, en una función separada:

```

import json

new*
def get_stored_username():
    """Obtiene el nombre de usuario si ya está almacenado."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        return None
    else:
        return username

new*
def greet_user():
    """Saluda al usuario por su nombre."""
    username = get_stored_username()
    if username:
        print(f"Bienvenido de nuevo, {username}")
    else:
        username = input("¿Cómo te llamas?: ")
        filename = 'username.json'
        with open(filename, 'w') as f:
            json.dump(username, f)
        print(f"Te recordaremos cuando vuelvas, {username}!")

greet_user()

```

La nueva función `get_stored_username()` tiene un claro propósito, como se indica en su docstring. Esta función recupera un nombre de usuario almacenado y lo devuelve si lo encuentra. Si el archivo `username.json` no existe, la función devuelve `None`. Esta es una buena práctica: una función puede devolver, o bien el valor que estás esperando o puede devolver `None`. Imprimimos un mensaje de bienvenida el usuario si el intento de recuperar el nombre de usuario ha sido exitoso, y si no, le pedimos un nuevo nombre de usuario.

Deberíamos refactorizar un bloque más de código fuera de `greet_user()`. Si el nombre de usuario no existe, podemos mover el código que pide al usuario un nuevo nombre de usuario a una función dedicada para ese propósito:

Cada función en esta versión final de `remember_me.py` tiene un sencillo y claro propósito. Llamamos a `greet_user()`, y esa función imprime un mensaje apropiado: devuelve la bienvenida a un usuario existente o saluda a un nuevo usuario. Esto se hace llamando a `get_stored_username()`, que sólo es responsable de recuperar un nombre de usuario almacenado si existe. Por último, `greet_user()` llama a `get_user_name()` si es necesario, que es la responsable sólo de obtener un nuevo usuario y almacenarlo. Esta compartimentalización del trabajo es una parte esencial de escritura de código limpio que será más fácil de mantener y extender.

EJERCICIOS – TRY IT YOURSELF.

10-11. Favorite Number: Escribe un programa que pida al usuario su número favorito. Usa `json.dump()` para almacenar este número en un archivo. Escribe un programa separado que lea en este valor e imprima el mensaje: “Se tu número favorito!, es___. ”.

10-12. Favorite Number Remembered: Combina los dos programas del ejercicio 10-11 en un fichero. Si el número ya está almacenado, informa al usuario del número favorito. Si no lo está, solicita el usuario su número favorito y almacénalo en un archivo. Ejecuta el programa dos veces para ver si funciona.

10-13. Verify User: El listado final de `remember_me.py` asume que el usuario ya ha introducido su nombre de usuario o que el programa se está ejecutando por primera vez. Deberíamos modificarlo en caso de que el usuario actual no sea la persona que utilizó el programa por última vez.

Antes de imprimir un mensaje de bienvenida en `greet_user()`, pregunta al usuario si este es el nombre de usuario correcto. Si no lo es, llama a `get_new_username()` para obtener el nombre de usuario correcto.

Resumen.

En este capítulo hemos aprendido cómo trabajar con ficheros. Hemos aprendido a leer un fichero todo de una vez y a leer el contenido del fichero línea a línea. Hemos aprendido a escribir en un

fichero y añadir texto al final del fichero. Has leído acerca de las excepciones y cómo manejar las excepciones que es probable que veas en tus programas. Finalmente hemos aprendido cómo almacenar estructuras de datos de Python para que podamos guardar la información que nuestros usuarios proporcionen, evitando que tengan que empezar de nuevo cada vez que ejecutan un programa. En el capítulo 11 aprenderás formas eficientes de probar tu código. Esto te ayudará a confiar en que el código que has desarrollado es correcto, y te ayudará a identificar los errores que se introducen a medida que continúas desarrollando los programas que has escrito.

Capítulo 11. Prueba del código. Testing your code.

Cuando escribes una función o una clase, también puedes escribir pruebas para ese código. Las pruebas demuestran que tu código funciona como debería en respuesta a todos los tipos de entradas que está diseñado para recibir. Cuando escribes tests puedes estar seguro de que tu código funcionará correctamente a medida que más personas empiecen a utilizar tus programas. También podrás probar el nuevo código a medida que lo vayas escribiendo para asegurarte de que los cambios no alteran el comportamiento del programa. Cada programador comete errores, por lo que todos los programadores deben probar su código con frecuencia, capturando problemas antes de que los encuentren los usuarios.

En este capítulo aprenderás cómo probar tu código utilizando herramientas en el módulo `unittest` de Python. Aprenderás a construir un caso de prueba y a comprobar que un conjunto de entradas da como resultado la salida deseada. Verás qué aspecto tiene una prueba que funciona y cual una prueba que falla y aprenderás cómo una prueba que falla puede ayudarte a mejorar tu código. Aprenderás a probar funciones y clases y empezarás a entender cuántas pruebas tendrás que escribir para un proyecto.

Probar una función.

