

3^a EDICIÓN

CURSO INTENSIVO DE PYTHON

INTRODUCCIÓN PRÁCTICA A LA PROGRAMACIÓN
BASADA EN PROYECTOS

ERIC MATTHES

MÁS DE
1 500 000
DE COPIAS VENDIDAS
EN TODO EL MUNDO



ANAYA
MULTIMEDIA

CURSO INTENSIVO DE
PYTHON
3^a EDICIÓN

**Introducción práctica a la
programación basada en proyectos**

Eric Matthes



A mi padre, que siempre sacó tiempo para responder a mis preguntas sobre programación, y a Ever, que está empezando a hacerme las suyas.

Agradecimientos

No habría sido posible escribir este libro sin el magnífico y extremadamente profesional equipo de No Starch Press. Bill Pollock me invitó a escribir un libro introductorio y le agradezco enormemente aquel primer ofrecimiento. Liz Chadwick ha trabajado en las tres ediciones del libro, que es mejor gracias a su implicación en el proyecto. Eva Morrow aportó una visión renovada a esta nueva edición, que se ha visto mejorada gracias a sus aportaciones. Agradezco la ayuda de Dong McNair con el lenguaje, sin que llegue a resultar demasiado formal. Jennifer Kepler ha supervisado el trabajo de producción, que ha transformado mis numerosos archivos en un producto acabado.

Son muchas las personas en No Starch Press que han contribuido al éxito de este libro, pero con las que no he podido trabajar directamente. No Starch cuenta con un fantástico equipo de marketing cuya labor va mucho más allá de vender libros. Se aseguran de que los lectores tengan a su disposición libros que les funcionen y les ayuden a alcanzar sus objetivos. Asimismo, No Starch dispone de un sólido departamento de derechos de autor en el extranjero. Este libro ha llegado a lectores de todo el mundo en numerosos idiomas gracias a la diligencia y buen hacer de este equipo. A todas esas personas con las que no he trabajado personalmente, gracias por ayudar a que este libro encuentre su público.

Me gustaría dar las gracias a Kenneth Love, editor técnico de las tres ediciones. Conocí a Kenneth en PyCon. Desde entonces, su entusiasmo por el lenguaje y la comunidad Python han sido una constante fuente de inspiración profesional. Como siempre, Kenneth ha ido mucho más allá de la simple comprobación de datos y ha revisado el libro con el objetivo de ayudar a los nuevos programadores a desarrollar una comprensión sólida del lenguaje y la programación Python en general. Se ha encargado de aquellas áreas que, aun habiendo funcionado bien en ediciones anteriores,

podían mejorarse y reescribirse por completo. Dicho esto, cualquier error es responsabilidad mía.

Me gustaría asimismo expresar mi gratitud a todos los lectores que han compartido su experiencia trabajando con este libro. Aprender los principios básicos de la programación puede transformar nuestra manera de ver el mundo, y en ocasiones tiene un profundo impacto en la vida de las personas. Escuchar sus historias ha sido una experiencia aleccionadora. Doy las gracias a todos aquellos que han compartido sus vivencias de forma tan generosa.

Quiero agradecer a mi padre el haberme introducido en el mundo de la programación a una edad temprana, sin miedo a que rompiera su equipo. Gracias a mi esposa, Erin, por apoyarme y animarme durante todo el proceso de escritura y de todo el trabajo que implica mantenerlo a lo largo de múltiples ediciones. Gracias también a mi hijo Ever, cuya curiosidad sigue inspirándome.

Sobre el autor

Eric Matthes trabajó como profesor de matemáticas y ciencias en secundaria durante 25 años. Impartió clases de introducción a Python siempre que encontraba la manera de encajar los contenidos en el currículum. Eric trabaja como escritor y programador a tiempo completo y colabora con diversos proyectos de código abierto. Sus proyectos abarcan objetivos muy diversos, desde ayudar a predecir corrimientos de tierra en zonas montañosas a simplificar el proceso de despliegue de proyectos Django. Cuando no está escribiendo o programando, a Eric le encanta escalar y pasar tiempo con su familia.

Sobre el editor técnico

Kenneth Love vive en la costa del Noroeste del Pacífico con su familia y sus gatos. Kenneth es programador Python, colaborador de

código abierto, profesor y conferenciante con una larga trayectoria a sus espaldas.

ÍNDICE DE CONTENIDOS

PREFACIO A LA TERCERA EDICIÓN

INTRODUCCIÓN

PARTE I. LO BÁSICO

1. PRIMEROS PASOS

Configurar un entorno de programación

Versiones de Python

Ejecutar *snippets* de código en Python

Sobre el editor VS Code

Python en distintos sistemas operativos

 Python en Windows

 Python en macOS

 Python en Linux

Ejecutar un programa Hello World

 Instalar la extensión Python para VS Code

 Ejecutar hello_world.py

Solución de problemas

Ejecutar programas de Python desde un terminal

 En Windows

 En macOS y Linux

Resumen

2. VARIABLES Y TIPOS DE DATOS SIMPLES

Lo que pasa en realidad cuando ejecutamos hello_world.py

Variables

 Nombrar y usar variables

 Evitar errores con los nombres al usar variables

 Las variables son etiquetas

Cadenas

 Cambiar mayúsculas y minúsculas en una cadena con métodos

- Uso de variables en cadenas
- Añadir espacios en blanco a cadenas con tabulaciones o nuevas líneas
- Eliminar espacios en blanco
- Eliminar prefijos
- Evitar errores de sintaxis con cadenas
- Números
 - Enteros
 - Flotantes
 - Enteros y flotantes
 - Guiones en números
 - Asignación múltiple
 - Constantes
- Comentarios
 - ¿Cómo se escriben los comentarios?
 - ¿Qué tipo de comentarios debería escribir?
- El Zen de Python
- Resumen

3. INTRODUCCIÓN A LAS LISTAS

- ¿Qué es una lista?
- Acceder a los elementos de una lista
- Las posiciones de índice empiezan en 0, no en 1
- Usar valores individuales de una lista
- Modificar, añadir y eliminar elementos
 - Modificar elementos en una lista
 - Añadir elementos a una lista
 - Eliminar elementos de una lista
- Organizar una lista
 - Ordenar una lista de manera permanente con el método sort()
 - Ordenar una lista temporalmente con la función sorted()
 - Imprimir una lista en orden inverso
 - Descubrir la longitud de una lista
- Evitar errores de índice al trabajar con listas
- Resumen

4. TRABAJO CON LISTAS

- Pasar en bucle por una lista completa
- Los bucles en detalle
- Sacar más partido a un bucle for
- Hacer algo después de un bucle for
- Evitar errores de sangrado

- Olvidar la sangría
- Olvidar sangrar líneas adicionales
- Sangrados innecesarios
- Sangrado innecesario después de un bucle
- Olvidar los dos puntos
- Hacer listas numéricas
 - Utilizar la función range()
 - Usar range() para hacer una lista de números
 - Estadística sencilla con una lista de números
 - Listas por comprensión
- Trabajar con parte de una lista
 - Partir una lista
 - Pasar en bucle por un trozo
 - Copiar una lista
- Tuplas
 - Definir una tupla
 - Pasar en bucle por todos los valores de una tupla
 - Sobrescribir una tupla
- Dar estilo a nuestro código
 - La guía de estilo
 - Sangrado
 - Longitud de línea
 - Líneas en blanco
 - Otras directrices de estilo

Resumen

5. SENTENCIAS IF

- Un ejemplo sencillo
- Pruebas condicionales
 - Comprobar la igualdad
 - Ignorar mayúsculas y minúsculas al comprobar la igualdad
 - Comprobar la desigualdad
 - Comparaciones numéricas
 - Comprobar varias condiciones
 - Comprobar si hay un valor en una lista
 - Comprobar si un valor no está en una lista
 - Expresiones booleanas
- Sentencias if
 - Sentencias if simples
 - Sentencias if-else
 - La cadena if-elif-else

- Utilizar múltiples bloques elif
- Omitir el bloque else
- Probar múltiples condiciones
- Utilizar sentencias if con listas
 - Detectar elementos especiales
 - Comprobar que una lista no está vacía
 - Usar múltiples listas
- Dar estilo a las sentencias if
- Resumen

6. DICCIONARIOS

- Un diccionario sencillo
- Trabajar con diccionarios
 - Acceder a los valores de un diccionario
 - Añadir nuevos pares clave-valor
 - Empezar con un diccionario vacío
 - Modificar valores en un diccionario
 - Eliminar pares clave-valor
 - Un diccionario de objetos similares
 - Usar get() para acceder a valores
- Pasar en bucle por un diccionario
 - Pasar en bucle por todos los pares clave-valor
 - Pasar en bucle por todas las claves del diccionario
 - Pasar en bucle por las claves de un diccionario en un orden particular
 - Pasar en bucle por todos los valores de un diccionario
- Anidación
 - Una lista de diccionarios
 - Una lista en un diccionario
 - Un diccionario en un diccionario
- Resumen

7. ENTRADA DEL USUARIO Y BUCLES WHILE

- Cómo funciona la función input()
- Escribir indicaciones claras
- Usar int() para aceptar entrada numérica
- El operador módulo
- Introducción a los bucles while
 - El bucle while en acción
 - Dejar que el usuario elija cuándo salir
 - Usar una bandera
 - Usar break para salir de un bucle

- Usar continue en un bucle
- Evitar bucles infinitos
- Usar un bucle while con listas y diccionarios
- Pasar elementos de una lista a otra
- Eliminar todos los casos de valores específicos de una lista
- Rellenar un diccionario con entrada del usuario

Resumen

8. FUNCIONES

- Definir una función
- Pasar información a una función
- Argumentos y parámetros
- Pasar argumentos
 - Argumentos posicionales
 - Argumentos de palabra clave
 - Valores predeterminados
 - Llamadas a funciones equivalentes
 - Evitar errores con argumentos
- Valores de retorno
 - Devolver un solo valor
 - Hacer un argumento opcional
 - Devolver un diccionario
 - Usar una función con un bucle while
- Pasar una lista
 - Modificar una lista en una función
 - Evitar que una función modifique una lista
- Pasar un número arbitrario de argumentos
 - Mezclar argumentos posicionales y arbitrarios
 - Usar argumentos de palabra clave arbitrarios
- Guardar las funciones en módulos
 - Importar un módulo completo
 - Importar funciones específicas
 - Usar as para dar un alias a una función
 - Usar as para dar un alias a un módulo
 - Importar todas las funciones de un módulo
- Dar estilo a las funciones
- Resumen

9. CLASES

- Crear y usar una clase
- Creación de la clase Dog

El método `__init__()`

Hacer una instancia de una clase

Trabajar con clases e instancias

La clase Car

Establecer un valor predeterminado para un atributo

Modificar el valor de un atributo

Herencia

El método `__init__()` para una clase derivada

Definir atributos y métodos para la clase derivada

Anular métodos de la clase base

Instancias como atributos

Modelar objetos del mundo real

Importar clases

Importar una sola clase

Almacenar varias clases en un módulo

Importar varias clases desde un módulo

Importar un módulo entero

Importar todas las clases de un módulo

Importar un módulo en otro módulo

Usar alias

Encontrar su propio flujo de trabajo

La biblioteca estándar de Python

Dar estilo a las clases

Resumen

10. ARCHIVOS Y EXCEPCIONES

Leer de un archivo

Leer los contenidos de un archivo

Rutas de archivo relativas y absolutas

Acceder a las líneas de un archivo

Trabajar con el contenido de un archivo

Archivos grandes: un millón de números

¿Está su cumpleaños contenido en pi?

Escribir a un archivo

Escribir una línea

Escribir múltiples líneas

Excepciones

Manejar la excepción `ZeroDivisionError`

Usar bloques `try-except`

Usar excepciones para evitar fallos

El bloque `else`

- Manejar la excepción FileNotFoundError
- Analizar texto
- Trabajar con múltiples archivos
- Fallos silenciosos
- Decidir qué errores informar
- Almacenar datos
 - Utilizar json.dumps() y json.loads()
 - Guardar y leer datos generados por usuarios
- Refactorización
- Resumen

11. PROBAR EL CÓDIGO

- Instalar pytest con pip
- Actualizar pip
- Instalar pytest
- Probar una función
 - Pruebas unitarias y casos de prueba
 - Una prueba que pasa
 - Ejecutar una prueba
 - Una prueba que falla
 - Responder a una prueba fallida
 - Añadir pruebas nuevas
- Probar una clase
 - Varios métodos assert
 - Una clase para probar
 - Probar la clase AnonymousSurvey
 - Configuración de pruebas
- Resumen

PARTE II. PROYECTOS

- Alien Invasion: Hacer un juego con Python
- Visualización de datos
- Aplicaciones web

12. UNA NAVE QUE DISPARA BALAS

- Planificación del proyecto
- Instalar Pygame
- Iniciar el proyecto del juego
 - Crear una ventana de Pygame y responder a entrada de usuario
 - Controlar la tasa de frames

- Configurar el color de fondo
- Crear una clase Settings
- Añadir la imagen de la nave
 - Crear la clase Ship
 - Dibujar la nave en la pantalla
- Refactorización: Los métodos `_check_events()` y `_update_screen()`
 - El método `_check_events()`
 - El método `_update_screen()`
- Pilotar la nave
 - Responder a pulsaciones de teclas
 - Permitir un movimiento continuo
 - Movimiento hacia la izquierda y hacia la derecha
 - Ajustar la velocidad de la nave
 - Limitar el alcance de la nave
 - Refactorización de `_check_events()`
 - Pulsar Q para salir
 - Ejecutar el juego en modo pantalla completa
- Un resumen rápido
 - `alien_invasion.py`
 - `settings.py`
 - `ship.py`
- Disparar balas
 - Añadir la configuración de las balas
 - Crear la clase Bullet
 - Agrupar balas
 - Disparar balas
 - Borrar las balas viejas
 - Limitar el número de balas
 - Crear el método `_update_bullets()`
- Resumen

13. ¡ALIENÍGENAS!

- Revisión del proyecto
- Crear el primer alien
 - Crear la clase Alien
 - Crear una instancia de Alien
- Crear la flota extraterrestre
 - Crear una fila de alienígenas
 - Refactorización de `_create_fleet()`
 - Añadir filas
- Hacer que se mueva la flota

- Mover los aliens hacia la derecha
- Crear configuraciones para la dirección de la flota
- Comprobar si un alien ha llegado al borde
- Descenso de la flota y cambio de dirección
- Disparar a los aliens
 - Detectar colisiones de balas
 - Hacer balas más grandes para pruebas
 - Re poblar la flota
 - Acelerar las balas
 - Refactorización de `_update_bullets()`
- Fin del juego
 - Detectar colisiones entre un alien y la nave
 - Responder a colisiones entre aliens y la nave
 - Aliens que llegan al fondo de la pantalla
 - Game Over
 - Identificar cuándo deberían ejecutarse partes del juego

Resumen

14. PUNTUACIÓN

- Añadir el botón Play
 - Crear una clase Button
 - Dibujar el botón en la pantalla
 - Iniciar el juego
 - Reiniciar el juego
 - Desactivar el botón Play
 - Ocultar el cursor del ratón
- Subir de nivel
 - Modificar las configuraciones de velocidad
 - Restablecer la velocidad
- Puntuaciones
 - Mostrar la puntuación
 - Crear un marcador
 - Actualizar la puntuación a medida que se abaten aliens
 - Restablecer la puntuación
 - Asegurarse de contabilizar todos los aciertos
 - Aumentar los valores en puntos
 - Redondear la puntuación
- Puntuaciones altas
 - Mostrar el nivel
 - Mostrar el número de naves

Resumen

15. GENERAR DATOS

- Instalar Matplotlib
- Trazar un gráfico de líneas sencillo
- Cambiar el tipo de etiqueta y el grosor de la línea
- Corregir el trazado
- Utilizar estilos integrados
- Trazar puntos individuales y darles estilo con scatter()
- Trazar una serie de puntos con scatter()
- Calcular datos automáticamente
- Personalizar las etiquetas de los puntos de los ejes
- Definir colores personalizados
- Utilizar un mapa de color
- Guardar los trazados automáticamente
- Caminatas aleatorias
 - Crear la clase RandomWalk()
 - Elegir direcciones
 - Trazar una caminata aleatoria
 - Generar múltiples caminatas aleatorias
 - Dar estilo a la caminata
- Tirar dados con Plotly
 - Instalar Plotly
 - Crear la clase Die
 - Tirar el dado
 - Analizar los resultados
 - Hacer un histograma
 - Personalizar el gráfico
 - Tirar dos dados
 - Más personalizaciones
 - Tirar dados de distinto tamaño
 - Guardar figuras
- Resumen

16. DESCARGAR DATOS

- El formato de archivo CSV
- Analizar los encabezados de archivo CSV
- Imprimir los encabezados y sus posiciones
- Extraer y leer datos
- Trazar datos en un gráfico de temperatura
- El módulo datetime
- Trazar fechas

- Trazar un periodo de tiempo más largo
- Trazar una segunda serie de datos
- Sombrear un área del gráfico
- Comprobación de errores
- Descargar sus propios datos
- Mapear conjuntos de datos globales: formato JSON
- Descargar datos de terremotos
- Examinar datos JSON
- Hacer una lista con todos los terremotos
- Extraer magnitudes
- Extraer datos de ubicación
- Crear un mapa del mundo
- Representar magnitudes
- Personalizar los colores de los marcadores
- Otras escalas de colores
- Añadir texto emergente

Resumen

17. TRABAJAR CON API

- Usar una API
 - Git y GitHub
 - Solicitar datos usando una llamada a la API
 - Instalar solicitudes
 - Procesar una respuesta de la API
 - Trabajar con el diccionario de respuesta
 - Resumir los principales repositorios
 - Monitorizar los límites de cuota de la API

Visualizar repositorios con Plotly

- Dar estilo al gráfico
- Añadir mensajes emergentes personalizados
- Añadir enlaces activos a nuestro gráfico
- Personalizar los colores de los marcadores
- Más sobre Plotly y la API de GitHub

La API de Hacker News

Resumen

18. PRIMEROS PASOS CON DJANGO

- Configurar un proyecto
- Escribir una especificación
- Crear un entorno virtual
- Activar el entorno virtual

- Instalar Django
- Crear un proyecto en Django
- Crear la base de datos
- Visionar el proyecto
- Iniciar una aplicación
 - Definir modelos
 - Activar modelos
 - El sitio admin de Django
 - Definir el modelo Entry
 - Migrar el modelo Entry
 - Registrar Entry con el sitio Admin
 - El intérprete de Django
- Crear páginas: La página de inicio de Learning Log
 - Asignar una URL
 - Escribir una vista
 - Escribir una plantilla
- Crear páginas adicionales
 - Herencia de plantillas
 - La página topics
 - Página de temas individuales
- Resumen

19. CUENTAS DE USUARIO

- Permitir que los usuarios introduzcan datos
 - Añadir temas nuevos
 - Añadir nuevas entradas
 - Editar entradas
- Configurar cuentas de usuario
 - La aplicación accounts
 - La página de inicio de sesión
 - Cerrar sesión
 - La página de registro
- Permitir que los usuarios controlen sus datos
 - Restringir el acceso con @login_required
 - Conectar datos con determinados usuarios
 - Restringir el acceso a temas a los usuarios adecuados
 - Proteger los temas de un usuario
 - Proteger la página edit_entry
 - Asociar temas nuevos con el usuario actual
- Resumen

20. ESTILO Y DESPLIEGUE DE UNA APP

Dar estilo a Learning Log
La aplicación django-bootstrap5
Usar Bootstrap para dar estilo a Learning Log
Modificar base.html
Dar estilo a la página de inicio con un *jumbotron*
Dar estilo a la página de inicio de sesión
Dar estilo a la página de temas
Dar estilo a las entradas en la página de un tema

Desplegar Learning Log
Crear una cuenta en Platform.sh
Instalar la CLI de Platform.sh
Instalar platformshconfig
Crear un archivo requirements.txt
Requisitos de despliegue adicionales
Añadir archivos de configuración
Modificar settings.py en Platform.sh
Usar Git para hacer un seguimiento de los archivos del proyecto
Crear un proyecto en Platform.sh
Pasar a Platform.sh
Ver el proyecto en vivo
Refinar el despliegue de Platform.sh
Crear páginas de error personalizadas
Desarrollo continuo
Borrar un proyecto en Platform.sh

Resumen

PARTE III. APÉNDICES

APÉNDICE A. INSTALACIÓN Y SOLUCIÓN DE PROBLEMAS

APÉNDICE B. EDITORES DE TEXTO E IDE

APÉNDICE C. CONSEGUIR AYUDA

APÉNDICE D. USAR GIT PARA EL CONTROL DE VERSIONES

APÉNDICE E. SOLUCIONAR PROBLEMAS DE IMPLEMENTACIÓN

CRÉDITOS

PREFACIO A LA TERCERA EDICIÓN

La respuesta a las dos primeras ediciones de este libro fue abrumadoramente positiva, con más de 1.000.000 de copias impresas, incluida la traducción a diez idiomas. He recibido cartas y correos de lectores jubilados, incluso niños de diez años, que quieren aprender a programar en su tiempo libre. El libro también se usa en clases de institutos y universidades. Estudiantes que llevan libros de texto más avanzados utilizan esta guía como referencia para sus clases como un complemento útil. También hay quien lo utiliza para mejorar sus habilidades profesionales, cambiar de trabajo y empezar a trabajar en sus propios proyectos. Dicho de otro modo: los lectores utilizan el libro para todo aquello que yo esperaba y mucho más.

La oportunidad de escribir una tercera edición ha sido una tarea placentera. Aunque Python es un lenguaje maduro, sigue evolucionando, como todos los lenguajes. Mi objetivo principal para esta revisión del libro es mantenerlo actualizado como un buen curso introductorio de Python. Con este libro, el lector aprenderá todo lo que debe saber para empezar a trabajar en sus propios proyectos, así como construir una base sólida para su aprendizaje futuro. He actualizado algunas secciones para reflejar nuevas formas, más sencillas, de trabajar con Python. También he aportado claridad en algunas secciones donde ciertos aspectos del lenguaje no se habían presentado con la exactitud deseable. Todos los proyectos se han actualizado con librerías populares y bien mantenidas que el lector podrá usar con confianza para crear sus propios proyectos.

Aquí tiene un resumen de los cambios introducidos en esta tercera edición:

- El capítulo 1 incluye ahora el editor de texto VS Code, popular entre programadores principiantes y profesionales, que funciona bien en todos los sistemas operativos.
- El capítulo 2 incluye los nuevos métodos `removeprefix()` y `removesuffix()`, útiles a la hora de trabajar con archivos y direcciones URL. Este capítulo incluye también los recientemente mejorados mensajes de error de Python, que ofrecen información mucho más concreta para ayudarle a resolver problemas del código cuando algo no funcione.
- El capítulo 10 utiliza el módulo `pathlib` para trabajar con archivos. Se trata de un enfoque mucho más sencillo a la hora de leer y escribir en archivos.
- El capítulo 11 utiliza `pytest` para escribir pruebas automatizadas para su código. La biblioteca `pytest` se ha convertido en herramienta estándar en la industria para escribir pruebas en Python. Es lo suficientemente amigable para utilizarla en sus primeras pruebas y, si se dedica profesionalmente a la programación en Python, la utilizará en su entorno profesional.
- El proyecto Alien Invasion (capítulos 12-14) incluye un ajuste para controlar la tasa de cuadros por segundo (fps), lo que permite que el juego se ejecute de forma más consistente en diferentes sistemas operativos. Se ha aplicado un enfoque más sencillo para la construcción de la flota de alienígenas, y la organización del proyecto también se ha simplificado.
- En los proyectos de visualización de datos de los capítulos 15-17 se utilizan las características más recientes de Matplotlib y Plotly. Las visualizaciones con Matplotlib emplean ajustes de estilo actualizados. El proyecto de paseo o caminata aleatoria incorpora una pequeña mejora que incrementa la exactitud de los gráficos, lo que significa que verá una mayor variedad de patrones cada vez que genere un nuevo paseo. Todos los proyectos que incluyen Plotly emplean ahora el módulo Plotly Express, que permite generar las visualizaciones iniciales con apenas unas líneas de código. Puede explorar fácilmente

diversas visualizaciones antes de elegir un gráfico concreto, para posteriormente refinar elementos concretos dentro de dicho gráfico.

- El proyecto *Learning Log* (capítulos 18-20) se creó con la versión más reciente de Django y se le aplicó estilo con la última versión Bootstrap. Se han renombrado distintas partes del proyecto para facilitar el seguimiento de su organización. El proyecto se despliega ahora en Platform.sh, un servicio de alojamiento moderno para proyectos Django. El proceso de despliegue está controlado mediante archivos de configuración YAML, que ofrecen un mayor grado de control sobre el despliegue del proyecto. Este enfoque es consistente con la manera en la que los programadores profesionales despliegan los proyectos Django más modernos.
- El apéndice A se ha actualizado por completo para recomendar las prácticas más actuales en la instalación de Python en los principales sistemas operativos. El apéndice B incluye instrucciones detalladas para configurar VS Code, así como breves descripciones de los principales editores de texto e IDE actualmente en uso. El apéndice C remite al lector a nuevos recursos populares para obtener ayuda. El apéndice D ofrece un minicurso intensivo sobre el uso de Git para el control de versiones. El apéndice E es una novedad en esta tercera edición. Incluso con un buen conjunto de instrucciones para desplegar las *apps* que escriba, muchas cosas pueden salir mal. Este apéndice ofrece una guía de resolución de problemas muy completa que podrá utilizar cuando el despliegue no funcione a la primera.
- El índice alfabético también se ha actualizado exhaustivamente para que pueda usar el libro como referencia para sus futuros proyectos con Python.

¡Gracias por leer este libro! Si tiene cualquier comentario o pregunta, no dude en ponerse en contacto conmigo en [@ehmatthes](#) en Twitter.

INTRODUCCIÓN



Todo programador tiene una historia que contar sobre cómo aprendió a escribir su primer programa. Yo empecé a programar de pequeño, cuando mi padre trabajaba para Digital Equipment Corporation, una empresa pionera en la era moderna de la computación. Escribí mi primer programa en un ordenador que mi padre había montado en el sótano. El ordenador era una simple placa base conectada a un teclado, sin carcasa, y el monitor era un tubo de rayos catódicos. Mi primer programa era un sencillo juego para adivinar números, más o menos con este aspecto:

I'm thinking of a number! Try to guess the number I'm thinking of: **25**

Too low! Guess again: **50**

Too high! Guess again: **42**

That's it! Would you like to play again? (yes/no) **no**

Thanks for playing!

Siempre recordaré la satisfacción que sentí viendo a mi familia jugar a un juego que yo había creado y que funcionaba como yo quería.

Aquella primera experiencia me dejó una huella duradera. Es un auténtico placer crear algo con un fin, algo que resuelva un problema. El software que escribo ahora cubre necesidades más importantes que aquellos esfuerzos de mi infancia, pero la satisfacción que me produce crear un programa que funciona es básicamente la misma.

¿Para quién es este libro?

El objetivo de este libro es introducir al lector en Python con la mayor rapidez posible para que pueda empezar a crear programas operativos, incluidos juegos, visualizaciones de datos y aplicaciones web, al tiempo que desarrolla una base de programación que le servirá para el resto de su vida. Este libro está escrito para personas de cualquier edad que nunca han programado con Python o que nunca han programado en general. Este libro es apto para todo aquel que quiera aprender rápidamente lo más básico de la programación y concentrarse en proyectos interesantes, y para los que quieran poner a prueba su comprensión de nuevos conceptos resolviendo problemas significativos. Esta guía también es perfecta para profesores de enseñanza media y superior que desean ofrecer a sus alumnos una introducción a la programación basada en proyectos. Si va a cursar una asignatura universitaria y necesita una introducción a Python más sencilla que su libro de texto, esta obra le facilitará el seguimiento de las clases. Si tiene pensado cambiar de trabajo, este libro le permitirá reorientar su carrera profesional. Ha funcionado para muchos lectores con objetivos muy diversos.

¿Qué puede esperar aprender?

Este libro tiene por objeto convertirle en un buen programador en general y en un buen programador de Python en particular. Al adquirir una buena base en conceptos generales de programación, aprenderá de manera eficiente y adoptará buenos hábitos. Cuando termine con este libro, debería estar listo para pasar a técnicas de Python más avanzadas y tendrá más facilidad para aprender nuevos lenguajes de programación.

En la primera parte del libro, descubrirá los conceptos de programación básicos que necesita conocer para escribir programas con Python. Estos conceptos son los mismos que aprendería al iniciarse en prácticamente cualquier lenguaje de programación. Aprenderá a crear diferentes tipos de datos y a almacenarlos en sus

programas. Aprenderá a construir colecciones de datos, como listas y diccionarios, y a manejar estas colecciones de forma eficaz. Aprenderá a usar bucles `while` y sentencias `if` para comprobar determinadas condiciones, con el fin de poder ejecutar secciones de código específico si se cumplen esas condiciones o ejecutar otras secciones cuando no sea así, una técnica de gran ayuda a la hora de automatizar numerosos procesos.

Aprenderá a aceptar entradas de usuarios para que sus programas sean interactivos y a hacer que se ejecuten mientras el usuario quiera. Explorará cómo escribir funciones que conviertan en reutilizables partes de sus programas; así, solo tendrá que escribir bloques de código que realicen determinadas acciones una vez mientras utiliza el código tantas veces como lo necesite. Más adelante, hará extensivo este concepto a comportamientos más complicados con clases, haciendo que programas bastante sencillos puedan responder a diversas situaciones. Aprenderá a escribir programas para gestionar con gracia errores frecuentes. Después de trabajar con estos conceptos básicos, escribirá unos cuantos programas de complejidad creciente que le permitirán poner en práctica lo aprendido. Por último, dará un primer paso hacia la programación intermedia aprendiendo a escribir pruebas para su código, para poder desarrollar más sus programas sin miedo a generar errores. Toda la información de la primera parte le preparará para emprender proyectos más complejos y ambiciosos.

En la segunda parte, aplicaremos lo aprendido en la primera a tres proyectos. Puede trabajar en todos o solo en algunos, en el orden que crea conveniente. En el primer proyecto (capítulos 12-14), creará un juego de disparar al estilo de Space Invaders, llamado Alien Invasion, que tendrá varios niveles de dificultad. Una vez completado este proyecto, debería estar ya encaminado hacia la creación de sus propios juegos en 2D. Incluso si no aspira a convertirse en programador de juegos, este proyecto es una manera muy satisfactoria de trabajar los contenidos aprendidos en la primera parte.

El segundo proyecto (capítulos 15-17) es una introducción a la visualización de datos. Los científicos de datos utilizan diferentes técnicas de visualización para dar sentido a la enorme cantidad de información de la que disponen. Trabajaremos con conjuntos de datos generados con código, conjuntos de datos descargados de recursos en línea y conjuntos de datos descargados automáticamente por nuestros programas. Una vez completado este proyecto, será capaz de escribir programas que filtren grandes conjuntos de datos y creen representaciones visuales de diferentes tipos de información.

En el tercer proyecto (capítulos 18-20), crearemos una pequeña aplicación web llamada Learning Log. Este proyecto permite mantener un registro organizado de la información aprendida sobre un tema particular. Podrá mantener registros separados para temas diferentes y permitir que otros creen una cuenta para iniciar sus propios registros. También aprenderá a desplegar su proyecto para que cualquiera pueda acceder en línea a él desde cualquier parte.

Recursos en línea

Puede descargarse los recursos del libro (en inglés) en la página web de Anaya Multimedia: <http://www.anayamultimedia.es>, en la opción Selecciona Complemento que encontrará en la ficha correspondiente a este libro. Además, dispone de estos mismos recursos y algunos adicionales en la página web del libro original en https://ehmatthes.github.io/pcc_3e/. Estos recursos (en inglés) incluyen:

- **Instrucciones de instalación:** Las instrucciones de instalación en línea son idénticas a las del libro, pero incluyen enlaces activos para cada uno de los pasos. Si tiene problemas de configuración, utilice este recurso.
- **Actualizaciones:** Como todos los lenguajes de programación, Python está en constante evolución. Tengo una

lista exhaustiva de actualizaciones, así que, si algo no le funciona, compruebe aquí si han cambiado las instrucciones.

- **Soluciones a los ejercicios:** Debería dedicar un buen rato a hacer los ejercicios de las secciones "Pruébelo". Pero, si se atasca y no puede avanzar, encontrará las soluciones de la mayoría de los ejercicios en línea.
- **Chuletas:** También encontrará en línea un conjunto completo de "chuletas" descargables para usar como referencia rápida de los principales conceptos.

Nota: Los ejemplos de código disponibles para descargar están en su forma original; sin embargo, al presentarlos en el libro, los comentarios se han traducido para facilitar la comprensión de la funcionalidad de los distintos fragmentos.

¿Por qué Python?

Todos los años me planteo si debería seguir usando Python o cambiar a un lenguaje distinto, tal vez alguno más nuevo en el mundo de la programación, pero al final me quedo con Python por varias razones. Es un lenguaje increíblemente eficiente: los programas harán más con menos líneas de código que las que requieren otros muchos lenguajes. La sintaxis de Python también ayuda a escribir código "limpio", que, en comparación con otros lenguajes, sea más fácil de leer, depurar y ampliar.

La gente usa Python para muchos fines: hacer juegos, crear aplicaciones web, resolver problemas de negocio y desarrollar herramientas internas en todo tipo de empresas interesantes. Python también se usa mucho en el ámbito científico para la investigación académica y el trabajo aplicado.

Una de las principales razones por las que sigo usando Python es la comunidad que tiene detrás, que incluye a un grupo de personas increíblemente diverso y acogedor. La comunidad es esencial para los programadores porque programar es una actividad solitaria. La mayoría de nosotros, incluso los programadores más

experimentados, necesitamos pedir consejo a otros que ya han resuelto problemas similares. Contar con una comunidad de apoyo bien conectada es crucial para resolver problemas, y la comunidad de Python ayuda mucho a personas que, como usted, están aprendiendo Python como primer lenguaje de programación o que llegan a Python con un bagaje en otros lenguajes.

Python es un lenguaje estupendo para aprender, así que... ¡allá vamos!

PARTE I

LO BÁSICO

La primera parte de este libro presenta los conceptos básicos necesarios para escribir programas en Python. Muchos de estos conceptos son comunes a todos los lenguajes de programación, así que le serán útiles a lo largo de toda su vida de programador.

En el **capítulo 1**, instalará Python en su ordenador y ejecutará un primer programa que muestre el mensaje *Hello world!* en la pantalla.

En el **capítulo 2**, aprenderá a asignar información a variables y a trabajar con valores de texto y numéricos.

Los **capítulos 3 y 4** introducen las listas. Las listas pueden almacenar toda la información que queramos en una ubicación dada, lo que nos permite trabajar con esos datos de una forma eficiente. Podrá trabajar con cientos, miles o incluso millones de valores con apenas unas líneas de código.

En el **capítulo 5**, usaremos sentencias `if` para escribir código que responda de una manera si se cumplen determinadas condiciones y de otra si no se cumplen.

El **capítulo 6** muestra cómo usar los diccionarios de Python, que permiten hacer conexiones entre distintos fragmentos de información. Al igual que las listas, los diccionarios pueden contener tanta información como sea necesario.

En el **capítulo 7**, veremos cómo aceptar entrada de usuarios para que los programas sean interactivos. También aprenderá sobre los bucles `while`, que ejecutan bloques de código repetidas veces mientras se sigan cumpliendo unas condiciones dadas.

En el **capítulo 8**, escribiremos funciones, que son bloques de código que realizan una tarea específica y se pueden ejecutar siempre que lo necesitemos.

El **capítulo 9** introduce las clases, que nos permiten modelar objetos del mundo real. Aprenderá a escribir código que represente perros, gatos, personas, coches, cohetes y mucho más.

El **capítulo 10** muestra cómo trabajar con archivos y gestionar errores para que los programas no fallen inesperadamente. Guardaremos los datos antes de que el programa se cierre y los leeremos de nuevo cuando el programa vuelva a funcionar. También aprenderá sobre las excepciones de Python, que permiten anticipar errores y hacer que los programas los manejen bien.

En el **capítulo 11**, aprenderá a escribir pruebas para su código con el fin de comprobar que sus programas funcionan como deberían. Con ello, podrá ampliar sus programas sin preocuparse por la introducción de errores nuevos. Probar el código es una de las primeras habilidades que le ayudarán a pasar de principiante a programador de nivel medio.

1

PRIMEROS PASOS



En este capítulo, ejecutará su primer programa de Python, `hello_world.py`. Primero, tendrá que comprobar si hay una versión reciente de Python instalada en su ordenador; si no la hay, la instalaremos. También instalaremos un editor de texto para trabajar con los programas de Python. Los editores de texto reconocen el código en Python y destacan secciones a medida que escribimos, lo que facilita la comprensión de la estructura del código.

Configurar un entorno de programación

Python es ligeramente diferente en función del sistema operativo que empleemos, así que habrá que tener algunas consideraciones en cuenta. En las siguientes secciones, nos aseguraremos de que Python esté correctamente instalado en su sistema.

Versiones de Python

Todos los lenguajes de programación evolucionan con la aparición de nuevas ideas y tecnologías y los desarrolladores de Python están continuamente haciendo el lenguaje más versátil y potente. En el momento de escribir estas líneas, la versión más reciente es Python 3.11, pero todos los ejemplos de este libro deberían funcionar con Python 3.9 o posterior. En esta sección, descubriremos si Python está ya instalado en su sistema y si necesita instalar una versión más reciente. El apéndice A también contiene información añadida

relativa a la instalación de la última versión de Python en cada uno de los principales sistemas operativos.

Ejecutar *snippets* de código en Python

Puede ejecutar el intérprete de Python en una ventana de terminal, lo que permite probar trozos de código sin tener que guardar y ejecutar un programa entero.

A lo largo del libro, encontrará *snippets* de código parecidos a esto:

```
>>> print("Hello Python interpreter!")  
Hello Python interpreter!
```

El símbolo `>>>`, al que nos referiremos como el "indicador de Python", indica que debería estar usando la ventana de terminal y el texto en negrita es el código que debería escribir y ejecutar pulsando **Intro**. La mayoría de los ejemplos de código de este libro son pequeños programas autónomos que ejecutaremos desde el editor de texto en lugar de hacerlo desde el terminal, ya que la mayor parte del código se escribe en el editor. En ocasiones, sin embargo, demostraremos ejemplos básicos a través de *snippets* ejecutados en una sesión de terminal de Python para que la demostración sea más efectiva. La presencia de tres díples o corchetes angulares en un listado de código indica que nos encontramos ante un código y salida en una sesión de terminal. Enseguida probaremos a escribir código en el intérprete de su sistema.

Asimismo, usaremos un editor de texto para crear un sencillo programa llamado `Hello World!`, un programa básico a la hora de aprender a programar. Existe una larga tradición en el mundo de la programación según la cual mostrar un mensaje `Hello world!` en la pantalla como tu primer programa con un lenguaje nuevo trae buena suerte. Un ejemplo de programa tan sencillo cumple un

objetivo muy real: si funciona bien en su sistema, cualquier programa que escriba en Python debería funcionar.

Sobre el editor VS Code

VS Code es un editor de texto profesional y muy potente, gratuito y apto para principiantes. VS Code funciona muy bien tanto en proyectos sencillos como complejos. A medida que se sienta cómodo en su aprendizaje de Python, podrá seguir utilizándolo a medida que progrese hacia proyectos de mayor envergadura y complejidad. VS Code puede instalarse en todos los sistemas operativos actuales y ofrece soporte para la mayoría de lenguajes de programación, incluido Python.

El apéndice B ofrece información sobre otros editores de texto. Si tiene curiosidad por conocer otras opciones, puede hojear ese apéndice ahora. Si desea empezar a programar ya, puede usar VS Code para arrancar. Podrá valorar el uso de otros editores a medida que adquiera experiencia como programador. En este capítulo, le mostraré cómo instalar VS Code en su sistema operativo.

Nota: Si ya dispone de un editor de texto y sabe cómo configurarlo para ejecutar programas en Python, puede utilizarlo en lugar de VS Code.

Python en distintos sistemas operativos

Python es un lenguaje de programación multiplataforma, lo que significa que funciona en todos los sistemas operativos principales. Cualquier programa de Python que escriba debería ejecutarse en cualquier ordenador moderno con Python instalado. Sin embargo, los métodos para instalar Python en los distintos sistemas operativos varían ligeramente. En esta sección, veremos cómo instalar Python en su sistema. Primero, comprobaremos si ya existe una versión reciente instalada en su sistema. De no ser así, la instalaremos. A continuación, instalaremos VS Code. Estos son los dos únicos pasos que varían entre sistemas operativos.

En las siguientes secciones, ejecutaremos `hello_world.py` y detectaremos lo que no funcione. Le guiaré en este proceso para cada sistema operativo, para que consiga un entorno de programación con Python en el que pueda confiar.

Python en Windows

Por lo general, Windows no viene con Python, así que es probable que tenga que instalarlo para posteriormente instalar VS Code.

Instalación de Python

Primero, compruebe si Python está instalado en su sistema. En el menú Inicio escriba **command** y haga clic sobre Símbolo de sistema. En la ventana que aparece escriba **python**, en minúsculas. Si obtiene como respuesta un indicador de Python (`>>>`), es que está instalado en su sistema. Si aparece un mensaje de error indicando que `python` no es un comando reconocido, entonces no lo tiene instalado. Si se abre la tienda Microsoft, ciérrela. Es preferible descargar Python desde un instalador oficial a utilizar la versión de Microsoft.

Si Python no está instalado en su sistema o si ve una versión anterior a Python 3.9, tendrá que descargar un instalador de Python para Windows. Vaya a <https://python.org/> y pase el ratón por encima de Downloads y seleccione Windows. Debería ver un enlace para descargar la versión más reciente de Python. Haga clic en él para que se inicie la descarga automática del instalador adecuado para su sistema. Tras descargar el archivo, ejecute el instalador. Asegúrese de seleccionar la opción Add Python.exe to PATH, que hará más fácil configurar bien el sistema. La figura 1.1 muestra esta opción seleccionada.

Ejecutar Python en una sesión de terminal

Abra una ventana de comandos y escriba **python** en minúscula. Debería aparecer un indicador de Python (`>>>`), lo que significa que

Windows ha encontrado la versión de Python que acabamos de instalar.

```
C:\> python
Python 3.x.x (main, Jun . . . , 13:29:14) [MSC v.1932 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Figura 1.1. Asegúrese de seleccionar la casilla de verificación Add Python.exe to PATH.

Nota: Si no ve esta salida o algo similar, consulte las instrucciones de instalación detalladas del apéndice A.

Escriba la siguiente línea en su sesión de Python:

```
>>> print("¡Hola, intérprete de Python!")
¡Hola, intérprete de Python!
>>>
```

Debería ver la salida "`Hola, intérprete de Python!`". Siempre que desee ejecutar un *snippet* de código Python, abra una ventana de comandos e inicie una sesión de terminal. Para terminar la sesión de Python, pulse **Control-D** y, a continuación, **Intro**, o escriba el comando `exit()`.

Instalación de VS Code

Puede descargar el instalar de VS Code en <https://code.visualstudio.com>. Haga clic en Download for Windows y ejecute el instalador. Puede saltarse las secciones sobre macOS y Linux, y seguir los pasos de la sección "Ejecutar un programa Hello World" un poco más adelante

Python en macOS

Python no viene instalado por defecto en la mayoría de los sistemas macOS. Si no lo ha hecho aún, deberá instalarlo en su equipo. En esta sección aprenderá a instalar la última versión de Python y posteriormente instalaremos VS Code, asegurándonos de que está correctamente configurado.

Nota: En versiones anteriores de macOS se incluía Python 2, pero se trata de una versión anticuada que no debe usar.

Comprobar si está instalado Python 3

Abra una ventana de terminal yendo a Aplicaciones>Utilidades>Terminal. También puede pulsar **Comando-Barra espaciadora**, escriba **terminal** y pulse **Intro**. Para comprobar si tiene instalada una versión de Python actualizada, escriba **python3**. Probablemente verá un mensaje sobre la instalación de "herramientas de desarrollo de línea de comandos". Es recomendable instalar estas herramientas una vez instalado Python. Si aparece este mensaje, cierre la ventana emergente.

Si la salida le muestra que la versión instalada es Python 3.9 o posterior, puede seguir leyendo en la siguiente sección ("Ejecutar Python en una sesión de terminal"). Si recibe un mensaje con una versión anterior a Python 3.9, siga las instrucciones que se muestran a continuación para instalar la última versión.

Tenga en cuenta que, cuando trabaje en un sistema macOS, siempre que vea el comando `python` en este libro, deberá reemplazarlo por el comando `python3` para asegurarse de estar utilizando Python 3. En la mayoría de sistemas macOS, el comando `python`, o bien señala una versión anticuada de Python que únicamente debe usarse en las herramientas internas del sistema, o bien no apunta a nada y generará un mensaje de error.

Instalación de la última versión de Python

Vaya a <https://python.org/> y pase el ratón por encima de Downloads y seleccione macOS para descargar la versión más reciente de Python. Debería ver un enlace para descargar la versión más reciente de Python. Haga clic en él para que se inicie la descarga automática del instalador adecuado para su sistema. Una vez descargado el archivo, ejecute el instalador.

Tras ejecutar el instalador, se abrirá una ventana, haga doble clic en el archivo `Install Certificates Command`. Al ejecutar este archivo, podrá instalar más fácilmente las bibliotecas necesarias para sus proyectos, incluidos aquellos de la parte dos de este libro.

Ejecutar Python en una sesión de terminal

Ya puede empezar a ejecutar *snippets* de código Python abriendo un terminal y escribiendo **python3**.

```
$ python3
Python 3.11.4 (v3.11.4:d2340ef257, Jun 6 2023, 19:15:51)
[Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Este comando iniciará una sesión de terminal Python. Verá el indicador de Python (`>>>`), lo que significa que macOS ha encontrado la versión de Python que acabamos de instalar.

Escriba el siguiente código en la sesión de terminal:

```
>>> print("Hello Python interpreter!")  
Hello Python interpreter!  
>>>
```

Deberá ver el mensaje `Hello Python interpreter!`, que aparecerá impreso directamente en la ventana del terminal abierta. Para cerrar el intérprete de Python, pulse **Control-D** o ejecute el comando `exit()`.

Nota: En los sistemas macOS más actuales, verá el símbolo de porcentaje (%) como indicador del terminal en lugar del símbolo del dólar (\$).

Instalación de VS Code

Para instalar el editor VS Code, necesitamos descargar el instalador desde <https://code.visualstudio.com/>. Haga clic en el enlace de descarga, abra una ventana del Finder y vaya a la carpeta `Descargas`. Arrastre el instalador de Visual Studio Code hasta la carpeta `Aplicaciones` y haga clic dos veces sobre el mismo para ejecutarlo.

Puede saltarse la siguiente sección sobre Python para Linux y seguir los pasos de "Ejecutar el programa Hola Mundo".

Python en Linux

Los sistemas Linux están diseñados para programar. Por este motivo, Python ya está instalado en la mayoría de los ordenadores Linux. Quienes escriben y mantienen Linux esperan que en algún momento sus usuarios programen y los animan a hacerlo. Por ello, hay muy poco que instalar y apenas un par de cosas que configurar para empezar a programar.

Comprobar la versión de Python

Abra una ventana de terminal ejecutando la aplicación Terminal en su sistema (en Ubuntu, puede pulsar **Control-Alt-T**). Para averiguar qué versión de Python tiene instalada, escriba **python3**, con la "p" minúscula. Cuando Python está instalado, este comando inicia el intérprete de Python. La salida debería indicarle cuál es la versión instalada y aparecerá `>>>` marcando dónde puede empezar a introducir comandos Python:

```
$ python3
Python 3.10.4 (main, Apr . . . , 09:04:19) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Esta salida indica que Python 3.10.4 es actualmente la versión predeterminada de Python instalada en este ordenador. Cuando la haya visto, pulse **Control-D** o escriba el comando `exit()` para salir del intérprete de Python y volver a un indicador del terminal. Siempre que vea el comando `python` en este libro, utilice `python3` en su lugar.

Necesitará Python 3.9 o posterior para ejecutar el código de este libro. Si la versión instalada en su sistema es anterior, consulte el apéndice A para instalar la más reciente.

Ejecutar Python en una sesión de terminal

Puede empezar a ejecutar *snippets* de código Python abriendo un terminal y escribiendo **python3**, como hemos hecho para comprobar la versión. Hágalo de nuevo y, cuando Python esté en ejecución, escriba la siguiente línea en la sesión de terminal:

```
>>> print("Hello Python interpreter!")
Hello Python interpreter!
>>>
```

El mensaje debería aparecer directamente en la ventana del terminal. Recuerde que puede cerrar el intérprete de Python pulsando **Control-D** o con el comando `exit()`.

Instalación de VS Code

En Linux Ubuntu, podrá instalar VS Code desde el Centro de Software Ubuntu. Haga clic en el icono de Software Ubuntu y busque **vscode**. Haga clic en la app llamada Visual Studio Code (en ocasiones llamada **code**) y haga clic en Instalar. Una vez instalado, busque en su sistema VS Code y ejecute la aplicación.

Ejecutar un programa Hello World

Con una versión reciente de Python y VS Code instalada, ya casi estamos listos para ejecutar nuestro primer programa de Python en un editor de texto. Antes, debemos asegurarnos de que hemos instalado la extensión de Python para VS Code.

Instalar la extensión Python para VS Code

VS Code funciona con diferentes lenguajes de programación. Para sacarle el máximo partido como programador Python, debe instalar la extensión Python, que ofrece soporte para la escritura, edición y ejecución de programas Python.

Para instalar la extensión Python, haga clic en el ícono Manage, que aparece representado como una especie de rueda en la esquina inferior izquierda de la app VS Code. En el menú, haga clic en Extensions. Escriba **python** en el cajetín de búsqueda y haga clic en la extensión Python (si ve más de una extensión llamada Python, elija la de Microsoft). Haga clic en Install e instale las herramientas adicionales que su sistema necesite para completar la instalación. Si ve un mensaje indicando que debe instalar Python una vez instalado, ignore el mensaje.

Nota: Si trabaja en macOS y una ventana emergente le pide instalar *command line developer tools*, haga clic en Install. Quizás vea una advertencia sobre el tiempo de instalación, pero no debería llevarle más de 10 o 20 minutos con una buena conexión de Internet.

Ejecutar hello_world.py

Antes de escribir su primer programa, cree en su sistema una carpeta en su escritorio llamada `python_work` para guardar sus proyectos. En los nombres de archivos y carpetas es recomendable usar minúsculas y guiones bajos en lugar de espacios, ya que Python utiliza estas convenciones de nomenclatura. Puede crear esta carpeta en otra ubicación, pero le resultará más sencillo seguir los pasos posteriores si guarda directamente dicha carpeta en su escritorio.

Abra VS Code y cierre la pestaña Get Started si aún está abierta. Cree un nuevo archivo (`File>New File`) o pulse **Control-N** en macOS. Guarde el archivo como `hello_world.py` en su carpeta de trabajo `python_work`. La extensión `.py` le indica a VS Code que su archivo se está escribiendo en Python y le indica cómo ejecutar el programar y destacar el texto de la forma adecuada.

Una vez guardado el archivo, escriba la siguiente línea en el editor de texto:

`hello_world.py`

`print("Hello Python world!")`

Para ejecutar el programa, seleccione `Run>Run Without Debugging` o pulse **Control-F5**. Debería aparecer una pantalla de terminal en la parte inferior de la ventana de VS Code con la siguiente salida de programa:

`Hello Python world!`

Es probable que aparezca algo más, mostrando el intérprete de Python usado para ejecutar el programa. Si desea simplificar la información mostrada para ver únicamente la salida del programa, consulte el apéndice B. Encontrará sugerencias útiles sobre un uso más eficiente de VS en dicho apéndice.

Si no ve esta salida, es probable que algo haya salido mal. Compruebe todos los caracteres de la línea introducida. ¿Ha escrito `print` con mayúscula sin darse cuenta? ¿Ha olvidado una o ambas comillas o los paréntesis? Los lenguajes de programación esperan una sintaxis muy específica y, si no se la damos, dan error. Si no consigue que el programa funcione, consulte las sugerencias de la siguiente sección.

Solución de problemas

Si no consigue ejecutar `hello_world.py`, puede aplicar algunos de los trucos que se explican a continuación y que, en general, ofrecen buenas soluciones para cualquier problema de programación:

- Cuando un programa contiene un error importante, Python muestra un "rastreo" o un informe de error. Python examina el archivo e intenta identificar el problema. Analice este rastreo: podría darle una pista del problema que está impidiendo que se ejecute el programa.
- Aléjese del ordenador, tómese un descanso y pruebe de nuevo. Recuerde que la sintaxis es muy importante en la programación. Algo tan sencillo como unas comillas mal colocadas o paréntesis mal escritos pueden impedir el buen funcionamiento de un programa. Relea las partes relevantes de este capítulo, analice su código e intente descubrir el error.
- Vuelva a empezar. Es probable que no necesite desinstalar software, pero podría ser conveniente borrar el archivo `hello_world.py` y volver a crearlo desde cero.
- Pida a otra persona que siga los pasos de este capítulo, en el mismo ordenador o en otro, mientras observa atentamente lo

que hace. Puede que se haya saltado algún pequeño paso que la otra persona sí haya hecho.

- Consulte las instrucciones adicionales de instalación del apéndice A. Algunos de los detalles incluidos en dicho apéndice podrían ayudarle a solucionar el problema.
- Busque a alguien que conozca Python y pídale que le ayude a empezar. Seguro que preguntando conoce a algún usuario de Python.
- Las instrucciones de instalación de este capítulo también están disponibles en el sitio web original del libro, https://ehmatthes.github.io/pcc_3e/. La versión en línea de estas instrucciones podría bastarle. Puede copiar y pegar código, y hacer clic en los enlaces a los recursos necesarios.
- Pida ayuda en línea. El apéndice C ofrece una serie de recursos, como foros y salas de chat en vivo, donde puede preguntar por soluciones a personas que ya han pasado por el problema al que se está enfrentando ahora.

No tema molestar a los programadores experimentados. Todo programador se ha atascado en algún momento y la mayoría estarán encantados de ayudar a otros a configurar correctamente sus sistemas. Siempre y cuando sea capaz de explicar con claridad lo que intenta hacer, lo que ha probado ya y los resultados que está obteniendo, es muy probable que alguien pueda ayudarle. Como ya dije en la introducción, la comunidad de Python es muy amable y acoge bien a los principiantes.

Python debería funcionar bien en cualquier ordenador moderno. Encontrar problemas ya con la configuración puede ser frustrante, pero merece la pena solucionarlos. Cuando consiga que `hello_world.py` funcione, puede empezar a aprender Python y su labor de programación se volverá más interesante y satisfactoria.

Ejecutar programas de Python desde un terminal

La mayoría de los programas que escriba en su editor de texto se ejecutarán directamente desde ahí, pero a veces es útil ejecutar programas desde un terminal. Por ejemplo, puede que quiera ejecutar un programa existente sin abrirlo para editar.

Puede hacerlo en cualquier sistema que tenga Python instalado si sabe cómo acceder al directorio en el que está almacenado el archivo. Para probar, asegúrese de guardar el archivo `hello_world.py` en la carpeta `python_work` de su escritorio.

En Windows

Puede usar el comando de terminal `cd`, de "cambiar de directorio", para navegar hasta su sistema de archivos en una ventana de comandos. El comando `dir`, de "directorío", muestra todos los archivos existentes en el directorio actual.

Abra una nueva ventana de terminal y escriba los siguientes comandos para ejecutar `hello_world.py`:

```
C:\> cd Desktop\python_work
C:\Desktop\python_work> dir
hello_world.py
C:\Desktop\python_work> python hello_world.py
Hello Python world!
```

En primer lugar, usamos el comando `cd` para navegar hasta la carpeta `python_work`, que está en la carpeta `Desktop` (el escritorio). A continuación, empleamos el comando `dir` para asegurarnos de que `hello_world.py` está en esta carpeta. Después, ejecutamos el archivo con el comando `python hello_world.py`.

La mayoría de sus programas se ejecutarán sin problemas desde su editor; sin embargo, a medida que su trabajo gane en complejidad, le interesará ejecutar algunos programas desde un terminal.

En macOS y Linux

La ejecución de un programa de Python desde una sesión de terminal es igual en Linux y en macOS. Puede usar el comando de terminal `cd`, de "cambiar de directorio", para navegar hasta su sistema de archivos. El comando `ls`, de "lista", muestra todos los archivos no ocultos que se encuentran en el directorio actual.

Abra una nueva ventana de terminal e introduzca los siguientes comandos para ejecutar `hello_world.py`:

```
~$ cd Desktop/python_work/  
~/Desktop/python_work$ ls  
hello_world.py  
~/Desktop/python_work$ python3 hello_world.py  
Hello Python world!
```

En primer lugar, utilice el comando `cd` para navegar hasta la carpeta `python_work`, que está en la carpeta `Desktop` (el escritorio). A continuación, utilice el comando `ls` para asegurarse de que `hello_world.py` está en esa carpeta. Finalmente, ejecute el archivo con el comando `python3 hello_world.py`.

La mayoría de sus programas se ejecutarán sin problemas desde su editor, pero, a medida que su trabajo gane en complejidad, le interesará ejecutar algunos programas desde un terminal.

PRUÉBELO

Los ejercicios de este capítulo son de carácter exploratorio. A partir del capítulo 2, los retos que resolverá se basarán en lo aprendido.

- **1-1. [python.org:](https://python.org/)** Explore la página de inicio de Python (<https://python.org/>) en busca de temas que le interesen. A medida que se familiarice con Python, encontrará más secciones que le resulten útiles en el sitio web.

- **1-2. Erratas en Hello World:** Abra el archivo `hello_world.py` que acabamos de crear. Introduzca una errata en cualquier parte de la línea y vuelva a ejecutar el programa. ¿Ha producido la errata un error? ¿Le ve sentido al mensaje de error? ¿Puede escribir una errata que no genere un error? ¿Por qué cree que no ha generado un error?
- **1-3. Habilidades infinitas:** Si tuviese habilidades de programación infinitas, ¿qué crearía? Está a punto de aprender a programar. Si tiene un objetivo en mente, tendrá un uso inmediato para sus nuevas habilidades; ahora es un buen momento para redactar un resumen de lo que le gustaría crear. Es buena opción tener una libreta de ideas que pueda consultar cada vez que quiera iniciar un proyecto nuevo. Tómese unos minutos para describir tres programas que le gustaría crear.

Resumen

En este capítulo, ha adquirido algunos conocimientos básicos sobre Python en general y lo ha instalado en su sistema, si es que no lo tenía ya. También ha instalado un editor de texto que facilita la escritura de código Python. Ha ejecutado *snippets* de código Python en una sesión de terminal y ha ejecutado su primer programa, `hello_world.py`. También es probable que haya aprendido algo sobre la solución de problemas.

En el próximo capítulo, aprenderá sobre los distintos tipos de datos con los que puede trabajar en sus programas de Python y también usará variables.

2

VARIABLES Y TIPOS DE DATOS SIMPLES



En este capítulo, conocerá los distintos tipos de datos con los que puede trabajar en sus programas de Python. También aprenderá a usar variables para representar datos en un programa.

Lo que pasa en realidad cuando ejecutamos `hello_world.py`

Echemos un vistazo a lo que hace Python cuando ejecutamos `hello_world.py`. Resulta que Python trabaja bastante, incluso cuando ejecuta un programa sencillo:

`hello_world.py`

```
print("Hello Python world!")
```

Al ejecutar este código, obtendrá el siguiente resultado:

```
Hello Python world!
```

Cuando ejecutamos el archivo `hello_world.py`, la extensión `.py` indica que se trata de un programa de Python. El editor ejecuta entonces el archivo a través del intérprete de Python, que lee el programa y determina lo que significa cada palabra. Por ejemplo, cuando ve la palabra `print` seguida de paréntesis, imprime (es decir, muestra) en la pantalla lo que hay entre los paréntesis. Cuando escribimos programas, el editor resalta distintas partes del programa

de formas diferentes. Por ejemplo, se da cuenta de que `print()` es el nombre de una función y muestra esa palabra de un color. Reconoce que "Hello Python world!" no es código Python y muestra esa frase en un color diferente. Esta característica se conoce con el nombre de "resultado de sintaxis" y le resultará muy útil cuando empiece a escribir sus propios programas.

Variables

Vamos a probar a usar una variable en `hello_world.py`. Añada una línea al principio del archivo y modifique la segunda línea:

`hello_world.py`

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

Ejecute el programa a ver qué pasa. Debería ver la misma salida de antes:

Hello Python world!

Hemos añadido una variable llamada `message`. Todas las variables están conectadas a un valor, que es la información asociada con ellas. En este caso, el valor es el texto "Hello Python world!".

Añadir una variable da un poco más de trabajo al intérprete de Python. Cuando procesa la primera línea, asocia la variable `message` con el texto "Hello Python world!". Cuando llega a la segunda línea, imprime el valor asociado a `message` en la pantalla.

Vamos a expandir este programa modificando `hello_world.py` para que imprima un segundo mensaje. Añada una línea en blanco a `hello_world.py` y luego escriba estas dos líneas de código:

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

```
message = "Hello Python Crash Course world!"  
print(message)
```

Ahora, cuando ejecute `hello_world.py`, debería ver una salida con dos líneas:

```
Hello Python world!  
Hello Python Crash Course world!
```

Puede cambiar el valor de una variable en su programa en cualquier momento y Python siempre seguirá la pista a su valor actual.

Nombrar y usar variables

Cuando utilizamos variables en Python, es necesario seguir una serie de normas y directrices. La ruptura de alguna de esas reglas dará lugar a errores; otras directrices solo nos ayudan a escribir código más fácil de leer y entender. Procure recordar estas reglas al trabajar con variables:

- Los nombres de variable solo pueden contener letras, números y guiones bajos. Pueden empezar por una letra o un guion, pero no por un número. Por ejemplo, podemos llamar a una variable `message_1`, pero no `1_message`.
- Los espacios no están permitidos en los nombres de variable, pero los guiones pueden servir para separar las palabras. Por ejemplo, `greeting_message` funciona, pero `greeting message` dará error.
- Evite usar palabras clave y nombres de función de Python como nombres de variable. Por ejemplo, no use palabras como `print` como nombre de variable, puesto que Python la tiene reservada para un uso de programación concreto (consulte la sección "Palabras clave y funciones integradas de Python").

- Los nombres de variable deben ser cortos pero descriptivos. Por ejemplo, `nombre` es mejor que `n`, `nombre_alumno`, que a su vez es mejor que `n_a` y `nombre_longitud`, que a su vez es mejor que `longitud_de_nombre_de_persona`.
- Tenga cuidado al usar la "l" minúscula y la "O" mayúscula porque se pueden confundir con los números 1 y 0.

Puede requerir un poco de práctica aprender a crear buenos nombres de variable, sobre todo a medida que los programas se vayan complicando. A medida que escriba programas y empiece a leer el código de otras personas, mejorará en la creación de nombres significativos.

Nota: Las variables de Python que usaremos de momento son en minúscula. No obtendrá errores si usa mayúsculas, pero las mayúsculas en nombres de variable tienen significados especiales que veremos en capítulos posteriores.

Evitar errores con los nombres al usar variables

Todo programador comete errores; la mayoría, a diario. Aunque los buenos programadores pueden generar errores, también saben cómo responder de forma eficaz. Vamos a ver un error que probablemente cometerá pronto y aprenderemos a solucionarlo.

Vamos a escribir código que genere un error a propósito. Escriba lo siguiente, incluida la palabra `mesage` mal escrita (en negrita):

```
message = "Hello Python Crash Course reader!"  
print(mesage)
```

Cuando se produce un error en un programa, el intérprete de Python intenta por todos los medios averiguar dónde está el problema y ofrece un rastreo (o *traceback*) cuando un programa no puede ejecutarse correctamente. Este rastreo es un registro que indica dónde encontró problemas el intérprete al intentar ejecutar el código.

Aquí tiene un ejemplo del *traceback* que ofrece Python cuando hemos escrito mal sin querer el nombre de una variable:

```
Traceback (most recent call last):
❶  File "hello_world.py", line 2, in <module>
❷      print(mesage)
           ^
❸ NameError: name 'mesage' is not defined. Did you mean: 'message'?
```

La salida indica que se ha producido un error en la línea 2 del archivo `hello_world.py` ❶. El intérprete muestra la línea ❷ para ayudarnos a detectar el error rápido y nos dice el tipo de error que ha encontrado ❸. En este caso, se trata de un error con un nombre y nos informa de que la variable que se va a imprimir, `mesage`, no se ha definido. Python no puede identificar el nombre de variable suministrado. Cuando hay un error con un nombre puede ser que hayamos olvidado configurar un valor para la variable antes de usarla o que hayamos escrito mal el nombre de la variable. Si Python encuentra un nombre de variable similar al que no reconoce, preguntará si es el nombre que desea utilizar.

En este ejemplo hemos omitido la letra "s" en el nombre de la variable de la segunda línea. El intérprete de Python no comprueba la ortografía de nuestro código, pero sí se asegura de que los nombres de las variables estén bien escritos. Veamos qué ocurre si, por ejemplo, escribimos mal `message` en la línea que define la variable.

```
mesage = "Hello Python Crash Course reader!"
print(mesage)
```

¡En este caso, el programa funciona!

```
Hello Python Crash Course reader!
```

Los nombres de las variables coinciden, de modo que Python no ve problema alguno. Los lenguajes de programación son estrictos,

pero no tienen en cuenta la ortografía. En consecuencia, no hace falta considerar las reglas ortográficas y gramaticales del inglés o el español al probar nombres de variable y escribir código.

Muchos errores de programación son simplemente erratas de un carácter en una línea del programa. Si pasa mucho tiempo buscando este tipo de errores, sepa que no es el único. Muchos programadores experimentados y con talento dedican horas a localizar este tipo de errores minúsculos. Tómeselo con humor y siga adelante, siendo consciente de que es algo que le pasará con frecuencia en su vida de programador.

Las variables son etiquetas

A menudo se describen las variables como cajas en las que se puede almacenar valores. Esta idea puede resultar útil cuando se empieza a usar una variable, pero no describe con precisión cómo se representan las variables internamente en Python. Conviene pensar en las variables como etiquetas a las que podemos asignar valores. También podría decirse que una variable hace referencia a un valor determinado.

Seguramente este matiz sea irrelevante en sus primeros programas, pero merece la pena conocerlo cuanto antes. En algún momento, se topará con un comportamiento inesperado de una variable: saber con exactitud cómo funcionan las variables le ayudará a descubrir qué ha pasado en su código.

Nota: La mejor manera de entender nuevos conceptos de programación es intentar usarlos en sus programas. Si se atasca mientras trabaja con un ejercicio de este libro, pruebe a hacer otra cosa un rato. Si sigue atascado, revise la parte relevante del capítulo. Si aun así necesita ayuda, consulte el apéndice C.

PRUÉBELO

Escriba un programa diferente para cada uno de estos ejercicios. Guarde cada programa con un nombre de archivo que siga las convenciones de Python, con minúsculas y guiones bajos, como `mensaje_simple.py` y `mensajes_simples.py`.

- **2-1. Mensaje simple:** Asigne un mensaje a una variable e imprima ese mensaje.
- **2-2. Mensajes simples:** Asigne un mensaje a una variable e imprímalo. Luego, cambie el valor de la variable por un mensaje diferente e imprima el nuevo mensaje.

Cadenas

Dado que la mayoría de los programas definen y recopilan algún tipo de datos para hacer algo útil con ellos, conviene clasificar distintos tipos de datos. El primero que veremos es la cadena. Las cadenas son bastante simples a primera vista, pero tienen múltiples usos.

Una cadena es una serie de caracteres. Cualquier cosa que se escriba entre comillas se considera una cadena en Python. Para delimitar las cadenas podemos utilizar comillas simples o dobles:

```
"Esto es una cadena."
```

```
'Esto también es una cadena.'
```

Esta flexibilidad nos permite usar comillas y apóstrofos dentro de nuestras cadenas:

```
'Le dije a mi amigo: ";Python es mi lenguaje favorito!"'
```

```
"El lenguaje 'Python' se llama así por Monty Python, no por la serpiente."
```

```
"Una de las fortalezas de Python es su comunidad diversa y solidaria."
```

Vamos a explorar algunas de las formas en las que podemos usar cadenas.

Cambiar mayúsculas y minúsculas en una cadena con métodos

Una de las tareas más sencillas que podemos hacer con cadenas es cambiar las mayúsculas y minúsculas de las palabras que las componen. Observe el siguiente código y trate de determinar qué está pasando:

`name.py`

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

Guarde el archivo como `name.py` y ejecútelo. Debería ver esta salida:

Ada Lovelace

En este ejemplo, la variable `name` se refiere a la cadena en minúsculas `"ada lovelace"`. El método `title()` aparece después de la variable en la llamada a `print()`. Un método es una acción que Python puede realizar con datos. El punto `(.)` detrás de `name` en `name.title()` le dice a Python que haga actuar el método `title()` sobre la variable `name`. Cada método va seguido de un par de paréntesis porque los métodos suelen requerir información adicional para hacer su trabajo. Esa información va entre los paréntesis. La función `title()` no necesita información adicional; por eso, los paréntesis están vacíos.

El método `title()` cambia cada palabra a formato de título, con inicial mayúscula en todas las palabras. Esto es útil porque a menudo nos interesa pensar en un nombre como una información. Por ejemplo, puede que queramos que nuestro programa reconozca los valores de entrada `Ada`, `ADA` y `ada` como el mismo nombre y muestre todos ellos como `Ada`.

Hay otros muchos métodos útiles para tratar con mayúsculas y minúsculas. Por ejemplo, podemos cambiar una cadena a todo mayúsculas o todo minúsculas así:

```
name = "Ada Lovelace"
print(name.upper())
print(name.lower())
```

Esto mostrará lo siguiente:

```
ADA LOVELACE
ada lovelace
```

El método `lower()` es especialmente útil para almacenar datos. Por regla general, no conviene fiarse del uso de las mayúsculas que hagan nuestros usuarios, así que podemos convertir las cadenas a minúsculas antes de guardarlas. Después, a la hora de mostrar la información, podemos usar la grafía que tenga más sentido para cada cadena.

Uso de variables en cadenas

En algunas situaciones, nos interesará usar un valor de variable dentro de una cadena. Por ejemplo, puede que queramos que dos variables representen un nombre y un apellido, respectivamente, y combinar esos valores para mostrar el nombre completo de alguien:

`full_name.py`

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = f"{first_name} {last_name}"
print(full_name)
```

Para insertar el valor de una variable en una cadena, coloque la letra `f` justo antes de abrir las comillas **❶**. Ponga entre llaves el nombre o los nombres de cualquier variable que quiera usar dentro

de la cadena. Python reemplazará cada variable con su valor cuando se muestre la cadena.

Estas cadenas se denominan "cadenas f". La "f" es de "formato" porque Python formatea la cadena reemplazando el nombre de cualquier variable entre llaves con su valor. Esta sería la salida del código anterior:

```
ada lovelace
```

Podemos hacer muchas cosas con las cadenas f. Por ejemplo, podemos usarlas para componer mensajes completos usando la información asociada a una variable, como se muestra aquí:

```
first_name = "ada"  
last_name = "lovelace"  
full_name = f"{first_name} {last_name}"  
❶ print(f"Hello, {full_name.title()}!")
```

Usamos el nombre completo en una frase que salude al usuario **❶**. El método `title()` cambia el nombre en formato de título. Este código devuelve un saludo sencillo, pero con un formato adecuado:

```
Hello, Ada Lovelace!
```

También podemos usar cadenas f para componer un mensaje y posteriormente asignar el mensaje completo a una variable:

```
first_name = "ada"  
last_name = "lovelace"  
full_name = f"{first_name} {last_name}"  
❶ message = f"Hello, {full_name.title()}!"  
❷ print(message)
```

Este código también muestra el mensaje `Hello, Ada Lovelace!`, pero asignando el mensaje a una variable ❶ hacemos la llamada final a `print()` ❷ mucho más sencilla.

Añadir espacios en blanco a cadenas con tabulaciones o nuevas líneas

En programación, un "espacio en blanco" es cualquier carácter que no se imprima, como un espacio, una tabulación o un símbolo de fin de línea. Podemos usar espacios en blanco para organizar la salida de forma que sea más legible para los lectores.

Para añadir una tabulación a un texto, utilice la combinación de caracteres `\t`:

```
>>> print("Python")
Python
>>> print("\tPython")
Python
```

Para añadir una nueva línea en una cadena, use la combinación de caracteres `\n`:

```
>>> print("Languages:\nPython\nC\nJavaScript")
Languages:
Python
C
JavaScript
```

También se puede combinar tabulaciones y nuevas líneas en una misma cadena. La cadena `"\n\t"` le indica a Python que pase a una línea nueva y la empiece con una tabulación. El siguiente ejemplo muestra cómo usar una cadena de una línea para generar cuatro líneas de salida:

```
>>> print("Languages:\n\tPython\n\tC\n\tJavaScript")
```

Languages:

Python

C

JavaScript

Las nuevas líneas y las tabulaciones serán muy útiles en los dos capítulos siguientes, donde empezaremos a producir muchas líneas de salida a partir de unas pocas líneas de código.

Eliminar espacios en blanco

Demasiados espacios en blanco pueden resultar confusos en un programa. Para los programadores, 'python' y 'python ' se parecen mucho; sin embargo, para un programa, son dos cadenas diferentes. Python detecta el espacio adicional en 'python ' y lo considera importante, a menos que se le indique lo contrario.

Es importante pensar en el espacio en blanco porque, con frecuencia, nos interesará comparar dos cadenas para determinar si son iguales. Un ejemplo claro podría implicar comprobar los nombres de usuario de la gente cuando se registran en un sitio web. Un espacio en blanco extra puede resultar confuso también en situaciones mucho más simples. Por suerte, Python hace que sea fácil eliminar espacios en blanco sobrantes de los datos introducidos por el usuario.

Python puede buscar espacios en blanco adicionales a la derecha y a la izquierda de una cadena. Para asegurarnos de que no haya espacios en blanco a la derecha, usaremos el método `rstrip()`:

```
❶  >>> favorite_language = 'python '
❷  >>> favorite_language
      'python '
❸  >>> favorite_language.rstrip()
      'python'
❹  >>> favorite_language
```

```
'python '
```

El valor asociado con `favorite_language` ❶ contiene espacios en blanco adicionales al final de la cadena. Cuando pedimos a Python este valor en una sesión de terminal, vemos el espacio al final del valor ❷. Cuando el método `rstrip()` actúa sobre la variable `favorite_language` ❸, este espacio adicional desaparece. Sin embargo, solo se elimina temporalmente. Si pedimos el valor de `favorite_language` otra vez, veremos que la cadena está exactamente igual que cuando se introdujo, incluyendo el espacio extra ❹.

Para eliminar permanentemente un espacio en blanco de una cadena, debemos asociar el valor modificado con el nombre de la variable:

```
>>> favorite_language = 'python '
❶ >>> favorite_language = favorite_language.rstrip()
>>> favorite_language
'python'
```

Para eliminar el espacio en blanco de la cadena, hay que quitarlo de la derecha de la cadena y asociar el nuevo valor con la variable original ❶. A menudo es necesario cambiar el valor de una variable en programación. Es así como se puede actualizar el valor de una variable cuando se ejecuta un programa o en respuesta a la entrada del usuario.

También se puede eliminar espacios en blanco a la izquierda de la cadena con el método `lstrip()`, o en ambos lados al mismo tiempo con `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> favorite_language.rstrip()
' python'
❸ >>> favorite_language.lstrip()
```

```
'python '
❸ >>> favorite_language.strip()
'python'
```

En este ejemplo, empezamos con un valor que tiene un espacio en blanco al principio y al final ❶. A continuación, eliminamos el espacio adicional de la derecha ❷, de la izquierda ❸ y de ambos lados ❹. Experimentar con estas funciones le ayudará a familiarizarse con la manipulación de cadenas. En el mundo real, estas funciones de eliminación suelen usarse para limpiar la entrada del usuario antes de guardarla en un programa.

Eliminar prefijos

Eliminar prefijos es una tarea muy común a la hora de trabajar con cadenas. Tomemos como ejemplo una URL con el prefijo 'https://'. Es preciso eliminar dicho prefijo para poder centrarnos en la parte de la URL que los usuarios deberán introducir en la barra de dirección. Veamos cómo hacerlo:

```
>>> nostarch_url = 'nostarch.com'
>>> nostarch.url.removeprefix('https:// ')
'nostarch.com'
```

Escriba el nombre de la variable seguida de un punto y a continuación el método `removeprefix()`. Entre paréntesis, escriba el prefijo que desea eliminar de la cadena original.

Al igual que los métodos para eliminar espacios en blanco, `removeprefix()` no realizará ningún cambio sobre la cadena original. Si desea mantener el nuevo valor con el prefijo eliminado, puede reasignarlo a la variable original o asignarlo a una nueva variable.

```
>>> simple_url = nostarch.url.removeprefix('https:// ')
```

Cuando vea una URL en una barra de dirección y la parte 'https://' no se muestre, es probable que el navegador esté utilizando un método como `removeprefix()`.

Evitar errores de sintaxis con cadenas

Un tipo de error que encontrará con cierta frecuencia es el error sintáctico. Estos errores se producen cuando Python no reconoce una sección de un programa como código Python válido. Por ejemplo, si usamos un apóstrofo dentro de comillas simples, obtendremos un error. Esto sucede porque Python interpreta lo que queda entre la primera comilla y el apóstrofo como una cadena y luego intenta interpretar el resto del texto como código Python, lo que produce el error.

Vamos a ver cómo usar bien las comillas simples y dobles. Guarde este programa como `apostrophe.py` y ejecútelo:

`apostrophe.py`

```
message = "One of Python's strengths is its diverse community."  
print(message)
```

El apóstrofo aparece dentro de un par de comillas dobles, así que el intérprete de Python no tiene problemas para leer la cadena correctamente:

One of Python's strengths is its diverse community.

Sin embargo, si usamos comillas simples, Python no puede identificar dónde debería terminar la cadena:

```
message = 'One of Python's strengths is its diverse community.'  
print(message)
```

Aparecerá la siguiente salida:

```
File "apostrophe.py", line 1
    message = 'One of Python's strengths is its diverse community.'
1
SyntaxError: undetermined string literal (detected at line 1)
```

En la salida, vemos que el error se produce justo después de la comilla final simple ❶. Este error sintáctico indica que el intérprete no reconoce algo como código Python válido y que el problema podría ser una cadena con comillas incorrectas. Los errores pueden tener orígenes muy distintos, iré señalando los más habituales cuando surja la oportunidad. Es probable que encuentre errores de sintaxis a menudo mientras aprende a escribir código Python. Estos errores son el tipo de fallo menos específico, así que puede ser difícil y frustrante identificarlos y corregirlos. Si se atasca con un error especialmente obstinado, consulte el apéndice C.

Nota: La función de resultado de sintaxis de su editor le ayudará a detectar con rapidez algunos errores sintácticos al escribir sus programas. Si ve el código Python resaltado como si fuese inglés o español resaltado como si fuese código Python, es posible que en alguna parte de su archivo haya alguna comilla equivocada.

PRUÉBELO

Guarde cada uno de los siguientes ejercicios como archivo independiente con un nombre como `name_cases.py`. Si se atasca, tómese un descanso o consulte las sugerencias del apéndice C.

- **2-3. Mensaje personal:** Use una variable para representar el nombre de una persona e imprima un mensaje para esa persona. El mensaje debería ser sencillo, por ejemplo: "Hola, Eric, ¿te gustaría aprender Python hoy?".
- **2-4. Grafía de nombres:** Use una variable para representar el nombre de una persona e imprima ese nombre en minúsculas, mayúsculas y con mayúscula inicial.

- **2-5. Cita célebre:** Busque una cita de un personaje al que admire. Imprima la cita y el nombre del autor. La salida debería tener un aspecto similar a esto, incluidas las comillas:

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

- **2-6. Cita célebre 2:** Repita el ejercicio 2-5, pero, esta vez, represente el nombre de la persona usando una variable llamada `famous_person`. Despues, componga el mensaje y represéntelo con una nueva variable llamada `message`. Imprima su mensaje.

- **2-7. Eliminar espacios de nombres:** Use una variable para representar el nombre de una persona e incluya algunos caracteres de espacio en blanco al principio y al final del nombre. Asegúrese de usar cada combinación de caracteres, "`\t`" y "`\n`", al menos una vez. Imprima el nombre una vez, de modo que se muestren los espacios de alrededor.

A continuación, imprima el nombre usando cada una de las tres funciones que hemos visto: `lstrip()`, `rstrip()` y `strip()`.

- **2-8. Extensiones de archivos:** Python cuenta con el método `removesuffix()`, que funciona exactamente igual que el método `removeprefix()`. Asigne el valor '`python_notes.txt`' a una variable llamada `filename`. A continuación, utilice el método `removesuffix()` para mostrar el nombre de archivo sin la extensión de archivo, como ocurre con algunos exploradores de archivo.

Números

Los números se usan con bastante frecuencia en programación para llevar las puntuaciones de los juegos, representar datos en visualizaciones, almacenar información en aplicaciones web, etc. Python trata los números de distintas maneras, dependiendo de cómo se usen. Primero, veremos cómo gestiona los enteros, ya que son lo más fácil para trabajar.

Enteros

Podemos sumar (+), restar (-), multiplicar (*) y dividir (/) enteros en Python.