Para aprender sobre pruebas, necesitamos código que probar. Aquí tenemos una función sencilla que recibe un nombre y un apellido y devuelve un nombre completo bien formateado:

```
def get_formatted_name(first, last):
    """Genera un nombre completo bien formateado."""
    full_name = f"{first} {last}"
    return full_name.title()
```

La función `get_formatted_name()` combina un nombre y un apellido con un espacio en medio para generar un nombre completo y a continuación pone la primera letra en mayúsculas y devuelve ese nombre completo. Para comprobar que `get_formatted_name()` funciona, hagamos un programa que utilice esa función. El programa `names.py` permite al usuario introducir un nombre y un apellido y ver el nombre completo bien formateado:

```
from name_function import get_formatted_name

print("Introduce 'q' en cualquier momento para terminar.")
while True:
    first = input("\nDime tu nombre: ")
    if first == 'q':
        break
    last = input("Dime tu apellido: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNombre bien formateado: {formatted_name}")
```

Este programa importa `get_formatted_name()` desde `name_function.py`. El usuario puede introducir una serie de nombres y apellidos y ver los nombres que se han generado, formateados:

```
Introduce 'q' en cualquier momento para terminar.

Dime tu nombre: Guido
Dime tu apellido: Van Rossum
    Nombre bien formateado: Guido Van Rossum

Dime tu nombre: Linus
Dime tu apellido: Torvalds
    Nombre bien formateado: Linus Torvalds

Dime tu nombre: q

Process finished with exit code 0
```

Podemos ver que los nombres generados aquí son correctos. Pero digamos que queremos modificar `get_formatted_name()` para que también pueda tener segundos nombres. Al hacerlo, queremos asegurarnos de no romper la forma en que la función maneja los nombres que sólo tienen nombre y apellido. Podríamos probar nuestro código ejecutando `names.py` e introduciendo un nombre como *Janis Joplin* cada vez que modificamos `get_formatted_name()`, pero eso sería tedioso. Afortunadamente, Python proporciona una manera eficiente para automatizar las pruebas de salida de una función. Si automatizamos las pruebas de `get_formatted_name()` podemos estar seguros de que la función funcionará cuando se le den los tipos de nombres para los que hemos escrito pruebas.

Pruebas unitarias y casos de prueba.

El módulo `unittest` de la librería estándar de Python proporciona herramientas para probar tu código. Un test unitario comprueba que un aspecto específico del comportamiento de una función es correcto. Un caso de prueba es una colección de test unitarios que, en conjunto, demuestran que una función se comporta como se supone que debe hacerlo en todo el rango de situaciones que se espera que maneje. Un buen caso de prueba considera todos los posibles tipos de entrada que una función puede recibir e incluye pruebas para representar cada una de esas situaciones. Un caso de prueba con cobertura total incluye un rango completo de pruebas unitarias que cubren todas las formas posibles de utilizar una función. Conseguir una cobertura total en un proyecto de gran envergadura puede resultar desalentador. A menudo es suficiente con escribir pruebas para los comportamientos críticos del código y luego aspirar a una cobertura completa sólo si el proyecto empieza a tener un uso generalizado.

Un test superado.

Es necesario acostumbrarse a la sintaxis para configurar un caso de prueba, pero una vez que has configurado el caso de prueba es sencillo añadir más pruebas unitarias para tus funciones. Para escribir un caso de prueba para una función, importa el módulo `unittest` y la función que quieras probar. Luego, crea una clase

que herede de `unittest.TestCase` y escribe una serie de métodos para probar diferentes aspectos del comportamiento de tu función. Aquí tenemos un caso de prueba con un método que verifica que la función `get_formatted_name()` funciona correctamente cuando recibe un nombre y un apellido:

```
import unittest
from name_function import get_formatted_name

new *
class NamesTestCase(unittest.TestCase):
    """Pruebas para name_function.py"""

    new *
    def test_first_last_name(self):
        """¿Funcionan nombres como Linus Torvalds?"""
        formatted_name = get_formatted_name('linus', 'torvalds')
        self.assertEqual(formatted_name, 'Linus Torvalds')

if __name__ == '__main__':
    unittest.main()
```

Primero importamos `unittest` y la función que queremos probar, `get_formatted_name()`. A continuación creamos una clase llamada `NamesTestCase`, que contendrá una serie de pruebas unitarias para `get_formatted_name()`. Puedes dar a la clase el nombre que quieras, pero es mejor llamarla con algo relacionado con la función que estás probando y usar la palabra `Test` en el nombre de la clase. Esta clase debe heredar de la clase `unittest.TestCase`, así Python sabrá cómo ejecutar las pruebas que tú escribas.

`NameTestCase` contiene un método sencillo que prueba un aspecto de `get_formatted_name()`. Llamamos a este método `test_first_last_name()` porque estamos verificando que los nombres con sólo un nombre y un apellido se formatean correctamente. Cualquier método que empiece con `test_` se ejecutará automáticamente cuando ejecutemos `test_name_function.py`. Con este método de prueba podemos llamar a la función que queramos probar. En este ejemplo llamamos a `get_formatted_name()` con los argumentos `'linus'` y `'torvalds'`, y asignamos el resultado a `formatted_name`. Después utilizamos una de las características más útiles de `unittest`: un método `assert`. El método `assert` comprueba que el

resultado que recibes coincide con el resultado que esperas recibir. En este caso, como sabemos que `get_formatted_name()` se supone que devuelve un nombre completo en mayúsculas y correctamente espaciado, esperamos que el valor de `formatted_name` sea `Linus Torvalds`. Para comprobar si esto es verdad, usamos el método `assertEqual()` de `unittest` y le pasamos `formatted_name` y '`Linus Torvalds`'. La línea

```
self.assertEqual(formatted_name, 'Linus Torvalds')
```

dice: "Compara el valor de `formated_name` con la cadena '`Linus Torvalds`'. Si son iguales como esperamos, bien. Pero si no coinciden, házmelo saber."

Vamos a ejecutar este archivo directamente, pero es importante tener en cuenta que muchos frameworks de pruebas importan los archivos antes de ejecutarlos. Cuando se importa un archivo, el intérprete ejecuta el archivo mientras se está importando. El bloque `if` mira una variable especial, `__name__`, que se establece cuando se ejecuta el programa. Si este archivo se ejecuta como programa principal, el valor de `__name__` se establece en `__main__`. En este caso, ejecutamos el caso de prueba. Cuando un framework de prueba importa este fichero, el valor de `__name__` no será `__main__` y este bloque no será ejecutado.

Cuando ejecutamos `test_name_funcion.py`, obtenemos la siguiente salida:

```
Ran 1 test in 0.002s  
OK
```

El **OK** final nos dice que todas las pruebas unitarias del caso de prueba han pasado.

Esta salida indica que la función `get_formatted_name()` siempre funcionará para nombres que tengan un nombre y un apellido a menos que modifiquemos la función. Cuando modificamos `get_formatted_name()` podemos ejecutar otra vez esta prueba. Si el caso de prueba pasa, sabremos que la función todavía funcionará para nombres como `Linus Torvalds`.

Un test fallido.

¿Cómo es una prueba fallida? Modifiquemos `get_formatted_name()` para que pueda manejar los segundos nombres, pero lo haremos de forma que la función falle para nombres con sólo nombre y apellido como Janis Joplin o Linus Torvalds.

Aquí está la nueva versión de `get_foramtted_name()` que requiere un segundo nombre como argumento:

```
def get_formatted_name(first, middle, last):
    """Genera un nombre completo bien formateado."""
    full_name = f'{first} {middle} {last}'
    return full_name
```

Esta versión puede funcionar para personas con segundos nombres, pero cuando la probamos, vemos que deja de funcionar para personas que sólo tengan nombre y apellido. Esta vez, ejecutando el archivo `test_name_function_failed.py` (creada para esta prueba con fallos), obtenemos esta salida:

```
E
=====
ERROR: test_first_last_name (__main__.NameTestCase)
¿Funcionarán nombres como linus torvalds?
-----
Traceback (most recent call last):
  File "/home/usuario/Escritorio/prueba_sub/test_name_function_failed.py", line 9, in test_first_last_name
    formatted_name = get_formatted_name('linus', 'torvalds')
TypeError: get_formatted_name() missing 1 required positional argument: 'last'
-----
Ran 1 test in 0.000s
```

APUNTE: LA ANTERIOR CAPTURA DE PRUEBAS ESTÁ REALIZADA CON SUBLIME TEXT, AUNQUE TODO LO ESTOY HACIENDO CON PYCHARM.

Hay mucha información aquí, porque hay muchas cosas que puedes necesitar saber cuando falla una prueba. El primer elemento de la salida es una simple `E` que nos dice que una prueba unitaria del caso de prueba ha dado lugar a un error. A continuación, vemos que `test_first_last_name` en `NameTestCase` ha provocado un error. Sabiendo qué prueba ha fallado es fundamental cuando el caso de prueba contiene muchas pruebas unitarias. Luego vemos un rastreo (`traceback`) estándar, que informa de que la llamada a la función `get_formatted_name('linus', 'torvalds')` ya no funciona porque le falta un argumento posicional que es requerido.

También vemos que se ha ejecutado una prueba unitaria. Por último, vemos un mensaje adicional que indica que el caso de prueba global ha fallado y que se ha producido un error al ejecutar el caso de prueba. Esta información aparece al final de la salida, por lo que se de inmediato; no es necesario desplazarse a través de una larga lista de avisos para averiguar cuántas pruebas fallaron.

Respuesta a un test fallido.

¿Qué hacemos cuando falla un test? Asumiendo que estás comprobando las condiciones correcta, el éxito de un test significa que la función se está comportando correctamente y el fallo de un test significa que hay un error en el nuevo código que has escrito. Así que, cuando un test falla, no cambies el test. En su lugar, arregla el código que ha causado que falle el test. Examina los cambios que acabas de hacer a la función y averigua cómo esos cambios provocaron que el comportamiento de la función no fuera el deseado.