```
>>> 2 + 3  
5  
>>> 3 - 2  
1  
>>> 2 * 3  
6  
>>> 3 / 2  
1.5
```

En una sesión de terminal, Python simplemente devuelve el resultado de la operación. Este lenguaje usa dos símbolos de multiplicación para representar exponentes:

```
>>> 3 ** 2  
9  
>>> 3 ** 3  
27  
>>> 10 ** 6  
1000000
```

Python respeta el orden de las operaciones, así que se pueden usar varias en una expresión. También podemos emplear paréntesis para modificar el orden de las operaciones y que Python evalúe las expresiones en el orden que especifiquemos. Por ejemplo:

```
>>> 2 + 3*4  
14  
>>> (2 + 3) * 4  
20
```

El espaciado de estos ejemplos no afecta a la forma en que Python evalúa las expresiones; simplemente nos ayuda a ver más rápido las operaciones que tienen prioridad al leer el código.

Flotantes

En Python, se considera "flotante" cualquier número con un decimal. Este término se usa en la mayoría de los lenguajes de programación y hace referencia al hecho de que un punto decimal puede aparecer en cualquier posición dentro de un número. Cualquier lenguaje de programación debe diseñarse cuidadosamente para manejar bien los números decimales y que los números se comporten de la forma adecuada, independientemente de dónde aparezca el decimal.

En general, puede usar decimales sin preocuparse por su comportamiento. Simplemente introduzca los números que quiere usar y Python hará lo que cabría esperar:

```
>>> 0.1 + 0.1  
0.2  
>>> 0.2 + 0.2  
0.4  
>>> 2 * 0.1  
0.2  
>>> 2 * 0.2  
0.4
```

No obstante, tenga en cuenta que, a veces, puede obtener un número arbitrario de posiciones decimales en la respuesta:

```
>>> 0.2 + 0.1  
0.3000000000000004  
>>> 3 * 0.1  
0.3000000000000004
```

Es algo que pasa en todos los lenguajes y no es para preocuparse. Python intenta buscar la forma de representar el resultado con la mayor precisión posible, lo que a veces es difícil viendo cómo los ordenadores representan los números internamente. De momento, ignore los decimales extra; ya aprenderá a lidiar con ellos cuando lo necesite en los proyectos de la segunda parte del libro.

Enteros y flotantes

Cuando dividimos dos números, aunque sean enteros que dan un resultado exacto, siempre obtenemos un flotante:

```
>>> 4/2  
2.0
```

Si mezclamos un entero y un flotante en una operación, también obtendremos un flotante:

```
>>> 1 + 2.0  
3.0  
>>> 2 * 3.0  
6.0  
>>> 3.0 ** 2  
9.0
```

Python va por defecto a un flotante siempre que una operación use un flotante, incluso aunque la salida sea un resultado exacto.

Guiones en números

Cuando escribimos números largos, podemos agrupar los dígitos usando guiones bajos para hacer los números más legibles:

```
>>> universe_age = 14_000_000_000
```

Al imprimir un número definido con guiones, Python imprime solo los dígitos:

```
>>> print(universe_age)  
14000000000
```

Python ignora los guiones cuando almacena este tipo de valores. Incluso aunque no agrupemos los dígitos de tres en tres, el valor no se verá afectado. Para Python, `1000` es lo mismo que `1_000`, que, a su vez, es igual que `10_00`. Esta característica funciona tanto para enteros y para decimales.

Asignación múltiple

Podemos asignar valores a más de una variable usando solo una línea. Esto ayuda a acortar los programas y hacerlos más fáciles de leer; usará esta técnica con mayor frecuencia cuando inicialice un conjunto de números.

Por ejemplo, así inicializaríamos las variables `x`, `y` y `z` a cero:

```
>>> x, y, z = 0, 0, 0
```

Hay que separar los nombres de las variables con comas y hacer lo mismo con los valores para que Python asigne cada valor a la variable respectiva. Siempre y cuando el número de valores coincida con el número de variables, Python los relacionará correctamente.

Constantes

Una constante es como una variable cuyo valor permanece invariable a lo largo del programa. Python no tiene tipos de constantes integrados, pero los programadores de Python utilizan mayúsculas para indicar que una variable debería tratarse como una constante y no alterarse nunca:

```
MAX_CONNECTIONS = 5000
```

Cuando quiera tratar una variable como una constante en su código, escriba todo el nombre en mayúsculas.

PRUÉBELO

- **2-9. Número ocho:** Escriba operaciones de suma, resta, multiplicación y división que den como resultado el número 8. Asegúrese de incluir sus operaciones en llamadas a `print()` para ver los resultados. Debería crear cuatro líneas con este aspecto:

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

La salida debería ser simplemente cuatro líneas con el número 8 una vez en cada una.

- **2-10. Número favorito:** Use una variable para representar su número favorito. A continuación, usando esa variable, cree el mensaje que revele su número favorito e imprímalo.

Comentarios

Los comentarios son una característica extremadamente útil en la mayoría de los lenguajes de programación. Todo lo que ha escrito en sus programas hasta ahora es código Python, pero, a medida que sus programas crezcan en longitud y complejidad, deberá ir añadiendo notas que describan su enfoque general para el problema que está resolviendo. Un comentario permite escribir notas en el idioma del usuario dentro de un programa.

¿Cómo se escriben los comentarios?

En Python, la almohadilla (`#`) indica un comentario. El intérprete de Python ignorará cualquier cosa que vaya detrás de una

almohadilla. Por ejemplo:

comment.py

```
# Saludar a todos.  
print("Hello Python people!")
```

Python ignora la primera línea y ejecuta la segunda.

```
Hello Python people!
```

¿Qué tipo de comentarios debería escribir?

La principal razón para escribir comentarios es explicar qué se supone que hace el código y cómo hacemos que funcione. Mientras trabajamos en un proyecto, entendemos cómo encajan todas las piezas; sin embargo, cuando lo abandonamos un tiempo y volvemos a él, es posible que hayamos olvidado los detalles. Siempre podemos estudiar el código y tratar de adivinar cómo se supone que actúa cada segmento, pero escribir buenos comentarios nos ahorra tiempo resumiendo el enfoque general en un lenguaje inteligible. Si quiere convertirse en programador profesional o colaborar con otros programadores, tendrá que escribir buenos comentarios. Actualmente, la mayoría del código se escribe de forma colaborativa, ya sea por un grupo de empleados de una empresa o por un grupo de personas que trabajan juntas en un proyecto de código abierto. Los programadores cualificados esperan encontrar comentarios en el código, así que lo mejor será que empiece a añadir comentarios descriptivos a sus programas. Escribir comentarios claros y concisos en el código es uno de los hábitos más beneficiosos que puede adoptar como nuevo programador.

A la hora de determinar si escribir o no un comentario, pregúntese si ha tenido que considerar varias opciones antes de dar con una forma razonable de hacer que algo funcione; si es el caso, deje un comentario sobre la solución. Es mucho más fácil borrar

después comentarios adicionales que retroceder y escribir comentarios para un programa poco comentado. A partir de ahora, usaré comentarios en los ejemplos del libro para explicar algunas secciones de código.

PRUÉBELO

- **2-11. Añadir comentarios:** Elija dos de los programas que ha escrito y escriba al menos un comentario en cada uno de ellos. Si no tiene nada concreto que decir porque los programas son todavía demasiado simples, añada su nombre y la fecha al principio de cada archivo de programa. Luego escriba una frase describiendo qué hace el programa.

El Zen de Python

Los programadores de Python experimentados le animarán a evitar complejidades y buscar la simplicidad siempre que sea posible. La filosofía de la comunidad Python está recogida en *The Zen of Python*, de Tim Peters. Puede acceder a este breve compendio (en inglés) de principios para escribir buen código Python escribiendo `import this` en su intérprete. No voy a reproducir aquí todo el "Zen de Python", pero compartiré algunas líneas para ayudarle a entender por qué estos principios son importantes para usted como programador de Python en ciernes.

```
>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
```

"Bonito mejor que feo". Los programadores de Python creen que el código puede ser bonito y elegante. Al programar, resolvemos problemas. Los programadores siempre han respetado las soluciones bien diseñadas, eficientes e incluso bonitas. A medida que aprenda

más sobre Python y lo use para escribir más código, alguien que observe su trabajo podrá decir: "¡Vaya, qué código más bonito!".

Simple is better than complex.

"Simple mejor que complejo". Si puede elegir entre una solución simple y una compleja y las dos funcionan, elija la más sencilla. Su código será más fácil de mantener y tanto usted como otros podrán usarlo como base con facilidad.

Complex is better than complicated.

"Complejo mejor que complicado". La vida real es caótica y a veces es imposible encontrar una solución simple para un problema. En ese caso, use la solución menos complicada que funcione.

Readability counts.

"La legibilidad importa". Incluso cuando el código sea complejo, intente que sea legible. Cuando trabaje en un proyecto de codificación compleja, concéntrese en escribir comentarios informativos para ese código.

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

"Debería haber una, y preferiblemente solo una, forma obvia de hacerlo". Si se pide a dos programadores de Python que resuelvan el mismo problema, deberían dar con soluciones relativamente compatibles entre sí. Eso no significa que no haya espacio para la creatividad en la programación. ¡Al contrario, siempre lo hay! Sin embargo, buena parte de la programación consiste en usar pequeños enfoques comunes a situaciones simples dentro de un proyecto más grande y creativo. Los entresijos de sus programas deberían tener sentido para otros programadores de Python.

Now is better than never.

"Ahora es mejor que nunca". Podría pasar el resto de su vida aprendiendo todas las complejidades de Python y la programación en general, pero entonces nunca completaría ningún proyecto. No intente escribir código perfecto; escriba código que funcione y luego decida si mejorarlo para ese proyecto o pasar a algo nuevo.

Cuando avance al siguiente capítulo y empiece a sumergirse en temas más complejos, recuerde esta filosofía de sencillez y claridad. Los programadores experimentados respetarán más su código y estarán encantados de darle su opinión y colaborar con usted en proyectos interesantes.

PRUÉBELO

- **2-12. Zen de Python:** Escriba `import this` en una sesión de terminal de Python y lea todos los principios.

Resumen

En este capítulo hemos visto cómo trabajar con variables. Ha aprendido a usar nombres de variable descriptivos y a resolver errores con los nombres y la sintaxis cuando se producen. Ha descubierto qué son las cadenas y cómo mostrar cadenas en minúsculas, mayúsculas y con iniciales mayúsculas. Hemos empezado a usar espacios en blanco para organizar una salida limpia y ya sabemos cómo eliminar elementos innecesarios de distintas partes de una cadena. Hemos empezado a trabajar con enteros y flotantes y ha aprendido algunas formas de trabajar con datos numéricos. También ha aprendido a escribir comentarios explicativos para que su código sea más fácil de entender. Por último, hemos repasado la filosofía de mantener el código lo más simple posible, siempre que se pueda.

En el capítulo 3, veremos cómo guardar colecciones de información en estructuras de datos llamadas "listas". Aprenderá a trabajar con ellas, manipulando la información que contienen.

3

INTRODUCCIÓN A LAS LISTAS



programadores, y

En este capítulo y el siguiente, descubrirá qué son las listas y cómo empezar a trabajar con los elementos que contienen. Las listas permiten almacenar conjuntos de información en una ubicación, independientemente de si se trata de unos pocos elementos o de millones. Las listas son una de las características más potentes de Python, fácilmente accesibles para los nuevos programadores, y aúnan muchos conceptos importantes de programación.

¿Qué es una lista?

Una lista es una colección de elementos dispuestos en un orden particular. Podemos crear una lista que incluya las letras del abecedario, los números del 0 al 9, o los nombres de todos nuestros familiares. Podemos incluir todo lo que queramos en una lista y esos elementos no tienen por qué estar relacionados de una forma concreta. Dado que una lista suele contener más de un elemento, conviene ponerle un nombre en plural, como `leturas`, `números` o `nombres`.

En Python, el uso de corchetes `[]` indica una lista. Dentro de ella, los elementos individuales se separan mediante comas. Aquí tiene un sencillo ejemplo de lista con diversos tipos de bicicleta:

`bicycles.py`

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

```
print(bicycles)
```

Si pedimos a Python que imprima una lista, nos devolverá su representación de la lista, incluidos los corchetes:

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

Dado que no es esa la salida que queremos que vean nuestros usuarios, vamos a aprender a acceder a los elementos individuales de una lista.

Acceder a los elementos de una lista

Las listas son colecciones ordenadas, así que podemos acceder a cualquiera de sus elementos indicando a Python la posición, o el índice, del elemento deseado. Para acceder a un elemento de una lista, escriba el nombre de la lista seguido del índice del elemento entre corchetes.

Por ejemplo, vamos a sacar la primera bicicleta de la lista `bicycles`:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

Cuando pedimos un elemento individual de una lista, Python devuelve justo ese elemento, sin corchetes:

```
trek
```

Este es el resultado que queremos que vean los usuarios: una salida limpia y con un formato cuidado.

También podemos usar los métodos de cadena del capítulo 2 con cualquier elemento de esta lista. Por ejemplo, podemos dar un formato más cuidado al elemento `'trek'` empleando el método `title()`:

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

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

Este ejemplo produce la misma salida que el anterior, pero en esta ocasión 'Trek' aparece en mayúscula.

Las posiciones de índice empiezan en 0, no en 1

Python considera que el primer elemento de una lista está en la posición 0, no en la 1. Esto ocurre en la mayoría de los lenguajes de programación, y la razón tiene que ver con cómo se implementan las operaciones con listas en un nivel inferior. Si recibe resultados inesperados, pregúntese si no estará cometiendo un simple (pero frecuente) error por uno.

El segundo elemento de una lista tiene el índice 1. Usando este sistema para contar, puede obtener cualquier elemento que deseé restando uno a su posición en la lista. Por ejemplo, para acceder al cuarto elemento, tendrá que solicitar el índice 3.

Lo siguiente pide las bicicletas de los índices $_1$ y $_3$:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[1])
print(bicycles[3])
```

Este código devuelve la segunda y la cuarta bicicleta de la lista:

```
cannondale
specialized
```

Python tiene una sintaxis especial para acceder al último elemento de una lista. Si pedimos el elemento con índice -1 , Python siempre devolverá el último de la lista:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[-1])
```

Este código devuelve el valor `'specialized'`. Esta sintaxis es bastante útil porque a menudo nos interesa acceder a los últimos elementos de una lista sin conocer exactamente la longitud de esta. Esta convención se extiende a otros valores de índice negativos. El índice `-2` devuelve el penúltimo elemento de la lista, `-3` el tercero empezando por el final, y así sucesivamente.

Usar valores individuales de una lista

Podemos usar valores individuales de una lista igual que haríamos con cualquier otra variable. Por ejemplo, podemos usar cadenas `f` para crear un mensaje basado en un valor de una lista.

Vamos a probar a sacar la primera bici de la lista y a componer un mensaje usando ese valor.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
message = f"My first bicycle was a {bicycles[0].title()}."
print(message)
```

Creamos una oración con el valor de `bicycles[0]` y le asignamos la variable `message`. La salida es una sencilla oración sobre la primera bicicleta de la lista:

```
My first bicycle was a Trek.
```

PRUÉBELO

Pruebe estos programas cortos para experimentar de primera mano con listas Python. Quizás le interese crear una carpeta nueva para los ejercicios de cada capítulo. Así, los tendrá organizados.

- **3-1. Nombres:** Guarde los nombres de unos cuantos amigos en una lista llamada `nombres`. Imprima el nombre de cada persona accediendo al elemento correspondiente de la lista, de uno en uno.

- **3-2. Saludos:** Empiece con la lista del ejercicio 3-1, pero, en vez de solo imprimir el nombre de cada persona, imprímales un mensaje. El texto debería ser el mismo, pero cada mensaje debería personalizarse e incluir el nombre de la persona.
- **3-3. Su propia lista:** Piense en su medio de transporte favorito, como la moto o el coche, y haga una lista que incluya varios ejemplos. Use la lista para imprimir una serie de frases sobre esos elementos, como "Me gustaría tener una moto Honda".

Modificar, añadir y eliminar elementos

La mayoría de las listas que cree serán dinámicas, lo que significa que creará la lista para posteriormente añadir y eliminar elementos a medida que el programa siga su curso. Por ejemplo, podría crear un juego en el que un jugador tenga que disparar alienígenas en el espacio. Podría guardar el conjunto inicial de alienígenas en una lista y eliminar uno cada vez que le disparen. La lista de extraterrestres aumentará y disminuirá en el transcurso del juego.

Modificar elementos en una lista

La sintaxis para modificar un elemento es similar a la usada para acceder a un elemento en una lista. Para cambiar un elemento, use el nombre de la lista seguido del índice del elemento que desea modificar y proporcione el nuevo valor que quiere darle.

Por ejemplo, supongamos que tenemos una lista de motos cuyo primer elemento es '`honda`'. ¿Cómo cambiamos el valor de este primer elemento una vez creada la lista?

`motorcycles.py`

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
```

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

En este ejemplo definimos la lista `motorcycles`, con `'honda'` como primer elemento.

A continuación, cambiamos el valor del primer elemento por `'ducati'`. La salida muestra que el primer elemento ha cambiado realmente, mientras que el resto de la lista permanece igual:

```
['honda', 'yamaha', 'suzuki']  
['ducati', 'yamaha', 'suzuki']
```

Podemos cambiar el valor de cualquier elemento de una lista, no solo del primero.

Añadir elementos a una lista

Puede que le interese añadir un elemento nuevo a una lista por varios motivos. Por ejemplo, a lo mejor quiere hacer que aparezcan nuevos alienígenas en un juego, nuevos datos en una visualización o nuevos usuarios registrados en un sitio web que ha creado. Python ofrece varias formas de añadir datos nuevos a listas existentes.

Adjuntar elementos al final de una lista

La forma más fácil de añadir un elemento nuevo es adjuntarlo a la lista. Al hacerlo, el nuevo elemento se añade al final. Usando la misma lista del ejemplo anterior, vamos a añadir el elemento `'ducati'` al final:

```
motorcycles = ['honda', 'yamaha', 'suzuki']  
print(motorcycles)  
  
motorcycles.append('ducati')  
print(motorcycles)
```

El método `append()` añade '`'ducati'`' al final de la lista sin afectar a ninguno de los otros elementos:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha', 'suzuki', 'ducati']
```

El método `append()` facilita la creación dinámica de listas. Por ejemplo, podemos empezar con una lista vacía y añadir elementos después con una serie de llamadas a `append()`. Usando una lista en blanco, vamos a añadirle los elementos '`'honda'`', '`'yamaha'`' y '`'suzuki'`':

```
motorcycles = []

motorcycles.append('honda')
motorcycles.append('yamaha')
motorcycles.append('suzuki')

print(motorcycles)
```

La lista resultante es exactamente igual que las listas de los ejemplos anteriores:

```
['honda', 'yamaha', 'suzuki']
```

Es muy habitual crear listas así porque, a menudo, no sabemos qué datos van a guardar nuestros usuarios en un programa hasta después de ejecutarlo. Para dar el control a los usuarios, empiece por definir una lista vacía que aloje los valores de los usuarios. A continuación, adjunte cada nuevo valor suministrado a la lista que acaba de crear.

Insertar elementos en una lista

Podemos añadir un elemento nuevo a una lista en cualquier posición con el método `insert()`. Para ello, especificaremos el índice del nuevo elemento y su valor.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')
print(motorcycles)
```

En este ejemplo, insertamos el valor 'ducati' al principio de la lista. El método `insert()` abre un espacio en la posición 0 y guarda ahí el valor 'ducati'.

```
['ducati', 'honda', 'yamaha', 'suzuki']
```

Esta operación desplaza todos los demás valores de la lista una posición a la derecha.

Eliminar elementos de una lista

Con frecuencia, necesitará eliminar un elemento o varios de una lista. Por ejemplo, cuando un jugador dispare a un extraterrestre, tendrá que quitarlo de la lista de alienígenas activos. O, si un usuario decide cancelar su cuenta en una aplicación web, tendrá que quitarlo de la lista de usuarios activos. Puede eliminar un elemento según su posición en la lista o según su valor.

Eliminar un elemento con la sentencia `del`

Si conoce la posición del elemento que desea eliminar de la lista, puede usar la sentencia `del`:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

print(motorcycles)

del motorcycles[0]
print(motorcycles)
```

En este ejemplo utilizamos la sentencia `del` para eliminar el primer elemento, 'honda', de la lista de motos:

```
['honda', 'yamaha', 'suzuki']
['yamaha', 'suzuki']
```

Podemos eliminar un elemento en cualquier posición de una lista con la sentencia `del` si sabemos su índice. Por ejemplo, para quitar el segundo elemento de la lista, `'yamaha'`, haríamos esto:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)

del motorcycles[1]
print(motorcycles)
```

Se ha borrado la segunda motocicleta:

```
['honda', 'yamaha', 'suzuki']
['honda', 'suzuki']
```

En ambos ejemplos, ya no se puede acceder al valor que se ha eliminado de la lista después de usar la sentencia `del`.

Eliminar un elemento con el método `pop()`

En ocasiones, necesitaremos usar el valor de un elemento tras haberlo eliminado de una lista. Por ejemplo, podríamos necesitar la posición `x` e `y` de un alienígena al que acaban de derribar para poder dibujar una explosión en esa posición. En una aplicación web podríamos necesitar eliminar a un usuario de una lista de miembros activos para añadirlo a una de miembros inactivos.

El método `pop()` elimina el último elemento de una lista, pero permite trabajar con él después de eliminarlo. El término "pop" viene del inglés y hace referencia a una pila de elementos de la que se saca el que está más arriba. En esta analogía, la parte superior de la pila corresponde al final de una lista.

Vamos a sacar una moto de nuestra lista:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki']
   print(motorcycles)

❷ popped_motorcycle = motorcycles.pop()
❸ print(motorcycles)
❹ print(popped_motorcycle)
```

Empezamos definiendo e imprimiendo la lista `motorcycles` ❶. A continuación, sacamos un valor de la lista y lo guardamos en la variable `popped_motorcycle` ❷. Imprimimos la lista ❸ para ver si el valor se ha eliminado de la lista. Luego imprimimos el valor retirado ❹ para comprobar que todavía tenemos acceso a él.

La salida muestra que el valor `'suzuki'` ha desaparecido del final de la lista y se ha asignado a la variable `popped_motorcycle`:

```
['honda', 'yamaha', 'suzuki']
['honda', 'yamaha']
suzuki
```

¿Qué utilidad puede tener este método `pop()`? Imagine que las motos de la lista están guardadas en orden cronológico por fecha de adquisición. En ese caso, podríamos utilizar el método `pop()` para mostrar una frase sobre la última moto que hemos comprado:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

last_owned = motorcycles.pop()
print(f"The last motorcycle I owned was a {last_owned.title()}.")
```

La salida es una oración sencilla sobre la moto más reciente que hemos tenido:

```
The last motorcycle I owned was a Suzuki.
```

Sacar elementos de cualquier posición de una lista

Podemos usar el método `pop()` para retirar un elemento de cualquier posición de una lista incluyendo el índice de ese elemento entre paréntesis.

```
motorcycles = ['honda', 'yamaha', 'suzuki']

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

Empezamos retirando la primera moto de la lista y a continuación imprimimos un mensaje sobre ella. La salida es una oración sencilla describiendo mi primera motocicleta:

```
The first motorcycle I owned was a Honda.
```

Recuerde que, cuando utilice `pop()`, el elemento con el que trabaje dejará de estar guardado en la lista.

Si no está seguro de si debe usar la sentencia `del` o el método `pop()`, siga este consejo para tomar su decisión: cuando quiera eliminar un elemento de una lista para no volver a usarlo jamás, use `del`; si desea usar un elemento al retirarlo de la lista, utilice `pop()`.

Eliminar un elemento por valor

A veces desconocemos la posición del valor que necesitamos eliminar de una lista. Si solo sabemos el valor del elemento, podemos usar el método `remove()`.

Por ejemplo, supongamos que queremos eliminar el valor `'ducati'` de la lista de motos.

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']
print(motorcycles)

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

El método `remove()` le indica a Python que averigüe dónde está 'ducati' en la lista y elimine ese elemento:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

También podemos usar el método `remove()` para trabajar con un valor que se va a eliminar de una lista. Vamos a eliminar el valor 'ducati' y a imprimir la razón por la que lo quitamos de la lista:

```
❶ motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']

print(motorcycles)

❷ too_expensive = 'ducati'

❸ motorcycles.remove(too_expensive)

print(motorcycles)

❹ print(f"\nA {too_expensive.title()} is too expensive for me.")
```

Después de definir la lista ❶, asignamos el valor 'ducati' a una variable llamada `too_expensive` ❷. Luego usamos esa variable para indicar a Python qué valor eliminar de la lista ❸. El valor 'ducati' se ha quitado de la lista ❹, pero todavía se puede acceder a él a través de la variable `too_expensive`, lo que nos permite imprimir una oración explicando que hemos quitado 'ducati' de nuestra lista de motos porque es demasiado cara:

```
['honda', 'yamaha', 'suzuki', 'ducati']
['honda', 'yamaha', 'suzuki']
```

A Ducati is too expensive for me.

Nota: El método `remove()` borra solo la primera aparición del valor especificado. Si cabe la posibilidad de que el valor aparezca más de una vez

en la lista, habrá que usar un bucle para asegurarnos de eliminar todas las apariciones. Veremos cómo hacerlo en el capítulo 7.

PRUÉBELO

Los siguientes ejercicios son un poco más difíciles que los del capítulo 2, pero le dan la oportunidad de trabajar con listas de todas las formas descritas.

- **3-4. Lista de invitados:** Si pudiese invitar a cualquiera, vivo o muerto, a cenar, ¿a quién invitaría? Haga una lista de al menos tres personas a las que le gustaría invitar a una cena y úsela para imprimir un mensaje para cada persona invitándole a cenar.
- **3-5. Cambiar la lista de invitados:** Acaba de enterarse de que uno de sus invitados no podrá asistir, así que tiene que enviar un nuevo juego de invitaciones. Tendrá que pensar en otra persona para invitar.
 - Empiece con el programa del ejercicio 3-4. Añada una llamada a `print()` al final del programa indicando el nombre del invitado que no puede asistir.
 - Modifique la lista sustituyendo el nombre del invitado que no puede venir por el de otra persona a la que va a invitar en su lugar.
 - Imprima un segundo grupo de mensajes de invitación, uno para cada persona que haya en la lista ahora.
- **3-6. Más invitados:** Acaba de encontrar una mesa más grande, así que dispone de más espacio. Piense en otros tres invitados más.
 - Empiece con el programa del ejercicio 3-4 o 3-5. Añada una llamada a `print()` al final para informar a la gente de que ha encontrado una mesa de comedor más grande.
 - Use `insert()` para añadir un nuevo invitado al principio de la lista.
 - Use `insert()` para añadir un nuevo invitado en mitad de la lista.
 - Use `append()` para añadir un nuevo invitado al final de la lista.
 - Imprima un nuevo juego de mensajes de invitación, uno para cada persona de la lista.
- **3-7. Reducir la lista de invitados:** Acaba de descubrir que la mesa no llegará a tiempo para la cena y solo tiene espacio para dos invitados.

- Empiece con el programa de 3-6. Añada una línea que imprima un mensaje diciendo que solo puede invitar a dos personas a cenar.
- Use `pop()` para eliminar invitados de la lista de uno en uno hasta que solo queden dos. Cada vez que retire un nombre de la lista, imprima un mensaje para esa persona diciendo que lo siente, pero que no puede invitarle a cenar.
- Imprima un mensaje para cada una de las dos personas que quedan en la lista informándoles de que siguen invitados.
- Use `del` para borrar los dos últimos nombres de la lista y que se quede vacía. Imprima la lista para asegurarse de que realmente tiene una lista vacía al final del programa.

Organizar una lista

Con frecuencia, las listas se crean siguiendo un orden impredecible, ya que no siempre podemos controlar el orden en el que los usuarios introducen sus datos. Aunque esto es inevitable en la mayoría de las circunstancias, con frecuencia necesitará presentar su información en un orden concreto. A veces, querrá conservar el orden de la lista, pero en otras tendrá que cambiar el orden original. Python ofrece distintas formas de organizar listas, dependiendo de la situación.

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

El método `sort()` de Python hace que sea relativamente fácil ordenar una lista. Imagine que tenemos una lista de coches y queremos ordenarla para guardarlos alfabéticamente. Para simplificar la tarea, asumiremos que todos los valores de la lista están en minúsculas.

`cars.py`

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

```
cars.sort()  
print(cars)
```

El método `sort()` cambia el orden de la lista de forma permanente. Los coches están ahora en orden alfabético y ya no se puede volver al orden original:

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

También se puede ordenar esta lista en orden alfabético inverso, pasando el argumento `reverse=True` al método `sort()`. El siguiente ejemplo ordena la lista de coches en orden alfabético inverso:

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

De nuevo, el cambio en el orden de la lista es permanente:

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

Ordenar una lista temporalmente con la función `sorted()`

Para mantener el orden original de una lista, pero presentarla ordenada de otra forma, podemos usar la función `sorted()`, que nos permite mostrar la lista en un orden concreto pero no afecta al orden real de la misma.

Vamos a probar esta función con la lista de coches.

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

- ❶

```
print("Here is the original list:")  
print(cars)
```
- ❷

```
print("\nHere is the sorted list:")
```

```
print(sorted(cars))

❸ print("\nHere is the original list again:")
print(cars)
```

Primero, imprimimos la lista en su orden original ❶ y luego en orden alfabético ❷. Después de mostrarla en un orden nuevo, mostramos que la lista sigue guardada en su orden original ❸.

```
Here is the original list:
['bmw', 'audi', 'toyota', 'subaru']
```

```
Here is the sorted list:
['audi', 'bmw', 'subaru', 'toyota']
```

❶ Here is the original list again:
['bmw', 'audi', 'toyota', 'subaru']

Observe que la lista sigue existiendo en su orden original en ❶ después de usar la función `sorted()`. Esta función también acepta un argumento `reverse=True` si se quiere mostrar una lista en orden alfabético inverso.

Nota: Ordenar una lista alfabéticamente es un poco más complicado cuando los valores no están en minúsculas. Hay varias formas de interpretar las mayúsculas al determinar un orden y especificar el orden exacto puede ser más complejo de lo que nos interesa ahora. Sin embargo, la mayoría de las aproximaciones a la organización se basan directamente en lo que hemos visto en esta sección.