En este caso, `get_formatted_name()` sólo requiere el uso de dos parámetros: un nombre y un apellido. Ahora, requiere un nombre, un segundo nombre y un apellido. Añadir el parámetro obligatorio del segundo nombre, rompió el comportamiento deseado de `get_formatted_name()`. La mejor opción aquí es hacer que el segundo nombre sea opcional. Una vez que lo hagamos, nuestras pruebas para nombres como *Linus Torvalds* deberían pasar de nuevo, y además deberíamos ser capaces de aceptar segundos nombres también. Modifiquemos `get_formatted_name()` para que los segundos nombres sean opcionales y luego ejecutemos de nuevo el caso de prueba. Si pasa, nos aseguraremos de que la función maneja correctamente los segundos nombres.

Para hacer que los segundos nombres sean opcionales, moveremos el parámetro `middle` al final de la lista de parámetros en la definición de la función y le daremos un valor vacío por defecto. También añadiremos un test `if` que construya el nombre completo correctamente, dependiendo de si se proporciona o no un segundo nombre:

```
def get_formatted_name(first, last, middle=''):
    """Genera un nombre completo bien formateado."""
    if middle:
        full_name = f'{first} {middle} {last}'
    else:
        full_name = f'{first} {last}'
    return full_name.title()
```

En esta nueva versión de `get_formatted_name()`, el segundo nombre es opcional. Si se pasa un segundo nombre a la función, el nombre completo contendrá nombre, segundo nombre y apellido. En caso contrario, el nombre completo consistirá tan sólo en nombre y apellido. Ahora la función podrá funcionar para ambos tipos de nombres. Para descubrir si la función todavía funciona para nombres como *Linus Torvalds*, ejecutemos `test_name_function.py` de nuevo:

```
.
```

```
Ran 1 test in 0.000s
```

```
OK
```

```
[Finished in 45ms]
```

El caso de prueba ahora funciona. Esto es ideal; significa que la función funcionará para nombres como *Linus Torvalds* de nuevo sin que tengamos que probar la función de forma manual. Arreglar nuestra función ha sido más fácil porque el test fallido nos ha ayudado a identificar el nuevo código que rompía el comportamiento existente.

Añadir nuevos tests.

Ahora que sabemos que `get_formatted_name()` funciona también para nombres sencillo, escribamos un segundo test para personas que incluyan un segundo nombre. Haremos esto añadiendo otro método a la clase `NameTestCase`:

```

import unittest
from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """Pruebas para name_function.py"""

    def test_first_last_name(self):
        """¿Funcionan nombres como Linus Torvalds?"""
        formatted_name = get_formatted_name('linus', 'torvalds')
        self.assertEqual(formatted_name, 'Linus Torvalds')

    def test_first_last_middle_name(self):
        """¿Funcionarán nombre como Wolfgang Amadeus Mozart?"""
        formatted_name = get_formatted_name(
            'wolfgang', 'amadeus', 'mozart')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

if __name__ == '__main__':
    unittest.main()

```

Nombramos este nuevo método `test_first_last_middle_name()`. El nombre del método debe empezar con `test_` así el método arrancará automáticamente cuando ejecutemos `test_name_function.py`. Nombramos el método para dejar claro qué comportamiento de `get_formatted_name()` estamos probando. Como resultado, si el test falla, sabremos inmediatamente qué tipo de nombres están afectados. Está bien tener nombres de métodos largos en nuestras clases de `TestCase`. Tienen que ser descriptivos para que puedas entender la salida cuando tus pruebas fallen, y como Python los llama automáticamente, nunca tendrás que escribir código que llame a estos métodos.

Para probar la función, llamamos a `get_formatted_name()` con un primer nombre, un apellido y un segundo nombre y a continuación usamos `assertEqual()` para comprobar que el nombre completo devuelto coincide con el nombre completo (primer nombre, segundo nombre y apellido) que esperamos. Cuando ejecutemos `test_name_function.py` de nuevo, ambos tests pasarán:

```

..
-----
Ran 2 tests in 0.000s

OK
[Finished in 49ms]

```

¡Genial! Ahora sabemos que la función todavía funciona para nombres como *Linus Torvalds* y podemos estar seguros de que también funcionará para nombres como *Wolfgang Amadeus Mozart*.

EJERCICIOS – TRY IT YOURSELF.

11-1. *City, Country*: Escribe una función que acepte dos parámetros: el nombre de una ciudad y el nombre de un país. La función debe devolver una cadena sencilla de la forma *City, Country* como *Santiago, Chile*. Almacena la función en un módulo llamado *city_functions.py*.

Crea un fichero llamado *test_cities.py* que pruebe la función que acabas de escribir (recuerda que necesitas importar *unittest* y la función que quieras probar). Escribe un método llamado *test_city_country()* para verificar que el resultado de llamar a tu función con valores como '*santiago*' y '*chile*' devuelve la cadena correcta. Ejecuta *test_cities.py* y asegúrate de que *test_city_country* pase la prueba.

11-2. *Population*: modifica tu función para que requiera un tercer parámetro, población. Ahora tiene que devolver una sencilla cadena de la forma *City, Country - population xxx*, como *Santiago, Chile - population 5000000*. Ejecuta *test_cities.py* de nuevo. Asegúrate de que *test_city_country()* falla esta vez.

Modifica la función para que el parámetro *population* sea opcional. Ejecuta *test_cities.py* otra vez y asegúrate de que *test_city_country()* pasa de nuevo.

Escribe un segundo test llamado *test_city_country_population()* que verifique que puedes llamar a tu función con los valores '*santiago*', '*chile*' y '*population=5000000*'. Ejecuta *test_cities.py* de nuevo y asegúrate de que el nuevo test pase.

Probar una clase.

En la primera parte de este capítulo escribimos test para una función sencilla. Ahora escribiremos test para una clase. Usaremos clases en muchos de nuestros propios programas, por lo que es de gran ayuda poder probar que esas clases funcionan correctamente. Si tienes pruebas pasadas (test ok) por una clase en la que estás

trabajando puedes estar seguro de que las mejoras que realices en la clase no romperán accidentalmente su comportamiento actual.

Una variedad de métodos Assert.

Python proporciona un número de métodos `assert` en la clase `unittest.TestCase`. Como se mencionó anteriormente, los métodos `assert` comprueban si una condición que crees que es verdadera en un punto específico del código realmente lo es. Si la condición se cumple según lo esperado, es confirma que el comportamiento de esa parte del programa es el correcto y puedes estar seguro de que no existen errores. Si la condición que asumes que es verdadera en realidad no lo es, Python lanza una excepción.

La tabla 11-1 describe los seis métodos `assert` usados más comúnmente. Con estos métodos puedes verificar que los valores devueltos son iguales o no a los valores especificados, que los valores son verdadero o falso y que los valores están o no en una lista dada. Sólo puedes usar estos métodos en una clase que herede de `unittest.TestCase`, así que vamos a ver cómo podemos utilizar uno de estos métodos en el contexto de la prueba de una clase real.

Table 11-1: Assert Methods Available from the `unittest` Module

Method	Use
<code>assertEqual(a, b)</code>	Verify that <code>a == b</code>
<code>assertNotEqual(a, b)</code>	Verify that <code>a != b</code>
<code>assertTrue(x)</code>	Verify that <code>x</code> is <code>True</code>
<code>assertFalse(x)</code>	Verify that <code>x</code> is <code>False</code>
<code>assertIn(item, list)</code>	Verify that <code>item</code> is in <code>list</code>
<code>assertNotIn(item, list)</code>	Verify that <code>item</code> is not in <code>list</code>

Probar una clase.

Probar una clase es muy parecido a probar una función – gran parte de tu trabajo implica probar el comportamiento de los métodos de la clase. Pero hay algunas diferencias, así que vamos a escribir una clase para probar. Considera una clase que ayuda a administrar encuestas anónimas:

```
class AnonymousSurvey:  
    """Recoge respuestas anónimas a una pregunta de la encuesta."""  
  
    new*  
    def __init__(self, question):  
        """Almacena una pregunta y se prepara para almacenar respuestas."""  
        self.question = question  
        self.responses = []  
  
    new*  
    def show_question(self):  
        """Muestra la pregunta de la encuesta."""  
        print(self.question)  
  
    new*  
    def store_response(self, new_response):  
        """Almacena una única respuesta a la encuesta."""  
        self.responses.append(new_response)  
  
    new*  
    def show_results(self):  
        """Muestra todas las respuestas que se han dado."""  
        print("Survey results: ")  
        for response in self.responses:  
            print(f"- {response}")
```

Esta clase comienza con una pregunta de encuesta que tú proporcionas e incluye una lista vacía para almacenar las respuestas. La clase tiene métodos para imprimir la pregunta de la encuesta, añadir una nueva respuesta a la lista de respuestas e imprimir todas las respuestas almacenadas en la lista. Para crear una instancia de esta clase, todo lo que tienes que proporcionar es una pregunta. Una vez que tienes una instancia representando una encuesta en particular, muestra la pregunta de la encuesta con

`show_question()`, almacena la respuesta utilizando `store_response()` y muestras los resultados con `show_results()`. Para mostrar que la clase `AnonymousSurvey` funciona, vamos a escribir un programa que utilice esa clase:

```
from survey import AnonymousSurvey

# Define una pregunta y haz una encuesta.
question = "¿Cuál fue el primer idioma que aprendiste a hablar?"
my_survey = AnonymousSurvey(question)

# Muestra la pregunta y almacena las respuestas a la pregunta
my_survey.show_question()
print("Introduce 'q' en cualquier momento para terminar.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    my_survey.store_response()

# Muestra los resultados de la encuesta.
print("\nGracias a todos por participar en la encuesta!")
my_survey.show_results()
```