Imprimir una lista en orden inverso

Para invertir el orden original de una lista, podemos usar el método `reverse()`. Si guardamos originalmente los coches en orden cronológico tomando en consideración el periodo en el que fuimos sus propietarios, podríamos reorganizar la lista fácilmente en orden cronológico inverso:

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

cars.reverse()
print(cars)
```

Observe que `reverse()` no genera un orden alfabético inverso, sino que simplemente invierte el orden de la lista:

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

El método `reverse()` cambia el orden de una lista de forma permanente, pero se puede volver al orden original aplicándolo de nuevo a la misma lista.

Descubrir la longitud de una lista

Podemos descubrir rápidamente la longitud de una lista con la función `len()`. La lista de este ejemplo tiene cuatro elementos, así que su longitud es ⁴:

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(cars)
```

4

La función `len()` le resultará útil cuando tenga que identificar el número de alienígenas pendientes de disparar en un juego, determinar la cantidad de datos que tiene que gestionar en una

visualización o saber el número de usuarios registrados en un sitio web, entre otras tareas.

Nota: Python cuenta los elementos de una lista empezando en uno, así que no debería toparse con errores por uno al determinar la longitud de una lista.

PRUÉBELO

- **3-8. Ver el mundo:** Piense en al menos cinco lugares del mundo que le gustaría visitar.
 - Guarde esos lugares en una lista. Asegúrese de no hacerlo en orden alfabético.
 - Imprima la lista en su orden original. No se preocupe por el formato, simplemente imprimala como una lista de Python en bruto.
 - Use `sorted()` para imprimir la lista en orden alfabético sin modificarla realmente.
 - Compruebe que la lista sigue en su orden original imprimiéndola.
 - Use `sorted()` para imprimir la lista en orden alfabético inverso sin cambiar el orden de la lista original.
 - Compruebe que la lista sigue en su orden original imprimiéndola otra vez.
 - Use `reverse()` para cambiar el orden de la lista. Imprímala para comprobar que el orden ha cambiado.
 - Use `reverse()` para cambiar el orden de la lista de nuevo. Imprímala para comprobar que ha vuelto al orden original.
 - Use `sort()` para cambiar la lista y guardarla en orden alfabético. Imprímala para comprobar que el orden ha cambiado.
 - Use `sort()` para cambiar la lista y guardarla en orden alfabético inverso. Imprímala para comprobar que el orden ha cambiado.
- **3-9. Invitados a la cena:** Trabajando con uno de los programas de los ejercicios 3-4 a 3-7, utilice `len()` para imprimir un mensaje indicando el número de personas que va a invitar a cenar.
- **3-10. Todas las funciones:** Piense en algo que podría guardar en una lista. Por ejemplo, podría hacer una lista de montañas, ríos, países, ciudades,

idiomas o cualquier otra cosa. Escriba un programa que cree una lista que contenga estos elementos y use cada función presentada en este capítulo al menos una vez.

Evitar errores de índice al trabajar con listas

Hay un tipo de error habitual cuando se trabaja con listas por primera vez. Supongamos que tiene una lista con tres elementos y preguntamos por el cuarto:

motorcycles.py

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[3])
```

Este ejemplo da como resultado un error de índice:

```
Traceback (most recent call last):
  File "motorcycles.py", line 2, in <module>
    print(motorcycles[3])
    ~~~~~^~~~^~~^
IndexError: list index out of range
```

Python intenta darle el elemento con índice `3`, pero, al buscarlo en la lista `motorcycles`, descubre que tal índice no existe. Dada la naturaleza de empezar a contar en 0 de las listas, este error es típico. La gente cree que el tercer elemento es el número 3 porque empiezan a contar en 1. Pero, en Python, el tercer elemento es el 2 porque empieza a indexar en 0.

Un error de índice significa que Python no puede encontrar un elemento en la posición solicitada. Si se produce en su programa, pruebe a ajustar en uno el índice que busca. Vuelva a ejecutar el programa para ver si el resultado es correcto.

Recuerde que, cuando quiera acceder al último elemento de una lista, debería usar el índice `-1`. Siempre funciona, incluso aunque la

lista haya cambiado de tamaño desde su último acceso:

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles[-1])
```

El índice `-1` siempre devuelve el último elemento de una lista, en este caso, el valor `'suzuki'`:

```
'suzuki'
```

La única ocasión en la que este enfoque provocará un error es cuando solicite el último elemento de una lista vacía:

```
motorcycles = []
print(motorcycles[-1])
```

Como no hay elementos en `motorcycles`, Python devuelve otro error de índice:

```
Traceback (most recent call last):
  File "motorcycles.py", line 3, in <module>
    print(motorcycles[-1])
    ~~~~~^~~~~~^~~^
IndexError: list index out of range
```

Nota: Si se produce un error de índice y no sabe cómo resolverlo, pruebe a imprimir la lista o imprima su longitud. Su lista puede ser muy diferente a lo que imaginaba, especialmente si la ha gestionado dinámicamente su programa. Ver la lista real o el número exacto de elementos que contiene puede ayudarle a resolver ese tipo de errores lógicos.

PRUÉBELO

- **3-11. Error intencionado:** Si no ha recibido todavía un error de índice en ninguno de sus programas, fuércelo. Cambie un índice en un programa para

producir un error, pero asegúrese de corregirlo antes de cerrar el programa.

Resumen

En este capítulo, hemos visto qué son las listas y cómo trabajar con sus elementos individuales. Ha aprendido a definir una lista y añadir y eliminar elementos. Ha aprendido a ordenar listas de forma permanente y temporal para mostrarlas, cómo descubrir la longitud de una lista y cómo evitar errores de índice al trabajar con listas.

En el capítulo 4, veremos cómo trabajar con los elementos de una lista de una forma más eficiente. Al pasar en bucle por cada elemento de una lista usando unas pocas líneas de código, podrá trabajar con eficiencia incluso cuando sus listas contengan miles o millones de elementos.

4

TRABAJO CON LISTAS



En el capítulo 3 hemos aprendido a crear una lista sencilla y a trabajar con sus elementos individuales. En este capítulo, descubriremos cómo pasar en bucle por una lista entera usando solo unas cuantas líneas de código, independientemente de la longitud de la lista.

Los bucles nos permiten realizar la misma acción, o el mismo conjunto de acciones, con todos los elementos de una lista. Como resultado, podemos trabajar eficientemente con listas de cualquier longitud, incluidas las que tienen miles o incluso millones de elementos.

Pasar en bucle por una lista completa

A menudo nos interesa pasar por todas las entradas de una lista, realizando la misma acción con cada elemento. Por ejemplo, en un juego podríamos necesitar desplazar cada elemento por la pantalla en cantidades iguales, o bien en una lista de números podríamos tener que realizar la misma operación estadística con cada elemento. O tal vez queramos mostrar todos los titulares de una lista de artículos de un sitio web. Para realizar la misma acción con todos los elementos de una lista, se puede usar el bucle `for` de Python.

Imaginemos que tenemos una lista de nombres de magos y queremos imprimir todos los nombres de la lista. Podríamos hacerlo recuperando cada nombre individualmente, pero este enfoque daría lugar a varios problemas. Para empezar, sería muy repetitivo hacerlo con una lista de nombres muy larga. Además, tendríamos que

cambiar el código cada vez que cambiase la longitud de la lista. Un bucle `for` evitaría estos problemas, permitiendo a Python gestionar estos asuntos internamente.

Vamos a usar un bucle `for` para imprimir cada nombre en una lista de magos:

`magicians.py`

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

Empezamos definiendo una lista, igual que hicimos en el capítulo 3. A continuación, definimos un bucle `for`. Esta línea le pide a Python que saque un nombre de la lista de magos y los asocie con la variable `magician`. Seguidamente, le decimos a Python que imprima el nombre que acaba de asignarse a `magician`. Después, Python repite las líneas 2 y 3 una vez por cada nombre de la lista. Este código sería como decir: "Por cada nombre de mago de la lista, imprime el nombre del mago". La salida es una impresión simple de cada nombre de la lista:

```
alice
david
carolina
```

Los bucles en detalle

El concepto de bucle es importante, ya que es una de las formas más habituales en las que un ordenador automatiza tareas repetitivas. Por ejemplo, en un bucle sencillo como el que hemos usado en `magicians.py`, Python lee inicialmente la primera línea del bucle:

```
for magician in magicians:
```

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

```
print(magician)
```

Python imprime el valor actual de `magician`, que es todavía `'alice'`. Como la lista contiene más valores, Python regresa a la primera línea del código:

```
for magician in magicians:
```

Python recupera el siguiente nombre de la lista, `'david'`, y asocia ese valor con la variable `magician`. Después, ejecuta la línea:

```
print(magician)
```

Python imprime el valor actual de `magician`, que ahora es `'david'`. Luego repite el bucle entero una vez más con el último valor de la lista, `'carolina'`. Como ya no hay más valores, Python pasa a la siguiente línea del programa. En este caso no hay nada después del bucle `for`, así que se acaba el programa.

Cuando use bucles por primera vez, tenga en cuenta que el conjunto de pasos se repite una vez por cada elemento de la lista, independientemente de cuántos haya. Si su lista tiene un millón de elementos, Python repite estos pasos un millón de veces, normalmente muy deprisa.

Tenga también en cuenta al escribir sus propios bucles `for` que puede elegir cualquier nombre que quiera para la variable temporal que se asociará con cada valor de la lista. Sin embargo, conviene usar un nombre significativo que represente un solo elemento de la lista. Por ejemplo, esta sería una buena forma de iniciar un bucle `for` para una lista de gatos, otra de perros y otra general:

```
for cat in cats:
```

```
for dog in dogs:  
    for item in list_of_items:
```

Estas convenciones de nomenclatura pueden ayudarle a seguir la acción que se realiza con cada elemento dentro de un bucle `for`. Usar nombres en singular y plural puede ayudarle a identificar si una sección de código trabaja con un solo elemento de una lista o con la lista entera.

Sacar más partido a un bucle `for`

Podemos hacer prácticamente cualquier cosa con cada elemento en un bucle `for`. Siguiendo con el ejemplo anterior, vamos a imprimir un mensaje para cada mago, diciéndoles que han hecho un truco genial:

magicians.py

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

La única diferencia en este código está en dónde componemos un mensaje para cada mago empezando con su nombre. En el primer paso por el bucle, el nombre del mago es `'alice'`, así que Python empieza el primer mensaje con el nombre `'Alice'`. En el segundo paso, el mensaje empezará con `'David'` y, en el tercero, con `'Carolina'`.

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

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

Podemos escribir todas las líneas de código que queramos en el bucle `for`. Todas las líneas sangradas bajo la línea `for magician in magicians` se consideran dentro del bucle y cada una se ejecuta una vez por cada valor de la lista. Así pues, podemos trabajar tanto como queramos con cada valor de la lista.

Vamos a añadir una segunda línea a nuestro mensaje diciendo a cada mago que estamos deseando ver su siguiente truco:

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

Como hemos sangrado ambas llamadas a `print()`, cada línea se ejecutará una vez con cada mago de la lista. La nueva línea ("`\n`") en la segunda llamada a `print()` inserta una línea vacía tras cada paso por el bucle. Así creamos mensajes agrupados claramente para cada persona de la lista:

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

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

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

Puede usar todas las líneas que quiera en sus bucles `for`. En la práctica, con frecuencia encontrará útil realizar varias operaciones diferentes con cada elemento de una lista usando uno de estos bucles.

Hacer algo después de un bucle `for`

¿Qué pasa cuando el bucle `for` termina de ejecutarse? Normalmente, querremos resumir un bloque de salida o pasar a la siguiente tarea que deba completar el programa.

Todas las líneas de código no sangradas que haya después de un bucle `for` se ejecutan una vez, sin repeticiones. Vamos a escribir un mensaje de agradecimiento al grupo de magos en conjunto, dándoles las gracias por un gran espectáculo. Para mostrar este mensaje grupal después de todos los mensajes individuales que hemos imprimido, colocaremos el agradecimiento detrás del bucle `for`, sin sangría:

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

Las dos primeras llamadas a `print()` se repiten una vez por cada mago de la lista. Sin embargo, dado que la última línea no está sangrada, se imprime una sola vez:

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

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

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

```
Thank you, everyone. That was a great magic show!
```

Cuando procese datos con un bucle `for`, se dará cuenta de que esta es una buena forma de resumir una operación realizada con un conjunto de datos completo. Por ejemplo, podría usar un bucle `for`

para inicializar un juego pasando por una lista de personajes y mostrando cada uno en la pantalla. Después, podría escribir código adicional después del bucle para que aparezca un botón para empezar a jugar cuando ya se hayan dibujado todos los personajes en la pantalla.

Evitar errores de sangrado

Python usa el sangrado para determinar cómo se relaciona una línea, o un grupo de líneas, con el resto del programa. En los ejemplos anteriores, las líneas que imprimían mensajes para los magos individuales eran parte del bucle `for` porque estaban sangradas. El uso de sangrías en Python facilita la lectura del código. Básicamente, Python utiliza el espacio en blanco para obligarnos a escribir código con un buen formato y una estructura visual clara. En programas de Python más largos, verá que los bloques de código presentan distintos niveles de sangrado. Estos niveles nos ayudan a tener una visión general de la organización global del programa.

Cuando empiece a escribir código que dependa de un sangrado adecuado, deberá tener cuidado con una serie de errores habituales. Por ejemplo, a veces la gente sangra líneas que no deberían estar sangradas u olvida sangrar otras que sí deberían estarlo. Ver ejemplos de estos errores ahora le ayudará a evitarlos en el futuro y a corregirlos si aparecen en sus programas.

Veamos algunos de los errores de sangrado más habituales.

Olvidar la sangría

Siempre hay que sangrar la línea que va justo después de la sentencia `for` de un bucle. Si no lo hacemos, Python nos lo recordará:

magicians.py

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

```
for magician in magicians:  
❶    print(magician)
```

La llamada a `print()` ❶ debería estar sangrada, pero no lo está. Cuando Python espera un bloque sangrado y no encuentra ninguno, nos indica en qué línea ha identificado el problema:

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

Por lo general este tipo de error se resuelve sangrando la(s) línea(s) inmediatamente después de la sentencia `for`.

Olvidar sangrar líneas adicionales

A veces, un bucle se ejecuta sin errores, pero no produce el resultado esperado. Esto puede suceder cuando intentamos hacer varias tareas en un bucle y olvidamos sangrar algunas líneas. Por ejemplo, esto es lo que pasa cuando olvidamos sangrar la segunda línea del bucle, que le dice a cada mago que estamos deseando ver su próximo truco:

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

La segunda llamada a `print()` ❶ debería estar sangrada, pero, como Python encuentra al menos una sangría después de la sentencia `for`, no informa de ningún error. El resultado es que la primera llamada a `print()` se ejecuta una vez para cada nombre de la

lista porque está sangrada. La segunda llamada a `print()` no está sangrada, así que se ejecutará solo una vez, cuando se haya completado el bucle. Puesto que el valor final está asociado a la maga 'carolina', es la única que recibe el mensaje sobre el próximo truco:

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

Se trata de un error lógico. La sintaxis es código Python válido, pero el código no produce el resultado deseado porque la lógica falla. Si espera ver una acción repetida una vez para cada elemento de una lista, pero se ejecuta una sola vez, compruebe si solo hay que sangrar una línea o un grupo de líneas.

Sangrados innecesarios

Si crea accidentalmente una sangría en una línea que no debería ir sangrada, Python le informará de la sangría inesperada:

hello_world.py

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

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

```
File "hello_world.py", line 2  
    print(message)  
    ^  
IndentationError: unexpected indent
```

Puede evitar este tipo de errores usando sangrías solo cuando tenga una razón concreta para hacerlo. En los programas que estamos escribiendo de momento, las únicas líneas que deberían ir sangradas son las acciones que deben repetirse con cada elemento en un bucle `for`.

Sangrado innecesario después de un bucle

Si crea accidentalmente una sangría en un código que debería ejecutarse cuando haya terminado un bucle, ese código se repetirá una vez por cada elemento de la lista. A veces, esto hace que Python informe de un error, pero, con frecuencia, suele ser un error lógico.

Por ejemplo, vamos a ver qué pasa si sangramos por error la línea que daba las gracias al grupo de magos por el espectáculo conjunto:

magicians.py

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

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

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

Thank you everyone, that was a great magic show!
David, that was a great trick!
```

I can't wait to see your next trick, David.

Thank you everyone, that was a great magic show!

Carolina, that was a great trick!

I can't wait to see your next trick, Carolina.

Thank you everyone, that was a great magic show!

Se trata de otro error lógico, similar al del apartado "Olvidar sangrar líneas adicionales". Como Python no sabe lo que pretendemos hacer con el código, ejecutará todo aquello que esté escrito con una sintaxis válida. Si una acción se repite varias veces cuando debería ejecutarse solo una, seguramente sea necesario eliminar la sangría de esa acción.

Olvidar los dos puntos

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

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

Si olvida accidentalmente los dos puntos ❶, obtendrá un error de sintaxis, porque Python no sabrá exactamente qué queremos hacer:

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

---


```

Python no sabe si simplemente hemos olvidado los dos puntos o si queríamos escribir más código para configurar un bucle más complejo. Si el intérprete puede identificar una posible solución, se

la sugerirá, como por ejemplo añadir dos puntos al final de una línea (como hace aquí con la respuesta `expected '...'`). Algunos errores tienen una solución fácil y obvia, gracias a las sugerencias de Python. Otros errores son mucho más difíciles de resolver, aun cuando la solución afecte a un único carácter. No se siente mal cuando le lleve mucho tiempo encontrar un error pequeño; no está solo.

PRUÉBELO

- **4-1. Pizzas:** Piense en al menos tres de sus pizzas favoritas. Guarde estos nombres en una lista y use un bucle `for` para imprimir el nombre de cada pizza.
 - Modifique su bucle `for` para imprimir una oración usando el nombre de la pizza en vez de solo el nombre. Para cada pizza debería tener al menos una línea de salida con una oración simple, como "Me gusta la pizza de pepperoni".
 - Añada una línea al final del programa, fuera del bucle, que indique cuánto le gusta la pizza. La salida debería tener tres o más líneas sobre sus pizzas favoritas y al final una frase adicional, como "¡Me encanta la pizza!"
- **4-2. Animales:** Piense en al menos tres animales diferentes que tengan una característica en común. Guarde los nombres de estos animales en una lista y use un bucle `for` para imprimir el nombre de cada animal.
 - Modifique su programa para imprimir una oración sobre cada animal, como "Un perro sería una excelente mascota".
 - Añada una línea al final del programa diciendo qué tienen estos animales en común. Podría imprimir una frase como "¡Cualquiera de estos animales sería una excelente mascota!".

Hacer listas numéricas

Hay muchas razones para almacenar un conjunto de números. Por ejemplo, necesitará seguir las posiciones de cada personaje en

un juego y puede que también tenga que llevar un registro de las puntuaciones más altas de un jugador. Al visualizar datos, casi con toda seguridad trabajará con números, como temperaturas, distancias, densidades de población o valores de latitud y longitud, entre otros tipos de conjuntos numéricos.

Las listas son perfectas para guardar conjuntos de datos. Python ofrece varias herramientas para trabajar eficientemente con listas de números. Cuando entienda cómo usar estas herramientas con eficacia, su código funcionará incluso aunque las listas contengan millones de elementos.

Utilizar la función range()

La función `range()` de Python hace que sea fácil generar una serie de números. Por ejemplo, podemos usarla para imprimir una serie de números como esta:

First_numbers.py

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

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

1
2
3
4

En este ejemplo, `range()` imprime solo los números del 1 al 4. Es otra consecuencia del comportamiento de error por uno que a menudo encontramos en los lenguajes de programación. La función `range()` hace que Python empiece a contar en el primer valor que le damos y se detiene cuando llega al segundo valor proporcionado. Al

detenerse en el segundo valor, la salida nunca contiene ese valor final, que en este caso habría sido 5.

Para imprimir los números del 1 al 5, tendríamos que usar `range(1, 6)`:

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

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

```
1
2
3
4
5
```

Si su salida es diferente de lo que esperaba al usar `range()`, pruebe a ajustar el valor final en 1.

También puede pasar a `range()` un solo argumento para que empiece la secuencia de números en 0. Por ejemplo, `range(6)` devolvería los números del 0 al 5.

Usar `range()` para hacer una lista de números

Si quiere hacer una lista de números, puede convertir los resultados de `range()` directamente en una lista usando la función `list()`. Al meter una llamada a la función `range()` en `list()`, la salida será una lista de números.

En el ejemplo del apartado anterior, solo hemos imprimido una serie de números. Podemos usar `list()` para convertir ese mismo conjunto en una lista:

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

Este es el resultado:

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

También podemos usar la función `range()` para decirle a Python que se salte números en un rango determinado. Si pasamos un tercer argumento a la función, Python usa ese valor como tamaño de paso al generar números.

El siguiente ejemplo hace una lista de números pares entre el 0 y el 10:

`even_numbers.py`

```
even_numbers = list(range(2, 11, 2))
print(even_numbers)
```

En este ejemplo, la función `range()` empieza con el valor 2 y suma 2 a ese valor. Suma 2 repetidas veces hasta alcanzar o superar el valor final, 11, y produce este resultado:

```
[2, 4, 6, 8, 10]
```

Podemos crear prácticamente cualquier conjunto de números que queramos usando la función `range()`. Por ejemplo, piense cómo podríamos hacer una lista de los 10 primeros números cuadrados (es decir, el cuadrado de cada entero del 1 al 10). En Python, dos asteriscos (`**`) representan exponentes. Así es como pondríamos los 10 primeros números cuadrados en una lista:

`squares_numbers.py`

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

print(squares)
```

Empezamos con una lista llamada `squares`. A continuación, le indicamos a Python que pase en bucle por cada valor entre el 1 y el 10 usando la función `range()`. Dentro del bucle, el valor actual se eleva a la segunda potencia y se asigna a la variable `square` ❶. Cada nuevo valor de `square` se adjunta a la lista `squares` ❷. Por último, cuando el bucle ha terminado de ejecutarse, se imprime la lista de cuadrados:

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

Para escribir este código de una forma más concisa, omita la variable temporal `square` y adjunte cada nuevo valor directamente a la lista:

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

print(squares)
```

Esta línea de código hace lo mismo que las líneas dentro del bucle `loop` en la lista anterior. Cada valor del bucle se eleva a la segunda potencia y se añade de inmediato a la lista de cuadrados.

Puede usar cualquiera de estos dos enfoques a la hora de crear listas más complejas. En ocasiones, usar una variable temporal hace que sea más fácil leer el código; otras veces, hace el código innecesariamente largo. Concéntrese primero en escribir código que pueda entender con claridad y le permita hacer lo que quiera que haga. Más adelante podrá investigar enfoques más eficaces a medida que revise su código.

Estadística sencilla con una lista de números

Hay unas cuantas funciones de Python que resultan de utilidad para trabajar con listas de números. Por ejemplo, podemos hallar

fácilmente el mínimo, el máximo y la suma de una lista de números:

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

Nota: Los ejemplos de esta sección utilizan listas cortas de números que encajen sin problema en las páginas, pero funcionarían igual de bien si las listas contuvieran un millón de números o más.

Listas por comprensión

El enfoque descrito antes para generar la lista `squares` consistía en usar tres o cuatro líneas de código. Las listas por comprensión permiten generar la misma lista con una sola línea de código. Esta construcción combina el bucle `for` y la creación de nuevos elementos en una línea y añade automáticamente cada elemento nuevo. Las listas por comprensión no siempre se enseñan a los principiantes, pero las he incluido aquí porque es probable que las descubra en cuanto empiece a mirar el código de otras personas.

El siguiente ejemplo crea la misma lista de números cuadrados que hemos visto antes, pero usa listas por comprensión:

`squares.py`

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

Para usar esta sintaxis, empiece con un nombre descriptivo para la lista, como `squares` (cuadrados) en este caso. A continuación, abra corchetes y defina la expresión para los valores que desea

almacenar en la nueva lista. En este ejemplo, la expresión es `value**2`, que eleva el valor a la segunda potencia. Después, escriba un bucle `for` para generar los números que va a dar a la expresión y cierre los corchetes. El bucle `for` de este ejemplo es `for value in range(1, 11)`, que aporta los valores del 1 al 10 a la expresión `value**2`. Observe que no utilizamos dos puntos al final de la sentencia `for`.

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

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

Requiere práctica escribir nuestras propias listas por comprensión, pero las encontrará útiles cuando se familiarice con la creación de listas ordinarias. Cuando esté escribiendo tres o cuatro líneas de código para generar listas y se le empiece a hacer repetitivo, considere pasar a las listas por comprensión.

PRUÉBELO

- **4-3. Contar hasta veinte:** Use un bucle `for` para imprimir los números del 1 al 20, ambos incluidos.
- **4-4. Un millón:** Haga una lista de números de uno a un millón y luego use un bucle `for` para imprimir los números. (Si la salida tarda mucho, deténgala pulsando **Control-C** o cerrando la ventana de salida).
- **4-5. Sumar un millón:** Haga una lista de los números del uno al millón y use `min()` y `max()` para asegurarse de que su lista empieza realmente en uno y acaba en un millón. Utilice también la función `sum()` para ver lo rápido que Python puede sumar un millón de números.
- **4-6. Números impares:** Use el tercer argumento de la función `range()` para hacer una lista de los números impares comprendidos entre 1 y 20. Utilice un bucle `for` para imprimir cada número.
- **4-7. Treses:** Haga una lista de los múltiplos de 3 comprendidos entre el 3 y el 30. Use un bucle `for` para imprimir los números de la lista.

- **4-8. Cubos:** Un número elevado a la tercera potencia es un cubo. Por ejemplo, el cubo de 2 se escribe como `2**3` en Python. Haga una lista de los 10 primeros cubos (es decir, el cubo de cada entero entre 1 y 10) y use un bucle `for` para imprimir el valor de cada cubo.
- **4-9. Comprensión de cubos:** Use una lista por comprensión de los 10 primeros cubos.

Trabajar con parte de una lista

En el capítulo 3, vimos cómo acceder a elementos individuales de una lista y en este hemos visto cómo trabajar con todos los elementos de una lista. También podemos trabajar con un grupo específico de elementos en una lista, que en Python denominamos "trozo" o *slice*.

Partir una lista

Para dividir una lista, es preciso especificar los índices del primero y el último elemento con los que queremos trabajar. Al igual que con la función `range()`, Python se detiene un elemento antes del segundo índice especificado. Para sacar los tres primeros elementos de una lista, deberíamos solicitar los índices del 0 al 3 y así obtendríamos los elementos `0`, `1` y `2`.

El siguiente ejemplo presenta una lista de jugadores en un equipo:

`players.py`

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

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

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

Podemos generar cualquier subconjunto de una lista. Por ejemplo, si quisieramos el segundo, tercer y cuarto elemento de una lista, empezaríamos el trozo en el índice `1` y terminaríamos en el `4`:

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

Esta vez, el trozo empieza con `'martina'` y termina con `'florence'`:

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

Si omitimos el primer índice en un trozo, Python empieza automáticamente en el principio de la lista:

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

Sin un índice inicial, Python empieza por el comienzo:

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

Si queremos que el trozo incluya el final de la lista, podemos usar una sintaxis similar. Por ejemplo, si queremos todos los elementos entre el tercero y el último, empezaremos con el índice `2` y omitiremos el segundo índice:

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

Python devuelve todos los elementos entre el tercero y el último de la lista:

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

Esta sintaxis nos permite sacar todos los elementos desde cualquier punto de la lista hasta el final, independientemente de lo larga que sea la lista. Recuerde que un índice negativo devuelve un elemento a una distancia dada del final de la lista: así pues, podemos sacar cualquier trozo desde el final de una lista. Por ejemplo, si queremos los tres últimos jugadores de la lista, podemos usar el trozo `players[-3:]`:

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

Esto imprime los nombres de los tres últimos jugadores y seguiría funcionando, aunque la lista cambiase de tamaño.