Este programa define una pregunta (“¿Cuál fue el primer idioma que aprendiste a hablar?”) y crea un objeto `AnonymousSurvey` con esa pregunta. El programa llama a `show_question()` para mostrar por pantalla la pregunta y pide al usuario la respuesta. Cada respuesta se almacena a medida que se recibe. Cuando se han introducido todas las respuestas (el usuario introducirá q para salir), `show_results()` imprime los resultados de la encuesta:

```
¿Cuál fue el primer idioma que aprendiste a hablar?  
Introduce 'q' en cualquier momento para terminar.  
  
Language: English  
Language: Spanish  
Language: English  
Language: Mandarin  
Language: q  
  
Gracias a todos por participar en la encuesta!  
Survey results:  
- English  
- Spanish  
- English  
- Mandarin  
  
Process finished with exit code 0
```

Esta clase funciona para una simple encuesta anónima. Pero digamos que queremos mejorar *AnonymousSurvey* y el módulo en el que está, *survey*. Podemos permitir que cada usuario introduzca más de una respuesta. Podríamos escribir un método para listar sólo las respuestas únicas e informar de cuántas veces se ha dado cada respuesta. Podríamos escribir otra clase que gestione encuestas que no sean anónimas.

Implementar tales cambios podría afectar al comportamiento actual de la clase *AnonymousSurvey*. Por ejemplo, es posible que mientras intentas permitir que cada usuario introduzca múltiples respuestas, podemos cambiar accidentalmente cómo se guardan las respuestas únicas. Para asegurarnos de que no rompemos el comportamiento existente mientras desarrollamos este módulo, podemos escribir pruebas para la clase.

Probar la clase *AnonymousSurvey*.

Escribamos un test que verifique un aspecto del comportamiento de *AnonymousSurvey*. Escribamos un test que verifique que una respuesta sencilla a la pregunta de la encuesta se almacenará correctamente. Utilizaremos el método `assertIn()` para verificar

que la respuesta está en la lista de respuestas después de haber sido almacenada:

```
from survey import AnonymousSurvey

new*
class TestAnonymousSurvey(unittest.TestCase):
    """Pruebas para la clase AnonymousSurvey."""

new*
    def test_store_single_response(self):
        """Comprueba que una respuesta sencilla se almacena correctamente."""
        question = "¿Cuál fue el primer idioma que aprendiste a hablar?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('English')
        self.assertIn('English', my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

Empezamos importando el módulo `unittest` y la clase que queremos probar, `AnonymousSurvey`. Llamamos a nuestro caso de prueba `TestAnonymousSurvey`, que de nuevo hereda de `unittest.TestCase`. El primer método comprueba que cuando almacenamos la pregunta a la encuesta, la respuesta termina en la lista de respuestas de la encuesta. Un buen nombre descriptivo para este método es `test_store_single_response()`. Si este test falla, sabremos por el nombre del método que aparece en la salida de la prueba fallida que hubo un problema al almacenar una única respuesta a la encuesta.

Para probar el comportamiento de la clase, necesitamos crear una instancia de la clase. Creamos una instancia llamada `my_survey` con la pregunta “*¿Cuál fue el primer idioma que aprendiste a hablar?*” Almacenamos una respuesta sencilla, `English`, usando el método `store_response()`. A continuación verificamos que la respuesta se ha almacenado correctamente comprobando que `English` está en la lista `my_survey.responses`.

Cuando ejecutamos `test_survey.py`, el test pasa la prueba:

```
.
-----
Ran 1 test in 0.000s
OK
[Finished in 54ms]
```

Esto está bien pero una respuesta sólo es útil si genera más de una respuesta. Comprobemos que se pueden almacenar tres respuesta correctamente. Para hacerlo, añadimos otro método a `TestAnonymousSurvey`:

```
def test_store_three_responses(self):
    """Comprueba si se pueden almacenar tres respuestas individuales correctamente."""
    question = "¿Cuál fue el primer idioma que aprendiste a hablar?"
    my_survey = AnonymousSurvey(question)
    responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        my_survey.store_response(response)

    for response in responses:
        self.assertIn(response, my_survey.responses)
```

Llamamos al nuevo método `test_store_three_responses()`. Creamos un objeto encuesta como hicimos en `test_store_single_response()`. Definimos una lista que contenga tres respuestas diferentes y luego llamamos a `store_response()` para cada una de esas respuestas. Una vez que las respuestas han sido almacenadas, escribimos otro bucle y comprobamos que cada respuesta está ahora en `my_survey.responses`.

Cuando ejecutamos `test_survey.py` de nuevo, ambos test (Para una única respuesta y para tres respuestas) pasa la prueba:

```
..
-----
Ran 2 tests in 0.000s
OK
[Finished in 49ms]
```

Esto funciona perfectamente. Sin embargo, estos tests son un poco repetitivos, así que usaremos otra característica de unittest para hacerlo más eficiente.

El método `setUp()`.

En `test_survey.py` creamos una nueva instancia de `AnonymousSurvey` en cada método de los test y creamos nuevas respuestas en cada método. La clase `unittest.TestCase` tiene un método `setUp()` que permite crear esos objetos una vez y luego usarlos en cada uno de los métodos de prueba. Cuando incluyes un método `setUp()` en una clase `TestCase`, Python ejecuta el método `setUp()` antes de ejecutar cada método que empiece por `test_`. Cualquier objeto creado en el método `setUp()` está también disponible en cada método de prueba que escribas.

Usemos `setUp()` para crear una instancia de encuesta y un conjunto de respuestas que se puedan usar en `test_store_single_response()` y en `test_store_three_responses()`.