Nota: Puede incluir un tercer valor entre los paréntesis que indican un trozo. Si lo hace, estará diciendo a Python cuántas veces tiene que saltar entre elementos en el rango especificado.

Pasar en bucle por un trozo

Podemos usar un trozo con `for` si queremos pasar en bucle por un subconjunto de elementos en una lista. En el siguiente ejemplo, pasaremos por los tres primeros jugadores e imprimiremos sus nombres como parte de una lista sencilla:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print("Here are the first three players on my team:")
❶ for player in players[:3]:
    print(player.title())
```

En lugar de pasar en bucle por toda la lista de jugadores de ❶, Python pasa solo por los tres primeros nombres:

```
Here are the first three players on my team:
```

Charles

Martina

Michael

Los trozos o particiones son muy útiles en diversas situaciones. Por ejemplo, al crear un juego, podemos añadir el resultado final de un jugador a una lista cada vez que termine de jugar. A continuación, podríamos obtener sus tres mejores puntuaciones organizando la lista en orden decreciente y sacando un trozo que incluya solo los tres primeros resultados. Cuando se trabaja con datos, los trozos pueden servir para procesar los datos en secciones de un tamaño específico. Y, si estamos creando una aplicación web, podemos emplear trozos para mostrar información en una serie de páginas con una cantidad de información apropiada en cada una.

Copiar una lista

A menudo, conviene empezar con una lista existente y crear una totalmente nueva basada en ella. Veamos cómo hacerlo copiando una lista. Analizaremos una situación en la que copiar una lista es útil.

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

Por ejemplo, imagine que tenemos una lista de nuestra comida favorita y queremos hacer otra con la que le gusta a un amigo. A este amigo le gusta todo lo que hay en nuestra lista, así que podemos crear la suya copiando la nuestra:

foods.py

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

print("My favorite foods are:")
```

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

En primer lugar, realizamos una llamada a una lista de la comida que nos gusta y la llamamos `my_foods`. A continuación, creamos una nueva lista llamada `friend_foods`. Creamos una copia de `my_foods` pidiendo un trozo de `my_foods` sin especificar índices ❶ y la guardamos en `friend_foods`. Al imprimir cada lista, vemos que las dos contienen los mismos alimentos:

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

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

Para comprobar que, efectivamente, tenemos dos listas aparte, añadiremos una nueva comida a cada una y verificaremos que cada lista lleva la cuenta de la comida favorita de cada persona:

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

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

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

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

Copiamos los elementos originales de `my_foods` en la nueva lista `friend_foods`, igual que en el ejemplo anterior ❶. Después, añadimos un alimento nuevo a cada lista: añadimos '`cannoli`' a `my_foods` ❷ y agregamos '`ice cream`' a `friend_foods` ❸. Por último, imprimimos las dos listas para ver:

```
My favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake', 'cannoli']
```

```
My friend's favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake', 'ice cream']
```

La salida muestra que '`cannoli`' aparece ahora en nuestra lista de comida favorita, pero no es el caso de '`ice cream`'. Vemos que '`ice cream`' aparece ahora en la lista de nuestro amigo, pero '`cannoli`' no. Si simplemente hubiésemos establecido `friend_foods` igual a `my_foods`, no produciríamos dos listas separadas. Por ejemplo, esto es lo que pasa cuando uno intenta copiar una lista sin usar un trozo:

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

```
# Esto no funciona:
```

```
friend_foods = my_foods
```

```
my_foods.append('cannoli')
```

```
friend_foods.append('ice cream')
```

```
print("My favorite foods are:")
```

```
print(my_foods)
```

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

```
print(friend_foods)
```

En lugar de asignar una copia de `my_foods` a `friend_foods`, establecemos que `friend_foods` es igual a `my_foods`. Esta sintaxis le indica a Python que asocie la nueva variable `friend_foods` con la lista

que ya está asociada con `my_foods`, así que ahora las dos variables apuntan a la misma lista. Como resultado, al añadir `'cannoli'` a `my_foods`, también aparecerá en `friend_foods`. Del mismo modo, `'ice cream'` aparecerá en ambas listas, aunque dé la impresión de que solo se ha añadido a `friend_foods`.

La salida muestra que las dos listas son iguales ahora, pero eso no es lo que queríamos:

```
My favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

```
My friend's favorite foods are:
```

```
['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']
```

Nota: Por el momento, no se preocupe por los detalles de este ejemplo. Si está intentando trabajar con una copia de una lista y observa un comportamiento inesperado, asegúrese de copiar la lista como trozo como hemos hecho en el primer ejemplo.

PRUÉBELO

- **4-10. Trozos:** Partiendo de uno de los programas que hemos escrito en este capítulo, añada varias líneas al final que hagan lo siguiente:
 - Imprimir el mensaje "Estos son los tres primeros elementos de la lista:". A continuación, use un trozo para imprimir los tres primeros elementos de la lista de ese programa.
 - Imprimir el mensaje "Estos tres elementos están en el medio de la lista:". A continuación, use un trozo para imprimir los tres elementos centrales de la lista.
 - Imprimir el mensaje "Estos son los tres últimos elementos de la lista:". A continuación, use un trozo para imprimir los tres últimos elementos de la lista.
- **4-11. Mis pizzas, sus pizzas:** Empiece con el programa del ejercicio 4-1. Haga una copia de la lista de pizzas y llámela `friend_pizzas`. A

continuación, haga lo siguiente:

- Añada una pizza nueva a la lista original.
- Añada una pizza diferente a la lista `friend_pizzas`.
- Compruebe que tiene dos listas separadas. Imprima el mensaje "Mis pizzas favoritas son:" y luego use un bucle `for` para imprimir la primera lista. Imprima el mensaje "Las pizzas favoritas de mi amigo son:" y después utilice un bucle `for` para imprimir la segunda lista. Asegúrese de que cada pizza se guarda en la lista adecuada.
- **4-12. Más bucles:** Todas las versiones de `foods.py` de esta sección han evitado usar bucles `for` al imprimir para ahorrar espacio. Elija una versión de `foods.py` y escriba dos bucles para imprimir cada lista de comida.

Tuplas

Las listas funcionan bien para almacenar colecciones de elementos que pueden cambiar a lo largo de la vida de un programa. La capacidad para modificar listas es de especial importancia cuando se trabaja con una lista de usuarios de un sitio web o de personajes de un juego. Sin embargo, a veces necesitamos crear listas de elementos que no se puedan alterar. Eso es justo lo que podemos hacer con las tuplas. Python se refiere a los valores que no pueden cambiar como inmutables, y una lista inmutable se denomina "tupla".

Definir una tupla

Una tupla es muy parecida a una lista, solo que emplea paréntesis en lugar de corchetes. Una vez definida la tupla, podemos acceder a los elementos individuales usando sus índices, como haríamos con una lista. Por ejemplo, si tenemos un rectángulo que siempre debería tener el mismo tamaño, podemos asegurarnos de que no cambia incluyendo sus dimensiones en una tupla:

`dimensions.py`

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

Definimos la tupla `dimensions`, usando paréntesis en vez de corchetes. A continuación, imprimimos cada elemento de la tupla de manera individual con la misma sintaxis que hemos estado usando para acceder a los elementos de una lista:

```
200
50
```

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

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

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

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

Esto es bueno porque queremos que Python dé error cuando una línea de código intente cambiar las dimensiones del rectángulo.

Nota: Las tuplas se definen técnicamente por la presencia de una coma; los paréntesis las hacen parecer más claras y legibles. Si queremos definir una tupla con un solo elemento, tendremos que incluir una coma:

```
my_t = (3,)
```

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

Pasar en bucle por todos los valores de una tupla

Podemos pasar en bucle por todos los valores de una tupla con un bucle `for`, igual que hemos hecho con las listas:

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

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

```
200
50
```

Sobrescribir una tupla

Aunque no se puede modificar una tupla, sí se puede asignar un nuevo valor a una variable que representa una tupla. Por ejemplo, si quisiéramos cambiar las dimensiones de este rectángulo, podríamos redefinir la tupla entera:

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

dimensions = (400, 100)
print("\nModified dimensions:")
for dimension in dimensions:
    print(dimension)
```

Las cuatro primeras líneas definen la tupla original e imprimen las dimensiones iniciales. A continuación, asociamos la nueva tupla con la variable `dimensions` e imprimimos las nuevas dimensiones. Python no da error esta vez porque reasignar una variable es válido:

Original dimensions:

```
200  
50
```

Modified dimensions:

```
400  
100
```

En comparación con las listas, las tuplas son estructuras de datos simples. Utilícelas cuando quiera guardar un conjunto de valores que no deberían cambiar durante la vida de un programa.

PRUÉBELO

- **4-13. Bufé:** Un restaurante de tipo bufé ofrece solo cinco comidas básicas. Piense en cinco platos básicos y guárdelos en una tupla.
 - Use un bucle `for` para imprimir cada comida que ofrece el restaurante.
 - Intente modificar alguno de los elementos y asegúrese de que Python rechaza el cambio.
 - El restaurante cambia su menú, sustituyendo dos elementos por comidas diferentes. Añada una línea que rehaga la tupla y use un bucle `for` para imprimir cada uno de los elementos del menú modificado.

Dar estilo a nuestro código

Ahora que empezamos a escribir programas más largos, es una buena idea aprender a dar a nuestro código un estilo consistente. Tómese su tiempo para hacer que su código sea tan fácil de leer como sea posible. Escribir código fácilmente legible nos ayuda a

saber lo que hacen nuestros programas y también ayuda a otros a entender nuestro código.

Los programadores de Python han acordado una serie de convenciones de estilo para asegurarse de que el código de todo el mundo se estructura más o menos de la misma forma. Cuando ya sepa escribir código Python limpio, debería ser capaz de entender la estructura general del de cualquier otro programador, siempre que sigan las mismas directrices. Si aspira a convertirse en programador profesional, debería empezar a seguir esas directrices lo antes posible para desarrollar buenos hábitos.

La guía de estilo

Cuando alguien quiere hacer un cambio en el lenguaje Python, escriben una PEP (*Python Enhancement Proposal*, propuesta de mejora de Python). Una de las más antiguas es la PEP 8, que instruye a los programadores de Python en la estilización de código. La PEP 8 es bastante larga, pero buena parte de ella está relacionada con estructuras de código más complejas que las que hemos visto hasta ahora.

La guía de estilo de Python se escribió entendiendo que es más frecuente leer código que escribir código. Escribimos el código una vez y luego empezamos a leerlo para depurarlo. Cuando añadimos funciones a un programa, pasamos más tiempo leyendo nuestro código. Cuando compartimos código con otros programadores, también leerán nuestro código.

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

Sangrado

La PEP 8 recomienda usar cuatro espacios por nivel de sangrado. Usar cuatro espacios mejora la legibilidad y deja sitio para varios

niveles de sangrado en cada línea.

En un documento elaborado con un procesador de texto, suelen usarse tabulaciones en vez de espacios para crear sangrías, pero el intérprete de Python se confunde cuando se mezclan tabulaciones y espacios. Todos los editores de texto ofrecen alguna configuración que permite usar el tabulador, pero luego convierte esa tabulación en un número de espacios. Debería usar el tabulador, pero asegúrese de que su editor está configurado para insertar espacios en vez de tabulaciones en el documento.

Mezclar tabuladores y espacios en un archivo puede causar problemas difíciles de diagnosticar. Si cree que tiene una mezcla de tabulaciones y espacios, puede convertir todas las tabulaciones de un archivo en espacios en la mayoría de editores.

Longitud de línea

Muchos programadores de Python recomiendan que cada línea tenga menos de 80 caracteres. Históricamente, esta directriz se desarrolló porque la mayoría de los ordenadores solo admitían 79 caracteres por línea en una ventana de terminal. Actualmente, podemos trabajar con líneas mucho más largas en pantalla, pero existen otras razones para adherirse al estándar de los 79 caracteres por línea.

Los programadores profesionales suelen tener varios archivos abiertos en la misma pantalla; ceñirse a la longitud estándar les permite ver líneas enteras en dos o tres archivos colocados en paralelo en una pantalla. La PEP 8 también recomienda limitar todos los comentarios a 72 caracteres por línea porque algunas herramientas que generan documentación automática para proyectos largos añaden caracteres de formato al principio de las líneas comentadas.

Las directrices de la PEP 8 para la longitud de línea no son un requisito inamovible y algunos equipos prefieren un límite de 99 caracteres. No se preocupe demasiado por la longitud de las líneas en su aprendizaje, pero tenga en cuenta que sus colaboradores casi

siempre seguirán las directrices de la PEP 8. La mayoría de los editores permiten configurar una pista visual, por lo general una línea vertical en la pantalla, que muestra dónde están los límites.

Nota: El apéndice B explica cómo configurar el editor de texto para que siempre inserte cuatro espacios al pulsar el tabulador y muestre una línea vertical que nos ayude a ceñirnos al límite de 79 caracteres.

Líneas en blanco

Use líneas en blanco para agrupar partes de su programa visualmente. Conviene utilizarlas para organizar los archivos, pero sin pasarse. Si sigue los ejemplos de este libro, encontrará el equilibrio adecuado. Por ejemplo, si tiene cinco líneas de código que crean una lista y otras tres para hacer algo con esa lista, sería adecuado dejar una línea en blanco entre ambas secciones. Sin embargo, no debería colocar tres o cuatro líneas en blanco entre las dos secciones.

Las líneas en blanco no afectan a la ejecución del código, pero sí a su legibilidad. Los intérpretes de Python usan los sangrados horizontales para interpretar el significado del código y no hacen caso a los espacios verticales.

Otras directrices de estilo

La PEP 8 tiene muchas recomendaciones de estilo adicionales, pero la mayoría se refieren a programas más complejos que los que estamos escribiendo de momento. A medida que veamos estructuras de Python más complejas, comentaré las directrices PEP 8 relevantes.

PRUÉBELO

- **4-14. PEP 8:** Eche un vistazo a la guía de estilo original (en inglés) de la PEP 8, en <https://python.org/dev/peps/pep-0008/>. Todavía no le dará un gran uso, pero no está de más echarle un vistazo.
- **4-15. Revisión de código:** Elija tres de los programas que ha escrito en este capítulo y modifíquelos para ajustarlos a la PEP 8:
 - Use cuatro espacios por cada nivel de sangrado. Configure el editor de texto para insertar cuatro espacios cada vez que pulse **Tab**, si es que no lo ha hecho ya (consulte el apéndice B si necesita instrucciones).
 - Use menos de 80 caracteres en cada línea y configure su editor para que muestre una guía vertical en la posición del 80º carácter.
 - No abuse de las líneas en blanco en sus archivos de programa.

Resumen

En este capítulo ha aprendido a trabajar eficientemente con los elementos de una lista. Ya sabe cómo pasar por una lista con un bucle `for`, cómo Python utiliza los sangrados para estructurar un programa y cómo evitar algunos errores de sangrado habituales. Hemos visto cómo hacer listas numéricas sencillas, además de unas cuantas operaciones que podemos realizar con ellas. Ha aprendido a partir una lista para trabajar con un subconjunto de elementos y a copiar listas correctamente usando un trozo. También conoce ya las tuplas, que proporcionan cierto grado de protección a un conjunto de valores que no deben cambiar, y sabe cómo estilizar su código para que resulte más fácil leerlo.

En el capítulo 5, veremos cómo responder adecuadamente a distintas condiciones utilizando sentencias `if`. Aprenderá a encadenar conjuntos de pruebas condicionales relativamente complejos para responder adecuadamente al tipo exacto de situación o información que busca. También descubriremos cómo usar sentencias `if` pasando en bucle por una lista para realizar acciones específicas con los elementos seleccionados.

5

SENTENCIAS IF



Con frecuencia, programar requiere examinar una serie de condiciones y decidir qué acción llevar a cabo basándose en esas condiciones. La sentencia `if` de Python permite analizar el estado actual de un programa y responder adecuadamente a dicho estado.

En este capítulo, veremos cómo escribir pruebas condicionales que nos permitirán comprobar cualquier condición de interés. También descubriremos cómo escribir sentencias `if` sencillas y cómo crear una serie más compleja de sentencias `if` para identificar cuándo se dan las condiciones exactas que queremos. Después, aplicaremos este concepto a las listas para escribir un bucle `for` que maneje la mayoría de los elementos de una lista de una forma y ciertos elementos con unos valores específicos de otra.

Un ejemplo sencillo

El siguiente ejemplo muestra cómo las pruebas `if` permiten responder correctamente a situaciones especiales. Imagine que tiene una lista de coches y quiere imprimir el nombre de cada uno. Se trata de nombres propios, así que la mayoría deberían imprimirse con mayúscula inicial. Sin embargo, el valor '`bmw`' debería imprimirse todo en mayúsculas. El siguiente código pasa en bucle por una lista de marcas de coche y busca el valor '`bmw`'. Siempre que el valor sea '`bmw`', se imprimirá con todas las letras mayúsculas en vez de solo la inicial:

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

for car in cars:
❶    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

El bucle de este ejemplo comprueba primero si el valor actual de `car` es `'bmw'` ❶. Si lo es, lo imprime en mayúsculas. Si es cualquier valor distinto de `'bmw'`, se imprime con mayúscula inicial solo:

```
Audi
BMW
Subaru
Toyota
```

Este ejemplo combina varios conceptos que veremos en este capítulo. Para empezar, nos centraremos en los tipos de pruebas que podemos usar para examinar las condiciones de un programa.

Pruebas condicionales

En el núcleo de toda sentencia `if` hay una expresión que puede evaluarse como verdadera (`True`) o falsa (`False`); es lo que llamamos una prueba condicional. Python usa los valores `True` y `False` para decidir si debería ejecutar o no el código de una sentencia `if`. Si una prueba condicional se evalúa como `True`, Python ejecutará el código que sigue a la sentencia `if`; si la prueba se evalúa como `False`, ignorará ese código.

Comprobar la igualdad

La mayoría de las pruebas condicionales comparan el valor actual de una variable con un valor específico de interés. La prueba condicional más sencilla comprueba si el valor de una variable es igual que el valor de interés:

```
>>> car = 'bmw'  
>>> car == 'bmw'  
True
```

La primera línea establece el valor de `car` como `'bmw'` usando un solo signo de igualdad, como ya hemos visto muchas veces. La línea siguiente comprueba si el valor de `car` es `'bmw'` usando un signo de igualdad doble (`==`). Este operador de igualdad devuelve `True` si los valores a la derecha y a la izquierda del operador coinciden y `False` si no lo hacen. Los valores de este ejemplo coinciden, así que Python devuelve `True`.

Cuando el valor de `car` es cualquier nombre distinto de `'bmw'`, esta prueba devuelve `False`:

```
>>> car = 'audi'  
>>> car == 'bmw'  
False
```

Un signo de igualdad simple es en realidad una sentencia; podríamos leer la primera línea de código como "Establecer el valor de `car` como igual a `'audi'`". Por otro lado, un doble signo de igualdad formula la pregunta: "¿Es el valor de `car` igual a `'bmw'`?". La mayoría de los lenguajes de programación usan así los signos de igualdad.

Ignorar mayúsculas y minúsculas al comprobar la igualdad

En Python, la comprobación de igualdad distingue entre mayúsculas y minúsculas, así que dos valores que las usen de distinta forma no se consideran iguales:

```
>>> car = 'Audi'  
>>> car == 'audi'  
False
```

Si el uso de mayúsculas y minúsculas es importante, este comportamiento es una ventaja. Pero, si no es relevante y solo queremos comprobar el valor de una variable, podemos convertir el valor a minúsculas antes de hacer la comparación:

```
>>> car = 'Audi'  
>>> car.lower() == 'audi'  
True
```

Esta prueba devolvería `True` independientemente de cómo esté escrito '`Audi`', ya que ahora no distingue entre mayúsculas y minúsculas. La función `lower()` no cambia el valor guardado originalmente en `car`, así que se puede hacer este tipo de comparación sin afectar a la variable original:

```
>>> car = 'Audi'  
>>> car.lower() == 'audi'  
True  
>>> car  
'Audi'
```

En primer lugar, asignamos la cadena con mayúscula inicial '`Audi`' a la variable `car`. A continuación, convertimos el valor de `car` a minúsculas y lo comparamos con la cadena '`audi`'. Las dos cadenas coinciden, así que Python devuelve `True`. Vemos que el valor almacenado en `car` no se ha visto afectado por el método `lower()`.

Los sitios web usan ciertas reglas similares a esto para los datos que introducen los usuarios. Por ejemplo, un sitio podría usar una prueba condicional como esta para asegurarse de que cada usuario tenga realmente un nombre de usuario único, que no sea solo una variación en la grafía del nombre de otra persona. Cuando alguien

envía un nombre, ese nuevo nombre de usuario se convierte a minúsculas y se compara con la versión en minúsculas de todos los nombres de usuario existentes. En esta comprobación, se rechazaría un nombre de usuario como '`John`' si ya se usa una variación de '`john`'.

Comprobar la desigualdad

Cuando queremos determinar si dos valores no son iguales, podemos combinar un signo de exclamación con uno de igualdad (`!=`). El signo de exclamación significa "no", como ocurre en muchos lenguajes de programación.

Vamos a usar otra sentencia `if` para analizar el funcionamiento del operador de desigualdad. Guardaremos los ingredientes de una pizza en una variable e imprimiremos un mensaje si el cliente no pide anchoas:

`toppings.py`

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':
    print("Hold the anchovies!")
```

Este código compara el valor de `requested_topping` con el valor '`anchovies`'. Si estos dos valores no coinciden, Python devuelve `True` y ejecuta el código que sigue a la sentencia `if`. Si ambos valores coinciden, Python devuelve `False` y no ejecuta ese código.

Como el valor de `requested_topping` no es '`anchovies`', se ejecuta la función `print()`:

Hold the anchovies!

La mayoría de las expresiones condicionales que escribirá comprobarán la igualdad, pero a veces puede ser más eficiente

buscar la desigualdad.

Comparaciones numéricas

Probar valores numéricos es bastante fácil. Por ejemplo, el siguiente código comprueba si una persona tiene 18 años:

```
>>> age = 18  
>>> age == 18  
True
```

También podemos hacer una prueba para ver si dos números no son iguales. Por ejemplo, el siguiente código imprime un mensaje si la respuesta dada es incorrecta:

magic_number.py

```
answer = 17  
  
if answer != 42:  
    print("That is not the correct answer. Please try again!")
```

La prueba condicional se supera porque el valor de `answer` (17) no es igual a 42. Al pasarse la prueba, se ejecuta el bloque sangrado:

That is not the correct answer. Please try again!

También podemos incluir varias comparaciones matemáticas en nuestras sentencias condicionales, como "menor que", "menor o igual que", "mayor que" y "mayor o igual que":

```
>>> age = 19  
>>> age < 21  
True  
>>> age <= 21  
True
```

```
>>> age > 21
False
>>> age >= 21
False
```

Cada comparación matemática se puede usar como parte de una `if`, lo que puede ayudarnos a detectar condiciones de interés exactas.

Comprobar varias condiciones

Puede que necesitemos comprobar múltiples condiciones al mismo tiempo. Por ejemplo, a veces necesitamos que se den dos condiciones para que se ejecute una acción. Otras veces, nos basta con que una sola sea `True`. Las palabras clave `and` y `or` nos ayudarán en estas situaciones.

Usar `and` para comprobar varias condiciones

Para comprobar si dos condiciones son `True` al mismo tiempo, usaremos la palabra clave `and` para combinar las dos pruebas condicionales; si se pasan las dos pruebas, la expresión general se evaluará como `True`. Si no se supera alguna de las pruebas, o las dos, la expresión se evalúa como `False`. Por ejemplo, podemos comprobar si dos personas tienen más de 21 años con esta prueba:

```
>>> age_0 = 22
>>> age_1 = 18
❶  >>> age_0 >= 21 and age_1 >= 21
False
❷  >>> age_1 = 22
>>> age_0 >= 21 and age_1 >= 21
True
```

En primer lugar, definimos dos edades, `age_0` y `age_1`. A continuación, comprobamos si ambas edades son 21 o más ❶. La prueba de la izquierda se pasa, pero la de la derecha falla, así que la expresión condicional general se evalúa como `False`. A continuación, cambiamos `age_1` a 22 ❷. El valor de `age_1` es ahora mayor que 21, así que se superan las dos pruebas y la expresión condicional general se evalúa como `True`.

Para mejorar la legibilidad, podemos poner las pruebas individuales entre paréntesis, pero no es obligatorio. Si los usamos, la prueba queda así:

```
(age_0 >= 21) and (age_1 >= 21)
```

Usar `or` para comprobar varias condiciones

La palabra clave `or` nos permite comprobar varias condiciones, pero la prueba se pasa cuando se pasa una o ambas pruebas individuales. Una expresión `or` falla solo cuando no se superan las dos pruebas individuales.

Volvamos al ejemplo de las dos edades, pero ahora solo vamos a buscar a una persona con más de 21 años:

```
>>> age_0 = 22
>>> age_1 = 18
❶ >>> age_0 >= 21 or age_1 >= 21
True
❷ >>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

De nuevo, empezamos con dos variables de edad. Como la prueba para `age_0` en ❶ se supera, la expresión general se evalúa como `True`. Luego bajamos `age_0` a 18. En la prueba final ❷, no se

supera ninguna prueba individual, por lo que la expresión general se evalúa como `False`.

Comprobar si hay un valor en una lista

A veces, es importante comprobar si una lista contiene un valor determinado antes de realizar una acción. Por ejemplo, es posible que necesitemos comprobar si ya existe un nombre de usuario en una lista de usuarios actuales antes de completar el registro de otra persona en un sitio web. O, en un proyecto con mapas, puede que necesitemos comprobar si una ubicación enviada está ya en una lista de ubicaciones conocidas.

Para descubrir si un valor en particular está ya en una lista, usaremos la palabra clave `in`. Tomemos como ejemplo un código que podríamos escribir para una pizzería. Haremos una lista de ingredientes que un cliente ha pedido para su pizza y luego comprobaremos si la lista incluye determinados ingredientes.

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
>>> 'mushrooms' in requested_toppings
True
>>> 'pepperoni' in requested_toppings
False
```

La palabra clave `in` le indica a Python que compruebe la existencia de `'mushrooms'` y `'pepperoni'` en la lista `requested_toppings`. Esta técnica es bastante potente porque podemos crear una lista de valores esenciales y luego comprobar fácilmente si el valor que estamos probando coincide con uno de los valores de la lista.

Comprobar si un valor no está en una lista

Otras veces, es importante saber si un valor no aparece en una lista. En este caso, usaremos la palabra clave `not`. Por ejemplo,

piense en una lista de usuarios que tienen prohibido comentar en un foro. Podemos comprobar si un usuario está vetado antes de dejarle enviar un comentario:

banned_users.py

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

if user not in banned_users:
    print(f"{user.title()}, you can post a response if you wish.")
```

La sentencia `if` está bastante clara. Si el valor de `user` no está en la lista `banned_users`, Python devuelve `True` y ejecuta la línea sangrada.

El nombre de usuario `'marie'` no está en la lista `banned_users`, así que verá un mensaje invitándola a escribir una respuesta:

Marie, you can post a response if you wish.

Expresiones booleanas

A medida que aprenda a programar, lo más seguro es que se tope con el término expresión booleana. Una "expresión booleana" no es más que otro nombre con el que se conoce una prueba condicional. Un valor booleano puede ser `True` o `False`, igual que el valor de una expresión condicional después de ser evaluado.

Los valores booleanos suelen usarse para seguir la pista a ciertas condiciones, como si un juego se está ejecutando o si un usuario puede editar determinado contenido en un sitio web:

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

Los valores booleanos ofrecen una forma eficiente de rastrear el estado de un programa o una condición que es importante en el

programa.