```
import unittest
from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Pruebas para la clase AnonymousSurvey."""

    def setUp(self):
        """
        Crea una encuesta y un conjunto de respuestas para usarlas en todos los métodos de prueba.
        """
        question = "¿Cuál fue el primer idioma que aprendiste a hablar?"
        self.my_survey = AnonymousSurvey(question)
        self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):
        """Comprueba que una respuesta sencilla se almacena correctamente."""
        self.my_survey.store_response(self.responses[0])
        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):
        """Comprueba si se pueden almacenar tres respuestas individuales correctamente."""
        for response in self.responses:
            self.my_survey.store_response(response)

        for response in self.responses:
            self.assertIn(response, self.my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

El método `setUp()` hace dos cosas: crea una instancia de encuesta y crea una lista de respuestas. Cada una de ellas lleva el prefijo `self`, así que pueden utilizarse en cualquier parte de la clase.

Esto simplifica los dos métodos de prueba, porque ninguno tiene que crear una instancia de encuesta o una respuesta. El método `test_store_single_response()` verifica que la primera respuesta en `self.responses` - `self.responses[0]` - puede ser almacenada correctamente y `test_store_three_responses()` verifica que todas las tres respuestas en `self.responses` pueden ser almacenadas correctamente.

Cuando ejecutamos `test_survey.py` de nuevo, ambos test pasan la prueba. Estos tests pueden ser particularmente útiles cuando intentamos expandir `AnonymousSurvey` para manejar múltiples respuestas de cada persona. Después de modificar el código para aceptar múltiples respuestas, puedes ejecutar esos tests y asegurarte de que no afecta a la capacidad de almacenar un respuesta sencilla o una serie de respuestas individuales.

Cuando estás testeando tus propias clases, el método `setUp()` puede hacer que tus métodos de prueba sean más fáciles de escribir. Se crea un conjunto de instancias y atributos en `setUp()` y luego utiliza esas instancias en todos tus métodos de prueba. Esto es mucho más fácil que crear un nuevo conjunto de instancias atributos en cada método de prueba.

NOTA: Cuando se está ejecutando un caso de prueba, Python imprime un carácter por cada test unitario a medida que se completa. Un test que pasa imprime un punto, un test que da error imprime una E, y un test que falla en un `assertion` imprime una F. Por eso se verán un número diferente de puntos y caracteres en la primera línea de salida cuando ejecutes tus casos de prueba. Si un test tarda mucho en ejecutarse porque contiene muchas pruebas unitarias, puedes observar esos resultados para hacerte una idea de cuántas pruebas se han superado.

EJERCICIOS – TRY IT YOURSELF.

11-3. Employee:

Escribe una clase llamada `Employee`. El método `__init__()` debe tomar un primer nombre, un apellido, un salario anual y almacenar cada uno de esos atributos. Escribe un método llamado `give_raise()` que añada 5,000 € al salario anual por defecto, pero que también acepte un aumento de una cantidad diferente.

Escribe un caso de prueba para `Employee`. Escribe dos métodos de prueba, `test_give_default_raise()` y `test_give_custom_raise()`. Usa el método `setUp()` para que no tengas que crear una instancia de nuevo empleado en cada método de prueba. Ejecuta tu caso de prueba y asegúrate de que los dos test pasan la prueba.

Resumen.

En este capítulo hemos aprendido a escribir test de prueba para funciones y clases utilizando herramientas del módulo `unittest`. Hemos aprendido a escribir clases que hereden de `unittest.TestCase` y hemos aprendido a escribir métodos de prueba que verifiquen comportamientos específicos que pueden mostrar tus funciones y tus clases. Hemos aprendido a usar el método `setUp()` de forma eficiente para crear instancias y atributos de tus clases que pueden ser usados en todos los métodos de prueba para una clase. Las pruebas son una cuestión importante que muchos principiantes no aprenden. No tienes que escribir pruebas para todos los proyectos sencillos que intentes como principiante. Pero tan pronto como empieces a trabajar en proyecto que impliquen un desarrollo significativo, debes testear los comportamientos de tus funciones y clases. Tendrás más confianza en que los nuevos trabajos en tu proyecto no romperán las partes que funcionan y esto te dará la libertad de realizar mejoras en tu código. Si accidentalmente rompes alguna funcionalidad en tu código, lo sabrás al momento, por lo que podrás arreglar el problema fácilmente. Responder al fallo de un test que has ejecutado es más fácil que responder un fallo reportado por un usuario descontento. Otros programadores respetarán más tus proyectos si incluyes algunos tests iniciales. Se sentirán más a gusto experimentando con tu código y tendrás más disposición a trabajar contigo en tus proyectos. Si quieres contribuir al proyecto en el que están trabajando otros programadores, se espera que demuestres que tu código supera las pruebas existentes y, por lo general, se espera que escribas pruebas para los nuevos comportamientos que introduzcas en el proyecto.