PRUÉBELO

- **5-1. Pruebas condicionales:** Escriba una serie de pruebas condicionales. Imprima una frase describiendo cada prueba y su predicción para el resultado. Su código debería tener un aspecto similar a esto:

```
car = 'subaru'  
print("Is car == 'subaru'? I predict True.")  
print(car == 'subaru')  
  
print("\nIs car == 'audi'? I predict False.")  
print(car == 'audi')
```

- Examine los resultados y asegúrese de comprender por qué cada línea se evalúa como `True` o `False`.
- Cree al menos 10 pruebas y haga que 5, como mínimo, se evalúen como `True` y otras 5 como `False`.
- **5-2. Más pruebas condicionales:** No hace falta que limite el número de pruebas condicionales a diez. Si quiere probar más comparaciones, escriba más pruebas y añádalas a `conditional_tests.py`. Haga que un resultado sea `True` y otro `False` para cada una de estas pruebas:
 - Pruebas de igualdad y desigualdad con cadenas.
 - Pruebas con el método `lower()`.
 - Pruebas numéricas que impliquen igualdad y desigualdad, mayor que y menor que, mayor o igual que y menor o igual que.
 - Pruebas con las palabras clave `and` y `or`.
 - Prueba para comprobar si un elemento está en una lista.
 - Prueba para comprobar si un elemento no está en una lista.

Sentencias if

Cuando comprenda las pruebas condicionales, puede empezar a escribir sentencias `if`. Hay varios tipos de sentencias `if`. La clase de sentencia que usemos dependerá del número de condiciones que tengamos que probar. Hemos visto varios ejemplos de sentencias `if` en la explicación de las pruebas condicionales, pero vamos a profundizar en el tema.

Sentencias if simples

El tipo más simple de sentencia `if` tiene una prueba y una acción:

```
if prueba_condicional:  
    Hacer algo
```

Podemos poner cualquier prueba condicional en la primera línea y casi cualquier acción en el bloque sangrado después de la prueba. Si la prueba condicional se evalúa como `True`, Python ejecuta el código que hay después de la sentencia `if`. Si la prueba se evalúa como `False`, Python ignora esa parte del código.

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

voting.py

```
age = 19  
if age >= 18:  
    print("You are old enough to vote!")
```

En Python comprueba si el valor de `age` es mayor o igual que 18. Lo es, así que ejecuta la llamada sangrada a `print()`:

You are old enough to vote!

El sangrado tiene el mismo papel en las sentencias `if` que en los bucles. Todas las líneas sangradas después de una sentencia `if` se ejecutarán si se supera la prueba y el bloque completo se ignorará si no se supera.

Podemos poner tantas líneas de código como queramos en el bloque que sigue a la sentencia `if`. Vamos a añadir otra línea de salida si la persona tiene edad para votar, preguntándole si está censada:

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
```

Se pasa la prueba condicional. Dado que las dos llamadas a `print()` están sangradas, se imprimen las dos líneas:

```
You are old enough to vote!
Have you registered to vote yet?
```

Si el valor de `age` fuese menor que 18, este programa no generaría salida.

Sentencias if-else

Con frecuencia, querremos realizar una acción cuando se pase una prueba condicional y una acción distinta en todos los demás casos. La sintaxis `if-else` de Python lo hace posible. Un bloque `if-else` es similar a una sentencia `if simple`, pero la sentencia `else` nos permite definir una acción o un conjunto de acciones que se excluyen cuando la prueba condicional falla.

Vamos a mostrar el mismo mensaje de antes si la persona tiene edad para votar, pero ahora añadiremos un mensaje para aquellos que no puedan votar todavía:

```
age = 17
❶ if age >= 18:
    print("You are old enough to vote!")
    print("Have you registered to vote yet?")
❷ else:
    print("Sorry, you are too young to vote.")
    print("Please register to vote as soon as you turn 18!")
```

Si se pasa la prueba condicional ❶, se ejecuta el primer bloque de llamadas a `print()`. Si la prueba se evalúa como `False`, se ejecuta el bloque `else` ❷. Dado que `age` es menor que 18 en esta ocasión, la prueba condicional falla y se ejecuta el código del bloque `else`:

```
Sorry, you are too young to vote.
Please register to vote as soon as you turn 18!
```

Este código funciona porque solo tiene dos situaciones posibles para evaluar: o bien la persona tiene edad para votar o no la tiene. La estructura `if-else` funciona en situaciones en las que queremos que Python siempre ejecute una de dos acciones posibles. En una cadena `if-else` sencilla, como esta, siempre se ejecutará una de las dos acciones.

La cadena `if-elif-else`

Con frecuencia, necesitará probar más de dos situaciones posibles y para evaluarlas puede usar la sintaxis de Python `if-elif-else`. Python ejecuta solo un bloque en una cadena `if-elif-else`. Ejecuta cada prueba condicional en orden hasta que se pasa una. Cuando se supera una prueba, se ejecuta el código que va con ella y Python omite el resto de las pruebas.

Muchas situaciones del mundo real implican más de dos condiciones. Piense, por ejemplo, en un parque de atracciones que

cobra distintas tarifas en función de la edad:

- La entrada es gratuita para menores de 4 años.
- Los menores de entre 4 y 18 años pagan 25 dólares.
- A partir de 18 años la entrada cuesta 40 dólares.

¿Cómo podemos usar una sentencia `if` para determinar el precio de la entrada de una persona? El siguiente código comprueba el grupo de edad al que pertenece una persona y muestra un mensaje con el precio de la entrada:

amusement_park.py

```
age = 12

if age < 4:
    print("Your admission cost is $0.")
elif age < 18:
    print("Your admission cost is $25.")
else:
    print("Your admission cost is $40.")
```

La prueba `if` ❶ comprueba si la persona tiene menos de 4 años. Si se pasa la prueba, aparece el mensaje correspondiente y Python omite el resto de las pruebas. La línea `elif` ❷ es, en realidad, otra prueba `if`, que se ejecuta solo si falla la anterior. En este punto de la cadena, sabemos que la persona tiene como mínimo 4 años porque la primera prueba ha fallado. Si la persona tiene menos de 18, se imprimirá el mensaje correspondiente y Python omitirá el bloque `else`. Si fallan las pruebas `if` y `elif`, Python ejecuta el código del bloque `else` ❸. En este ejemplo, la prueba `if` ❶ se evalúa como `False`, así que no se ejecuta su bloque de código. Sin embargo, la segunda prueba se evalúa como `True` (12 es menor que 18), así que se ejecuta su código. La salida es una oración que informa al usuario del precio de su entrada:

Your admission cost is \$25.

Cualquier edad superior a 17 haría que las dos primeras pruebas fallasen y, en ese caso, se ejecutaría el bloque `else`, mostrando un precio de entrada de 40 dólares.

En lugar de imprimir el precio de admisión dentro del bloque `if-elif-else`, sería más conciso establecer solo el precio dentro de la cadena `if-elif-else` y usar una única llamada a `print()` que se ejecute después de que la cadena se evalúe:

```
age = 12

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

print(f"Your admission cost is ${price}.")
```

Las líneas sangradas establecen el valor de `price` de acuerdo con la edad de la persona, como en el ejemplo anterior. Una vez determinado el precio por la cadena `if-elif-else`, una llamada independiente y sin sangrar a `print()` usa este valor para mostrar un mensaje informando a la persona del precio de su entrada.

Este código produce la misma salida que el ejemplo anterior, pero la finalidad de la cadena `if-elif-else` es más limitada. En lugar de determinar un precio y mostrar un mensaje, solo determina el precio de la entrada. Además de ser más eficiente, este código revisado es más fácil de modificar que el del enfoque original. Para cambiar el texto del mensaje de salida, solo habría que cambiar una llamada a `print()` y no tres independientes.

Utilizar múltiples bloques elif

Podemos usar todos los bloques `elif` que queramos en nuestro código. Por ejemplo, si el parque de atracciones fuese a implementar un descuento para mayores, podríamos añadir una prueba adicional para determinar si se puede aplicar a alguien ese descuento. Supongamos que los mayores de 65 años o más pagan la mitad de la entrada normal, es decir, 20 dólares:

```
age = 12

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

print(f"Your admission cost is ${price}.")
```

La mayor parte de este código no cambia. El segundo bloque `elif` comprueba ahora si una persona tiene menos de 65 años antes de asignarle el precio completo de la entrada, 40 dólares. Observe que es preciso cambiar el valor asignado en el bloque `else` por 20 dólares porque las únicas personas que entrarían en este bloque tienen 65 años o más.

Omitir el bloque `else`

Python no requiere un bloque `else` al final de una cadena `if-elif`. A veces, este bloque es útil; otras veces es más claro usar una sentencia `elif` adicional que capture la condición de interés específica:

```
age = 12
```

```
if age < 4:  
    price = 0  
elif age < 18:  
    price = 25  
elif age < 65:  
    price = 40  
elif age >= 65:  
    price = 20  
  
print(f"Your admission cost is ${price}.")
```

El bloque `elif` final asigna un precio de 20 dólares cuando la persona tiene 65 años o más, lo cual añade algo más de claridad que el bloque general `else`. Con este cambio, todos los bloques de código deben pasar una prueba concreta para ejecutarse.

El bloque `else` es una sentencia multifunción. Cumple cualquier condición que no cumpla una prueba `if` o `elif` específica, y eso a veces puede incluir datos no válidos o maliciosos. Si tiene una condición final específica para probar, considere usar un último bloque `elif` y omitir el bloque `else`. Como resultado, tendrá más garantías de que su código funcionará solo en las condiciones correctas.

Probar múltiples condiciones

La cadena `if-elif-else` es potente, pero solo es apropiado usarla cuando necesitamos superar una única prueba. En el momento en que Python encuentra una prueba que pasa, omite las demás pruebas. Este comportamiento es beneficioso porque es eficiente y nos permite probar una condición específica.

Sin embargo, a veces es importante comprobar todas las condiciones de interés. En esos casos, conviene usar una serie de sentencias `if` simples sin bloques `elif` ni `else`. Esta técnica tiene sentido cuando puede darse más de una condición y queremos actuar sobre todas las condiciones evaluadas como `True`.

Volviendo al ejemplo de la pizzería, si alguien pidiese una pizza con dos ingredientes, deberíamos asegurarnos de incluirle los dos:

toppings.py

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")

❶ if 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")

if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

Comenzamos con una lista de los ingredientes solicitados. La primera sentencia `if` comprueba si el cliente ha pedido champiñones en su pizza. Si es así, se imprime un mensaje confirmando ese ingrediente. La comprobación de pepperoni ❶ es otra sentencia `if`, no una sentencia `elif` o `else`, así que la prueba se ejecuta independientemente de si se ha pasado la primera prueba o no. La última sentencia `if` comprueba si quiere un extra de queso, independientemente del resultado de las dos primeras pruebas. Estas tres pruebas independientes se completan cada vez que se ejecuta el programa.

Como todas las condiciones del ejemplo se han evaluado, se añaden champiñones y extra de queso a la pizza:

```
Adding mushrooms.
Adding extra cheese.

Finished making your pizza!
```

Este código no funcionaría bien si usásemos un bloque `if-elif-else`, ya que el código se detendría en cuanto se pasase una prueba. Este es el aspecto que tendría:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
    print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza!")
```

La prueba `'mushrooms'` es la primera que pasa, así que se añaden champiñones a la pizza. Sin embargo, los valores `'extra cheese'` y `'pepperoni'` no se comprueban nunca porque Python no ejecuta más pruebas que la primera que se supera en una cadena `if-elif-else`. Se añadirá a la pizza el primer ingrediente solicitado por el cliente, pero se perderá cualquier otro que quisiera:

Adding mushrooms.

Finished making your pizza!

En resumen, si solo quiere que se ejecute un bloque de código, use una cadena `if-elif-else`. Si necesita ejecutar más de un bloque de código, utilice una serie de sentencias `if` independientes.

PRUÉBELO

- **5-3. Colores de aliens #1:** Imagine un alien que acaba de ser derribado en un juego. Cree una variable llamada `color_alien` y asígnele como valor `'verde'`, `'amarillo'` o `'rojo'`.

- Escriba una sentencia `if` para comprobar si el color del alien es verde. Si lo es, imprima un mensaje informando al jugador de que ha ganado 5 puntos.
- Escriba una versión de este programa que pase la prueba `if` y otra que no. (La versión que no supera la prueba no tendrá salida).
- **5-4. Colores de aliens #2:** Elija un color para un alien igual que en el ejercicio 5-3 y escriba una cadena `if-else`.
 - Si el color del alien es verde, imprima un mensaje informando al jugador de que ha ganado 5 puntos por disparar al alien.
 - Si el color del extraterrestre no es verde, imprima una frase informando al jugador de que acaba de ganar 10 puntos.
 - Escriba una versión de este programa que ejecute el bloque `if` y otra que ejecute el bloque `else`.
- **5-5. Colores de aliens #3:** Convierta la cadena `if-else` del ejercicio 5-4 en una cadena `if-elif-else`.
 - Si el alien es verde, imprima un mensaje diciendo al jugador que ha ganado 5 puntos.
 - Si el alien es amarillo, imprima un mensaje diciendo al jugador que ha ganado 10 puntos.
 - Si el alien es rojo, imprima un mensaje diciendo al jugador que ha ganado 15 puntos.
 - Escriba tres versiones de este programa, asegurándose de que se imprime cada mensaje para el color de alien adecuado.
- **5-6. Etapas vitales:** Escriba una cadena `if-elif-else` para determinar la etapa vital de una persona. Atribuya un valor a la variable `edad` y:
 - Si la persona tiene menos de 2 años, imprima un mensaje diciendo que es un bebé.
 - Si la persona tiene entre 2 y 4 años, imprima un mensaje diciendo que es un niño pequeño.
 - Si la persona tiene como mínimo 4 años, pero menos de 13, imprima un mensaje diciendo que es un niño.
 - Si la persona tiene como mínimo 13 años, pero menos de 20, imprima un mensaje diciendo que es un adolescente.

- Si la persona tiene al menos 20 años, pero no llega a 65, imprima un mensaje diciendo que es un adulto.
- Si la persona tiene 65 años o más, imprima un mensaje diciendo que tiene más de 65 años.
- **5-7. Fruta favorita:** Haga una lista de sus frutas favoritas y escriba una serie de sentencias `if` independientes que comprueben ciertas frutas en su lista.
 - Haga una lista de sus frutas favoritas y llámela `frutas_favoritas`.
 - Escriba cinco sentencias `if`. Cada una debería comprobar si una fruta concreta está en su lista. Si lo está, el bloque `if` debería imprimir un mensaje como "¡Pues sí que te gustan los plátanos!".

Utilizar sentencias `if` con listas

Podemos hacer un trabajo interesante combinando listas y sentencias `if`. Por ejemplo, podemos detectar valores especiales que requieren un tratamiento distinto al resto de valores de la lista. También podemos gestionar eficazmente condiciones cambiantes, como la disponibilidad de algunos elementos en un restaurante durante un turno, o empezar a demostrar que nuestro código funciona como queremos en todas las situaciones posibles.

Detectar elementos especiales

Este capítulo empezó con un sencillo ejemplo que mostraba cómo manejar un valor especial '`'bmw'`', que tenía que imprimirse en un formato diferente al del resto de los valores de la lista. Ahora que tiene unos conocimientos básicos de las pruebas condicionales y las sentencias `if`, vamos a concentrarnos en cómo puede detectar valores especiales en una lista para manejarlos adecuadamente.

Vamos a seguir con el ejemplo de la pizzería. La pizzería muestra un mensaje cada vez que se añade un ingrediente a una pizza mientras se está preparando. El código para esta acción puede

escribirse de una forma muy eficiente haciendo una lista de los ingredientes que ha pedido el cliente y usando un bucle para ir anunciándolos según se añaden a la pizza:

toppings.py

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

La salida es sencilla porque este código es un simple bucle `for`:

```
Adding mushrooms.
Adding green peppers.
Adding extra cheese.

Finished making your pizza!
```

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

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping == 'green peppers':
        print("Sorry, we are out of green peppers right now.")
    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

En esta ocasión, comprobamos cada elemento solicitado antes de añadirlo a la pizza. La sentencia `if` comprueba si la persona ha

pedido pimientos verdes. En ese caso, se muestra un mensaje informándole de por qué no puede ser. El bloque `else` se asegura de que todos los demás ingredientes se añadan a la pizza.

La salida muestra que se han gestionado bien todos los ingredientes.

Adding mushrooms.

Sorry, we are out of green peppers right now.

Adding extra cheese.

Finished making your pizza!

Comprobar que una lista no está vacía

Hemos dado por sentado algo con todas las listas con las que hemos trabajado hasta ahora: que contenían como mínimo un elemento. Pronto dejaremos que los usuarios suministren la información que se guarda en una lista, de manera que ya no podremos dar por sentado que la lista contiene elementos cada vez que se ejecute un bucle. En esta situación, es útil comprobar si una lista está vacía antes de ejecutar un bucle `for`.

A modo de ejemplo, vamos a comprobar si la lista de ingredientes está vacía antes de hacer la pizza. Si la lista está vacía, nos dirigiremos al usuario para asegurarnos de que quiere una pizza básica. Si la lista no está vacía, haremos la pizza como en los ejemplos anteriores:

```
requested_toppings = []

if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Adding {requested_topping}.")
    print("\nFinished making your pizza!")

else:
    print("Are you sure you want a plain pizza?")
```

En esta ocasión comenzaremos con una lista de ingredientes vacía. En lugar de saltar al bucle `for`, hacemos una comprobación rápida. Cuando se usa el nombre de una lista en una sentencia `if`, Python devuelve `True` si la lista contiene al menos un elemento; una lista vacía se evaluará como `False`. Si `requested_toppings` pasa la prueba condicional, ejecutaremos el mismo bucle `for` que hemos usado en el ejemplo anterior. Si la prueba condicional falla, imprimiremos un mensaje preguntando al usuario si realmente quiere una pizza básica sin ingredientes.

La lista está vacía en este caso, así que la salida es la pregunta para el usuario:

Are you sure you want a plain pizza?

Si la lista no está vacía, la salida mostrará todos los ingredientes solicitados añadidos a la pizza.

Usar múltiples listas

La gente pide de todo, sobre todo en lo que a ingredientes de pizza se refiere. ¿Y si un cliente quiere patatas fritas en su pizza? Podemos usar listas y sentencias `if` para asegurarnos de que la entrada tenga sentido antes de actuar sobre ella.

Hay que tener cuidado con las peticiones de ingredientes raras antes de preparar una pizza. El siguiente código define dos listas. La primera es una lista de los ingredientes disponibles en la pizzería y la segunda es una lista de los que ha pedido el cliente. En esta ocasión, cada elemento de `requested_toppings` se comprueba con la lista de ingredientes disponibles antes de añadirse a la pizza:

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

❶ `requested_toppings = ['mushrooms', 'french fries', 'extra cheese']`

```
for requested_topping in requested_toppings:  
    ❷     if requested_topping in available_toppings:  
            print(f"Adding {requested_topping}.")  
    ❸     else:  
            print(f"Sorry, we don't have {requested_topping}.")  
  
print("\nFinished making your pizza!")
```

En primer lugar, definimos una lista de ingredientes disponibles en esta pizzería. Observe que podría tratarse de una tupla si la pizzería tuviese una selección de ingredientes estable. A continuación, hacemos una lista de los ingredientes que ha pedido el cliente. Fíjese en la extraña petición de patatas fritas (`'french fries'`) como *topping* en este ejemplo ❶. A continuación, pasamos en bucle por la lista de ingredientes solicitados. Dentro del bucle, primero comprobamos si cada ingrediente solicitado está en la lista de ingredientes disponibles ❷. Si está, lo añadimos a la pizza. Si el ingrediente solicitado no está en la lista de ingredientes disponibles, se ejecutará el bloque `else` ❸, que imprime un mensaje informando al usuario de los ingredientes que no están disponibles.

Esta sintaxis produce una salida limpia e informativa:

```
Adding mushrooms.  
Sorry, we don't have french fries.  
Adding extra cheese.  
  
Finished making your pizza!
```

¡Con solo unas líneas de código, hemos gestionado con bastante efectividad una situación del mundo real!

PRUÉBELO

- **5-8. Hola, Admin:** Cree una lista de cinco o más nombres de usuario, incluyendo el nombre 'admin'. Imagine que está escribiendo código que imprimirá un mensaje para cada usuario cuando inicien sesión en un sitio web. Pase en bucle por la lista e imprima un saludo para cada usuario:
 - Si el nombre de usuario es 'admin', imprima un saludo especial, como "Hola, admin, ¿quieres ver un informe de estado?".
 - De lo contrario, imprima un saludo genérico, como "Hola, Juan, gracias por volver a entrar".
- **5-9. Sin usuarios:** Añada una prueba `if` al programa del ejercicio anterior (`hello_admin.py`) para asegurarse de que la lista no está vacía.
 - Si la lista está vacía, imprima el mensaje "¡Necesitamos encontrar algún usuario!".
 - Borre todos los nombres de usuario de la lista y asegúrese de que se imprime el mensaje correcto.
- **5-10. Comprobar nombres de usuario:** Haga lo siguiente para crear un programa que simule cómo los sitios web se aseguran de que todos los usuarios tienen un nombre único.
 - Cree una lista con cinco o más nombres de usuario llamada `current_users`.
 - Cree otra lista de cinco nombres de usuario llamada `new_users`. Asegúrese de que uno o dos de los nuevos nombres de usuario estén también en la lista `current_users`.
 - Pase en bucle por la lista `new_users` para ver si cada nuevo nombre de usuario está ya usado. Si lo está, imprima un mensaje diciendo al usuario que tendrá que introducir otro nombre. Si un nombre no se ha usado todavía, imprima un mensaje diciendo que el nombre de usuario está disponible.
 - Asegúrese de que su comparación no distingue entre mayúsculas y minúsculas. Si se ha usado '`Juan`', no debería aceptarse '`JUAN`'. (Para hacer esto, necesitará una copia de `current_users` con la versión en minúsculas de todos los usuarios existentes).
- **5-11. Números ordinales en inglés:** Los números ordinales indican su posición en una lista, como 1^o, 2^o, etc. En inglés, la mayoría terminan en *th*, excepto el 1, el 2 y el 3, que terminan en *st*, *nd* y *rd*, respectivamente.

- Guarde los números del 1 al 9 en una lista.
- Pase en bucle por la lista.
- Use una cadena `if-elif-else` dentro del bucle para imprimir la terminación adecuada en inglés de cada número. La salida debería ser "1st 2nd 3rd 4th 5th 6th 7th 8th 9th", y cada resultado debería aparecer en una línea separada.

Dar estilo a las sentencias if

En todos los ejemplos de este capítulo, ha visto buenos hábitos de estilo. La única recomendación de la PEP 8 sobre el estilo de las pruebas condicionales es que usemos un solo espacio entre los operadores de comparación, como `==`, `>=`, `<=`. Por ejemplo:

```
if age < 4:
```

es mejor que:

```
if age<4:
```

Este espacio no afecta a la forma en la que Python interpreta el código; solo lo hace más fácil de leer para nosotros y para otros.

PRUÉBELO

- **5-12. Dar estilo a sentencias `if`:** Revise los programas que ha escrito en este capítulo y asegúrese de que ha dado el estilo apropiado a sus pruebas condicionales.
- **5-13. Sus ideas:** Llegados a este punto, ya es un programador más capacitado que cuando empezó este libro. Ahora que tiene más idea de cómo se modelan situaciones del mundo real en programas, es posible que ya esté pensando en problemas a los que podría dar solución con sus propios programas. Tome nota de cualquier idea que se le ocurra sobre los problemas que podría interesarle resolver a medida que sus habilidades de

programación mejoran. Piense en los juegos que le gustaría desarrollar, los conjuntos de datos que quiere explorar o las aplicaciones web que le gustaría crear.

Resumen

En este capítulo, hemos aprendido a crear pruebas condicionales, que siempre se evalúan como `True` o `False`. También hemos aprendido a escribir sentencias `if` sencillas, cadenas `if-else` y cadenas `if-elif-else`. Ha comenzado a usar estas estructuras para identificar condiciones particulares que puede tener que probar para saber si esas condiciones se cumplen en sus programas. También sabe ya cómo gestionar algunos elementos de una lista de forma diferente al resto, usando con eficiencia un bucle `for`. Por último, hemos revisado las recomendaciones de estilo de Python para garantizar que los programas cada vez más complejos que escriba sigan siendo relativamente fáciles de leer y entender.

En el capítulo 6, hablaremos de los diccionarios de Python. Un diccionario es similar a una lista, pero nos permite conectar informaciones. Aprenderemos a crear diccionarios, pasar en bucle por ellos y usarlos en combinación con listas y sentencias `if`. Conocer los diccionarios le permitirá modelar una variedad aún mayor de situaciones reales.

6

DICCIONARIOS



En este capítulo aprenderá a usar los diccionarios de Python, que permiten conectar informaciones relacionadas. Descubrirá cómo acceder a la información una vez que esté en el diccionario y cómo modificarla. Dado que los diccionarios pueden almacenar una cantidad de información casi ilimitada, veremos cómo pasa en bucle por los datos que contiene. Además, aprenderá a anidar diccionarios dentro de listas, listas dentro de diccionarios e incluso diccionarios dentro de otros diccionarios.

Comprender los diccionarios le permitirá modelar con mayor precisión una amplia variedad de objetos reales. Podrá crear un diccionario que represente a una persona y guardar toda la información que quiera sobre esa persona: nombre, edad, ubicación, profesión y cualquier otro aspecto personal que pueda describir. Podrá almacenar dos clases de información que se pueda relacionar entre sí, como una lista de palabras y sus significados, una lista de nombres de personas con sus números favoritos, una lista de montañas con su altura.

Un diccionario sencillo

Piense en un juego con aliens de distintos colores que valgan distintos puntos. Este sencillo diccionario guarda información sobre un alien concreto:

alien.py

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

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

El diccionario `alien_0` almacena el color del alien y los puntos que vale. Las dos últimas líneas acceden a esa información y nos la muestran así:

```
green
5
```

Como sucede con la mayoría de conceptos nuevos en programación, el uso de diccionarios requiere práctica. Cuando lleve un tiempo trabajando con ellos, no tardará en comprobar lo eficaces que pueden ser para modelar situaciones del mundo real.

Trabajar con diccionarios

Un diccionario de Python es una colección de pares clave-valor. Cada clave se conecta a un valor y podemos usar una clave para acceder al valor asociado a la misma. El valor de una clave puede ser un número, una cadena, una lista o incluso otro diccionario. De hecho, podemos usar cualquier objeto que creemos en Python como valor en un diccionario.

En Python, un diccionario va entre llaves (`{}`), con una serie de pares clave-valor entre ellas, como hemos visto en el ejemplo anterior:

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

Un par clave-valor es un conjunto de valores asociados entre sí. Cuando damos una clave, Python devuelve el valor asociado con ella. Cada clave se conecta con su valor mediante dos puntos y varios pares clave-valor se separan entre ellos por comas. Podemos

almacenar tantos pares clave-valor como queramos en un diccionario.

El diccionario más sencillo tiene exactamente un par clave-valor, como vemos en esta versión modificada de `alien_0`:

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

Este diccionario guarda una información sobre `alien_0`: su color. La cadena `'color'` es la clave y su valor asociado es `'green'`.

Acceder a los valores de un diccionario

Para obtener el valor asociado con una clave, daremos el nombre del diccionario y colocaremos la clave entre corchetes, así:

`alien.py`

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

Esto devuelve el valor asociado con la clave `'color'` en el diccionario `alien_0`:

```
green
```

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

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

Ahora podemos acceder al color o a los puntos de `alien_0`. Si un jugador dispara al alien verde, podemos comprobar cuántos puntos gana con un código como este:

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

```
new_points = alien_0['points']
print(f"You just earned {new_points} points!")
```

Una vez definido el diccionario, extraemos el valor asociado con la clave `'points'` del diccionario. A continuación, ese valor se asigna a la variable `new_points`. La última línea imprime una frase sobre los puntos que el jugador acaba de ganar:

```
You just earned 5 points!
```

Si ejecutamos este código cada vez que derribamos un alienígena, se recuperarán los puntos.

Añadir nuevos pares clave-valor

Los diccionarios son estructuras dinámicas. Podemos añadirles nuevos pares clave-valor en cualquier momento. Para añadir un nuevo par, daríamos el nombre del diccionario seguido por la nueva clave entre corchetes junto con el nuevo valor.