Juega con las pruebas para familiarizarte con el proceso de comprobación de tu código. Escribe pruebas para los comportamientos más críticos de tus funciones y tus clases, pero

no intentes conseguir una cobertura total en los primeros proyectos a menos que tengas una razón específica para hacerlo.

=====

PARTE II.

Proyectos.

¡Enhorabuena! Ahora sabes lo suficiente acerca de Python como para empezar a construir proyectos interesantes y significativos. Crear tus propios proyectos te enseñará nuevas habilidades y asentará tu compresión de los conceptos introducidos en la Parte I.

La Parte II contiene tres tipos de proyectos, y puedes elegir hacer cualquiera de ellos o todos en el orden que quieras. He aquí una breve descripción de cada proyecto para ayudarte a decidir en cual profundizar primero.

Alien Invasion: Hacer un juego con Python.

EN el proyecto Alien Invasion (Capítulos 12, 13 y 14), usarás el paquete Pygame para desarrollar un juego en 2D en el que el objetivo es derribar una flota de alienígenas a medida que caen por la pantalla en niveles que aumentan en velocidad y dificultad. Al final del proyecto habrás aprendido habilidades que te permitirán desarrollar tus propios juegos 2D en Pygame.

Data Visualization.

El proyecto de visualización de datos comienza en el capítulo 15, en el que aprenderás a generar datos y crear una serie de bonitas y funcionales visualizaciones de datos usando Matplotlib y Plotly. El capítulo 16 te enseña a acceder a los datos desde recursos online y a introducirlos en un paquete de visualización para crear gráficos de datos meteorológicos y un mapa de la actividad sísmica mundial. Por último, el capítulo 17 te muestra cómo escribir un programa para automatizar la descarga y la visualización de datos. Aprender a hacer visualizaciones te permite explorar campos de minado de datos, que es una habilidad muy buscada en el mundo de hoy.

Web Applications .

En el proyecto de Aplicaciones Web (Capítulos 18, 19 y 20), usarás el paquete Django para crear una aplicación web sencilla que permita a los usuarios llevar un diario sobre cualquier tema que hayan estado aprendiendo. Los usuarios crearán una cuenta con nombre de usuario y contraseña, introducir un tema y luego hacer entradas sobre lo que están aprendiendo. También aprenderás a desplegar tu aplicación para que cualquier persona del mundo pueda acceder a ella.

Después de completar este proyecto serás capaz de empezar a construir tus propias aplicaciones web simples y estarás listo para profundizar en recursos más completos sobre la construcción de aplicaciones con Django.

PROYECTO 1

ALIEN INVASION

Capítulo 12. Un barco que dispara balas.

Vamos a construir un juego llamado Alien Invasion! Usaremos Pygame, una colección de divertidos y potentes módulos de Python que gestionan gráficos, animación e incluso sonido, facilitándote la creación de juegos sofisticados. Con Pygame manejando tareas como dibujar imágenes en la pantalla, puedes centrarte en la lógica de alto nivel de la dinámica del juego.

En este capítulo instalarás Pygame, y luego crearemos un cohete espacial que se mueva a la derecha y a la izquierda y dispare balas en respuesta a la entrada del usuario. En los dos siguientes capítulos, crearás una flota de aliens para destruir, y luego continuarás refinando el juego configurando límites en el número de naves que se pueden utilizar y añadir un marcador.

Mientras construyes este juego, aprenderás cómo gestionar proyectos más grandes que abarquen varios archivos. Refactorizaremos gran cantidad de código y gestionaremos contenidos para organizar el proyecto y hacer el código más eficiente.

Hacer juegos es una forma ideal para divertirse mientras aprendes un lenguaje. Es profundamente satisfactorio jugar a un juego que has escrito tú y escribir un juego sencillo, te ayudará a comprender desarrollan juegos los profesionales. A medida que trabajes en este capítulo, introduce y ejecuta el código para identificar cómo contribuye cada bloque de código a la jugabilidad general. Experimenta con diferentes valores y configuraciones para comprender mejor cómo perfeccionar las interacciones en tus juegos.

NOTA: Alien Invasion abarca un número de ficheros diferentes, así que haz una nueva carpeta “alien_invasion” en tu equipo. Asegúrate de guardar todos los ficheros para el proyecto en esta carpeta así tus sentencias import funcionarán correctamente.

Además, si te sientes cómodo utilizando el control de versiones, es posible que deseas utilizarlo para este proyecto.