Vamos a añadir dos datos al diccionario `alien_0`: las coordenadas `x` e `y` del alien, que nos ayudarán a mostrarlo en una posición determinada en la pantalla. Colocaremos al extraterrestre en el borde izquierdo de la pantalla, a 25 píxeles de la parte superior. Dado que las coordenadas en una pantalla suelen empezar en la esquina superior izquierda, colocaremos al alien en el borde izquierdo estableciendo la coordenada `x` como 0 y a 25 píxeles del borde superior estableciendo la coordenada `y` como 25, así:

alien.py

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

alien_0['x_position'] = 0
alien_0['y_position'] = 25
```

```
print(alien_0)
```

Empezamos definiendo el mismo diccionario con el que estábamos trabajando y posteriormente lo imprimimos para ver su información. A continuación, añadimos un nuevo par clave-valor al diccionario: la clave `'x_position'` con el valor `0`. Hacemos lo mismo con la clave `'y_position'`. Al imprimir el diccionario modificado, veremos los dos pares nuevos:

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

La versión final del diccionario contiene cuatro pares clave-valor. Los dos pares originales especifican el color y el valor en puntos del alien. Los otros dos especifican su posición.

Los diccionarios mantienen el orden en el que se definen. Cuando imprima un diccionario o pase en bucle por sus elementos, los encontrará en el mismo orden en el que los añadió al diccionario.

Empezar con un diccionario vacío

En ocasiones puede resultar conveniente, incluso necesario, comenzar con un diccionario vacío e ir añadiéndole elementos. Para empezar a llenar un diccionario vacío, defina uno con unas llaves vacías y luego añada cada par clave-valor en su propia línea. Por ejemplo, así es como crearíamos el diccionario `alien_0` con esta técnica:

`alien.py`

```
alien_0 = {}  
  
alien_0['color'] = 'green'  
alien_0['points'] = 5  
  
print(alien_0)
```

En primer lugar definimos un diccionario `alien_0` vacío; a continuación, le añadimos los valores de color y valor. El resultado es el diccionario que hemos usado en ejemplos anteriores:

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

Por regla general, usaremos diccionarios vacíos cuando almacenemos datos suministrados por el usuario o cuando escribamos código que genere muchos pares clave-valor automáticamente.

Modificar valores en un diccionario

Para modificar un valor en un diccionario, daremos el nombre del diccionario con la clave entre corchetes y el nuevo valor que queremos asociar a esa clave. Por ejemplo, si quisiéramos cambiar el color del alien de verde a amarillo cuando avance el juego, haríamos esto:

alien.py

```
alien_0 = {'color': 'green'}
print(f"The alien is {alien_0['color']}.")

alien_0['color'] = 'yellow'
print(f"The alien is now {alien_0['color']}.)")
```

Primero, definimos un diccionario para `alien_0` que contiene solo el color y, luego, cambiamos el valor asociado con la clave `'color'` por `'yellow'`. La salida muestra que el alien ha cambiado de verde a amarillo:

```
The alien is green.
The alien is now yellow.
```

Para ver un ejemplo más interesante, vamos a hacer un seguimiento de la posición de un alien que se puede mover a distintas velocidades. Almacenaremos un valor que represente su velocidad actual y luego lo usaremos para determinar cuánto debería moverse el alien hacia la derecha:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}
print(f"Original position: {alien_0['x_position']}")

# Mueve el alien hacia la derecha.

# Determina cuánto se mueve el alien basándose en su velocidad
actual.

❶ if alien_0['speed'] == 'slow':
    x_increment = 1
elif alien_0['speed'] == 'medium':
    x_increment = 2
else:
    # Debe ser un alien rápido.
    x_increment = 3

# La nueva posición es la antigua más el incremento.

❷ alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']})
```

Empezamos definiendo un alienígena con una posición inicial `x`, una posición `y`, y una velocidad `'medium'`. Hemos omitido los valores de color y puntos para simplificar, pero este ejemplo funcionaría igual si hubiésemos incluido esos pares. También imprimimos el valor de `x_position` para ver cuánto se desplaza el alien a la derecha.

Una cadena `if-elif-else` determina cuánto debería moverse el alien a la derecha y asigna este valor a la variable `x_increment` ❶. Si la velocidad del alienígena es `'slow'`, se mueve una unidad a la

derecha; si es `'medium'`, se mueve dos unidades a la derecha, y si es `'fast'`, se mueve tres unidades. Una vez calculado el incremento, se suma al valor de `x_position` ❷ y el resultado se guarda en la clave `x_position` del diccionario.

Dado que es un alienígena de velocidad media, su posición se desplaza dos unidades a la derecha:

```
original position: 0
```

```
New position: 2
```

Esta técnica es genial: cambiando un valor en el diccionario del alien, podemos cambiar el comportamiento general del extraterrestre. Por ejemplo, para convertir a este alien de velocidad media en uno rápido, añadiríamos esta línea:

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

El bloque `if-elif-else` asignará un valor más alto a `x_increment` la próxima vez que se ejecute el código.

Eliminar pares clave-valor

Cuando ya no necesite una información almacenada en un diccionario, puede usar la sentencia `del` para eliminar por completo un par clave-valor. Lo único que necesita `del` es el nombre del diccionario y la clave que desea borrar.

Por ejemplo, vamos a quitar la clave `'points'` del diccionario `alien_0` junto con su valor:

`alien.py`

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

❶ `del alien_0['points']`

```
print(alien_0)
```

La sentencia `del` ❶ le indica a Python que borre la clave `'points'` del diccionario `alien_0` y elimine también el valor asociado a esa clave. La salida muestra que la clave `'points'` y su valor, `5`, han sido eliminados, pero el resto del diccionario no se ve afectado:

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

Nota: Tenga en cuenta que el par clave-valor eliminado se borrará de manera permanente.

Un diccionario de objetos similares

El ejemplo anterior implicaba almacenar distintos tipos de información sobre un objeto: un alienígena en un juego. También podemos usar un diccionario para guardar un tipo de información sobre varios objetos. Por ejemplo, podríamos sondear a varias personas para preguntarles por su lenguaje de programación favorito. Un diccionario es útil para guardar los resultados de un sondeo simple como este:

favorite_languages.py

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

Como ve, hemos descompuesto un diccionario más grande en varias líneas. Cada clave es el nombre de una persona que respondió

a la encuesta y cada valor es el lenguaje elegido por esa persona. Cuando necesite más de una línea para definir un diccionario, pulse **Intro** después de la llave inicial. A continuación, sangre la línea un nivel (cuatro espacios) y escriba el primer par clave-valor, seguido de una coma. A partir de aquí, cada vez que pulse **Intro**, su editor de texto debería sangrar todos los pares clave-valor para alinearlos con el primero.

Cuando termine de definir el diccionario, añada una llave de cierre en una nueva línea detrás del último par clave-valor y sángrela un nivel para alinearla con las claves. Conviene incluir una coma también después del último par, así se quedará preparada para añadir otro par en la siguiente línea si hace falta.

Nota: La mayoría de los editores de texto incorporan alguna funcionalidad que nos ayuda a dar formato a listas y diccionarios grandes, como en este ejemplo. Existen otras formas de dar formato a diccionarios largos, así que puede encontrar formatos ligeramente diferentes en su editor o en otras fuentes.

Para usar este diccionario, sabiendo el nombre de una persona que hizo la encuesta, podemos buscar rápidamente su lenguaje favorito:

favorite_languages.py

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
❶ language = favorite_languages['sarah'].title()  
print(f"Sarah's favorite language is {language}.")
```

Para ver el lenguaje que eligió Sarah, pedimos el valor así:

```
favorite_languages['sarah']
```

Usamos esta sintaxis para sacar el lenguaje favorito de Sarah del diccionario ❶ y lo asignamos a la variable `language`. Crear una nueva variable consigue una llamada a `print()` mucho más limpia. La salida muestra el lenguaje favorito de Sarah:

```
Sarah's favorite language is C.
```

Podríamos usar la misma sintaxis con cualquier sujeto representado en el diccionario.

Usar `get()` para acceder a valores

Usar claves entre corchetes para recuperar el valor que nos interesa en un diccionario puede causar un problema: si la clave que pedimos no existe, nos dará error.

Veamos qué ocurre si pedimos el valor en puntos de un alien que no tiene ese dato configurado:

```
alien_no_points.py
```

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

El resultado es un rastreo que muestra un `KeyError`:

```
Traceback (most recent call last):  
  File "alien_no_points.py", line 2, in <module>  
    print(alien_0['points'])  
    ~~~~~^~~~~~  
KeyError: 'points'
```

Veremos más sobre cómo manejar errores como este en general en el capítulo 10. Para los diccionarios en concreto, podemos usar el método `get()` para configurar un valor predeterminado que se devuelva si la clave solicitada no existe.

El método `get()` requiere una clave como primer argumento. Como segundo argumento opcional, podemos pasar el valor que se devolverá si la clave no existe:

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

```
point_value = alien_0.get('points', 'No point value assigned.')
print(point_value)
```

Si la clave `'points'` existe en el diccionario, obtendremos el valor correspondiente. Si no existe, obtendremos el valor predeterminado. En este caso, `points` no existe, así que obtenemos un mensaje indicándonos que no hay puntos asignados en vez de un error:

```
No point value assigned.
```

Si cabe la posibilidad de que no exista la clave que busca, considere usar el método `get()` en vez de la notación de los corchetes.

Nota: Si dejamos fuera el segundo argumento en la llamada a `get()` y la clave no existe, Python devolverá el valor `None`. El valor especial `None` significa "no existe ningún valor". No es un error, sino un valor especial que indica la ausencia de un valor. Veremos más usos de `None` en el capítulo 8.

PRUÉBELO

- **6-1. Persona:** Use un diccionario para almacenar información sobre una persona que conozca. Guarde su nombre, apellido, edad y la ciudad en la que vive. Debería tener claves como `nombre`, `apellido`, `edad` y `ciudad`. Imprima toda la información almacenada en su diccionario.

- **6-2. Números favoritos:** Use un diccionario para guardar los números favoritos de distintas personas. Piense en cinco nombres y úselos como claves en su diccionario. Piense el número favorito de cada persona y guárdelo como valor en su diccionario. Imprima el nombre y el número favorito de cada persona. Para que sea más divertido, puede preguntar a sus amigos para conseguir datos reales para su programa.
- **6-3. Glosario:** Puede usar un diccionario de Python para modelar un diccionario real. Sin embargo, para evitar confusiones, lo llamaremos "glosario".
 - Piense en cinco palabras de programación que haya aprendido en los capítulos anteriores y úselas como claves en su glosario. Guarde sus significados como valores.
 - Imprima cada palabra con su significado en una salida con formato limpio. Podría imprimir la palabra seguida de dos puntos y luego la definición, o la palabra en una línea y la definición sangrada en una segunda línea. Use el carácter de nueva línea (`\n`) para insertar una línea en blanco entre cada par palabra-definición en la salida.

Pasar en bucle por un diccionario

Un diccionario de Python puede contener solo unos pocos pares clave-valor o millones de ellos. Dado que puede contener una gran cantidad de datos, Python nos deja pasar en bucle por un diccionario. Los diccionarios sirven para guardar información de distintas maneras, por lo que existen formas diferentes de pasar en bucle por ellos. Podemos pasar por todos los pares clave-valor de un diccionario, solo por las claves o solo por los valores.

Pasar en bucle por todos los pares clave-valor

Antes de explorar los distintos enfoques de los bucles, vamos a considerar un nuevo diccionario pensado para guardar información sobre un usuario de un sitio web. El siguiente diccionario guardará el nombre de usuario de una persona, su nombre y su apellido:

`user.py`

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

Puede acceder a cualquier dato de `user_0` poniendo en práctica lo que ha aprendido en este capítulo. Pero ¿y si quisieramos ver todo lo que hay en este diccionario? Para ello, podríamos pasar en bucle por él con un `for`:

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

Para escribir un bucle `for` para un diccionario, creamos nombres para las dos variables que contendrán la clave y el valor de cada par. Puede elegir el nombre que quiera para estas dos variables. Este código funcionará incluso aunque usemos abreviaturas para los nombres de las variables, como aquí:

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

La segunda mitad de la sentencia `for` incluye el nombre del diccionario seguido del método `items()`, que devuelve una secuencia de pares clave-valor. Después, el bucle `for` asigna cada uno de estos pares a las dos variables proporcionadas. En el ejemplo anterior, usamos variables para imprimir cada `key`, seguida del `value` asociado.

El "`\n`" de la primera llamada a `print()` garantiza que se insertará una línea antes de cada par clave-valor en la salida:

```
Key: username
```

```
Value: efermi
```

```
Key: first
```

```
Value: enrico
```

```
Key: last
```

```
Value: fermi
```

El paso en bucle por todos los pares clave-valor funciona especialmente bien en diccionarios como el ejemplo `favorite_languages.py`, que guarda el mismo tipo de información para varias claves diferentes. Si pasamos en bucle por el diccionario `favorite_languages`, obtendremos el nombre de cada persona incluida en el diccionario y su lenguaje de programación preferido. Como las claves siempre hacen referencia a un nombre de persona y el valor es siempre un lenguaje, usaremos las variables `name` y `language` en el bucle en vez de `key` y `value`. Así será más fácil entender lo que pasa dentro del bucle:

favorite_languages.py

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
for name, language in favorite_languages.items():  
    print(f"{name.title()}'s favorite language is {language.title()}")
```

Este código le indica a Python que pase en bucle por cada par clave-valor del diccionario. A medida que pasa por cada par, la clave se asigna a la variable `name` y el valor a la variable `language`. Estos nombres descriptivos hacen mucho más fácil ver lo que está haciendo la llamada a `print()`.

Ahora, con unas pocas líneas de código, podemos mostrar toda la información de la encuesta:

```
Jen's favorite language is Python.  
Sarah's favorite language is C.  
Edward's favorite language is Rust.  
Phil's favorite language is Python.
```

Este tipo de bucle funcionaría también si el diccionario almacenase miles o millones de personas.

Pasar en bucle por todas las claves del diccionario

El método `keys()` es útil cuando no hace falta trabajar con todos los valores de un diccionario. Vamos a pasar en bucle por el diccionario `favorite_languages` para imprimir los nombres de todos los que hicieron la encuesta:

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

Este bucle `for` le dice a Python que saque las claves del diccionario `favorite_languages` y las asigne de una en una a la variable

`name`. La salida muestra los nombres de todos los que respondieron a la encuesta:

```
Jen  
Sarah  
Edward  
Phil
```

Pasar en bucle por las claves es, en realidad, el comportamiento predeterminado al pasar en bucle por un diccionario, así que este código debería tener exactamente la misma salida que si escribiésemos:

```
for name in favorite_languages:
```

en vez de:

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

Puede usar el método `keys()` explícitamente si eso hace su código más legible, u omitirlo si lo prefiere.

Es posible acceder al valor asociado con cualquier clave que nos interese dentro del bucle usando la clave actual. Vamos a imprimir un mensaje para un par de amigos sobre el lenguaje que eligieron. Pasaremos en bucle por los nombres del diccionario como hicimos antes, pero, cuando el nombre coincida con uno de nuestros amigos, mostraremos el mensaje:

```
favorite_languages = {  
    --fragmento omitido--  
}  
  
friends = ['phil', 'sarah']  
for name in favorite_languages.keys():  
    print(f"Hi {name.title()}")
```

```
❶ if name in friends:  
❷     language = favorite_languages[name].title()  
     print(f"\t{name.title()}, I see you love {language}!")
```

En primer lugar, hacemos una lista de los amigos para los que queremos imprimir un mensaje. Dentro del bucle, imprimimos el nombre de cada persona. A continuación, comprobamos si el nombre (`name`) con el que estamos trabajando está en la lista `Friends`❶. Si es el caso, determinamos el lenguaje favorito de esa persona usando el nombre del diccionario y el valor actual de `name` como clave ❷. Entonces imprimimos un saludo especial, incluyendo una referencia a su lenguaje favorito.

Se imprime el nombre de todos, pero solo nuestros amigos reciben un mensaje especial:

```
Hi Jen.  
Hi Sarah.  
Sarah, I see you love C!  
Hi Edward.  
Hi Phil.  
Phil, I see you love Python!
```

También podemos usar el método `keys()` para averiguar si una persona en concreto hizo la encuesta. Veamos si Erin respondió:

```
favorite_languages =  
    --fragmento omitido--  
if 'erin' not in favorite_languages.keys():  
    print("Erin, please take our poll!")
```

El método `keys()` no es solo para bucles. En realidad, devuelve una secuencia de todas las claves y la sentencia `if` se limita a comprobar

si 'erin' está en esta lista. Dado que no está, se imprime un mensaje invitándole a completar la encuesta:

```
Erin, please take our poll!
```

Pasar en bucle por las claves de un diccionario en un orden particular

Pasar en bucle por un diccionario devuelve los elementos en el mismo orden en el que se introdujeron. Pero a veces nos interesa que el bucle siga un orden diferente.

Una forma de hacerlo es ordenar las claves a medida que se devuelven en el bucle `for`. Podemos usar la función `sorted()` para obtener una copia de las claves en orden:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
for name in sorted(favorite_languages.keys()):  
    print(f"{name.title()}, thank you for taking the poll.")
```

Esta sentencia `for` es como otras de la misma categoría, salvo porque hemos envuelto `dictionary.keys()` en una función `sorted()`. Esto dice a Python que haga una lista con todas las claves del diccionario y la ordene antes iniciar el bucle. La salida muestra a todos los que hicieron la encuesta, con los nombres en orden:

```
Edward, thank you for taking the poll.  
Jen, thank you for taking the poll.  
Phil, thank you for taking the poll.
```

Sarah, thank you for taking the poll.

Pasar en bucle por todos los valores de un diccionario

Si le interesan principalmente los valores que contiene un diccionario, puede usar el método `values()` para obtener una lista de valores sin claves. Por ejemplo, si solo quisieramos una lista de todos los lenguajes elegidos en nuestra encuesta sobre programación sin el nombre de la persona que eligió cada uno, haríamos esto:

```
favorite_languages = {  
    'jen': 'python',  
    'sarah': 'c',  
    'edward': 'rust',  
    'phil': 'python',  
}  
  
print("The following languages have been mentioned:")  
for language in favorite_languages.values():  
    print(language.title())
```

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

```
The following languages have been mentioned:  
Python  
C  
Rust  
Python
```

Este enfoque extrae todos los valores del diccionario sin comprobar si hay repeticiones. Esto puede funcionar con un número reducido de valores, pero, en un sondeo con muchos encuestados,

daría lugar a una lista muy repetitiva. Para ver cada lenguaje elegido sin repeticiones, podemos usar un conjunto. Un "conjunto" es una colección en la que cada elemento debe ser único:

```
favorite_languages =  
    --fragmento omitido--  
  
print("The following languages have been mentioned:")  
for language in set(favorite_languages.values()):  
    print(language.title())
```

Al meter una colección de valores con elementos duplicados en `set()`, Python identifica los elementos únicos de la colección y crea un conjunto con ellos. Aquí usamos `set()` para sacar los lenguajes únicos en `favorite_languages.values()`.

El resultado es una lista sin repeticiones de los lenguajes que han mencionado los encuestados:

```
The following languages have been mentioned:  
Python  
C  
Rust
```

A medida que aprenda a trabajar con Python, irá descubriendo características integradas del lenguaje que le ayudarán a hacer exactamente lo que quiere con sus datos.

Nota: Puede crear un conjunto directamente usando llaves y separando los elementos con comas:

```
>>> languages = {'python', 'rust', 'python', 'c'}  
>>> languages  
{'rust', 'python', 'c'}
```

Es fácil confundir conjuntos y diccionarios porque ambos usan llaves. Cuando vea llaves, pero no pares clave-valor, lo más probable es que se trate de un

conjunto. A diferencia de lo que ocurre con listas y diccionarios, los conjuntos no mantienen los elementos en un orden específico.

PRUÉBELO

- **6-4. Glosario 2:** Ahora que sabe cómo pasar en bucle por un diccionario, limpie el código del ejercicio 6-3 sustituyendo las llamadas a `print()` por un bucle que pase por las claves y valores del diccionario. Cuando sepa con seguridad que su bucle funciona, añada a su glosario cinco términos más relacionados con Python. Cuando vuelva a ejecutar el programa, estas nuevas palabras y definiciones deberían incluirse automáticamente en la salida.
- **6-5. Ríos:** Haga un diccionario con tres ríos importantes y el país por el que discurre cada uno. Un par clave-valor podría ser '`nilo': 'egipto'`'.
 - Use un bucle para imprimir una frase sobre cada río, como "El Nilo discurre por Egipto".
 - Use un bucle para imprimir el nombre de cada río incluido en el diccionario.
 - Use un bucle para imprimir el nombre de cada país incluido en el diccionario.
- **6-6. Sondeos:** Use el código de `favorite_languages.py`.
 - Haga una lista de personas que deberían hacer la encuesta sobre lenguajes preferidos. Incluya algunos nombres que estén ya en el diccionario y otros que no lo estén.
 - Pase en bucle por la lista de personas que deberían hacer la encuesta. Si ya la han hecho, deles las gracias por responder. Si todavía no la han completado, imprima un mensaje invitándoles a hacerlo

Anidación

A veces, necesitamos almacenar múltiples diccionarios en una lista o una lista de elementos como valor en un diccionario. Esto se llama "anidación". Podemos anidar diccionarios dentro de una lista,

una lista dentro de un diccionario o incluso un diccionario dentro de otro diccionario. La anidación es una característica potente, como demostrarán los siguientes ejemplos.

Una lista de diccionarios

El diccionario `alien_0` contiene distintos datos sobre un alien, pero no tiene sitio para guardar información sobre un segundo alien, y mucho menos sobre una pantalla llena de aliens. Entonces, ¿cómo podemos gestionar una flota extraterrestre? Una forma consiste en hacer una lista de aliens en la que cada uno sea un diccionario de información. Por ejemplo, el siguiente código crea una lista de tres aliens:

aliens.py

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

❶ aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(alien)
```

En primer lugar, crearemos tres diccionarios, cada uno representando un alien diferente. Almacenamos cada uno de estos diccionarios en una lista llamada `aliens` ❶. Por último, pasamos en bucle por la lista e imprimimos cada alien:

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

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

```
# Hace una lista vacía para guardar aliens.  
aliens = []  
  
# Hace 30 aliens verdes.  
❶ for alien_number in range(30):  
❷     new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}  
❸     aliens.append(new_alien)  
  
# Muestra los 5 primeros aliens.  
❹ for alien in aliens[:5]:  
    print(alien)  
    print("...")  
  
# Muestra cuántos aliens se han creado.  
print(f"Total number of aliens: {len(aliens)}")
```

Este ejemplo comienza con una lista vacía para alojar todos los aliens que se crearán. La función `range()` ❶ devuelve una serie de números que simplemente le indicará a Python cuántas veces queremos que se repita el bucle. Cada vez que se ejecuta el bucle, creamos un alien nuevo ❷ y lo añadimos a la lista `aliens` ❸. Usamos un trozo para imprimir los cinco primeros aliens ❹ y por último imprimimos la longitud de la lista para comprobar que, en efecto, hemos generado la flota completa de 30 marcianitos:

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

...

```
Total number of aliens: 30
```

Todos estos alienígenas tienen las mismas características, pero Python los considera como un objeto separado, lo que nos permite modificar cada extraterrestre individualmente.

¿Cómo podríamos trabajar con un grupo de alienígenas como este? Imagine que un aspecto de un juego hace que algunos aliens cambien de color y se muevan más rápido a medida que el juego avanza. A la hora de cambiar de color, podemos usar un bucle `for` y una sentencia `if` para el cambio. Por ejemplo, para cambiar los tres primeros aliens a amarillo, con velocidad media y un valor de 10 puntos cada uno, podríamos hacer esto:

```
# Hace una lista vacía para guardar aliens.  
aliens = []  
  
# Hace 30 aliens verdes.  
for alien_number in range (30):  
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}  
    aliens.append(new_alien)  
  
for alien in aliens[:3]:  
    if alien['color'] == 'green':  
        alien['color'] = 'yellow'  
        alien['speed'] = 'medium'  
        alien['points'] = 10  
  
# Muestra los 5 primeros aliens.  
for alien in aliens[:5]:  
    print(alien)  
print("...")
```

Como queremos modificar los tres primeros aliens, pasamos en bucle por un trozo que incluye solo esos extraterrestres. En este

punto todos son de color verde, pero no siempre será así, con lo que escribimos una sentencia `if` para asegurarnos de modificar solo los alienígenas de color verde. Si el alienígena es verde, su color cambiará a `'yellow'`, su velocidad a `'medium'` y su valor a 10 puntos, como vemos en la siguiente salida:

```
{'color': 'yellow', 'points': 10, 'speed': 'medium'}  
...
```

Podríamos expandir este bucle añadiendo un bloque `elif` que convierta los aliens amarillos en aliens rojos que se desplacen con rapidez, y con un valor de 15 puntos cada uno. Sin volver a mostrar todo el programa, ese bucle quedaría así:

```
for alien in aliens[0:3]:  
    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 habitual guardar un número de diccionarios en una lista cuando cada diccionario contiene distintos tipos de información sobre un objeto. Por ejemplo, podemos crear un diccionario para cada usuario de un sitio web, como hemos hecho en `user.py`, y guardar los diccionarios individuales en una lista llamada `users`. Todos los diccionarios de la lista deberían tener una estructura idéntica

para que podamos pasar en bucle por ella y trabajar del mismo modo con cada diccionario.

Una lista en un diccionario

En vez de colocar un diccionario dentro de una lista, a veces es útil poner una lista dentro de un diccionario. Por ejemplo, piense en cómo describir una pizza que un cliente ha pedido. Si empleáramos solo una lista, lo único que podríamos guardar en realidad sería una lista de los ingredientes de la pizza. Con un diccionario, la lista de ingredientes puede ser uno solo de los aspectos de la pizza que estamos describiendo.

En el siguiente ejemplo, vamos a guardar dos tipos de información para cada pizza: un tipo de masa y la lista de ingredientes. Esta última es un valor asociado con la clave `'toppings'`. Para usar los elementos de la lista, damos el nombre del diccionario y la clave `'toppings'`, como haríamos con cualquier valor del diccionario. En vez de recuperar un solo valor, obtenemos una lista de ingredientes:

pizza.py

```
# Guarda la información sobre una pizza que se está pidiendo.

pizza = {
    'crust': 'thick',
    'toppings': ['mushrooms', 'extra cheese'],
}

# Resume el pedido.
❶ print(f"You ordered a {pizza['crust']}-crust pizza "
       "with the following toppings:")

❷ for topping in pizza['toppings']:
    print(f"\t{topping})
```

Comenzaremos con un diccionario que alberga información sobre una pizza que alguien ha pedido. Una de las claves del diccionario es 'crust' y su valor asociado es la cadena 'thick'. La siguiente clave, 'toppings', tiene como valor una lista de todos los ingredientes solicitados. Resumimos el pedido antes de preparar la pizza ❶. Cuando tenga que romper una línea larga en una llamada a `print()`, elija un punto adecuado para partir la línea que se va a imprimir y termine la línea con unas comillas. Sangre la siguiente línea, añada unas comillas al principio y continúe con la cadena. Python combinará automáticamente todas las cadenas que encuentre dentro de los paréntesis. Para imprimir los ingredientes, escribimos un bucle `for` ❷. Para acceder a la lista de ingredientes, utilizamos la clave 'toppings' para que Python extraiga los ingredientes del diccionario.

La siguiente salida resume la pizza que planeamos preparar:

```
You ordered a thick-crust pizza with the following toppings:  
mushrooms  
extra cheese
```

Podemos anidar una lista en un diccionario siempre que queramos asociar más de un valor a una sola clave. En el ejemplo de los lenguajes de programación favoritos, si almacenásemos las respuestas de cada persona en una lista, los encuestados podrían elegir más de un lenguaje preferido. Al pasar en bucle por el diccionario, el valor asociado con cada persona sería una lista de lenguajes en vez de uno solo. Dentro del bucle `for` del diccionario, usamos otro bucle `for` para pasar por la lista de lenguajes asociados con cada persona:

`favorite_languages.py`

```
favorite_languages = {  
    'jen': ['python', 'rust'],  
    'sarah': ['c'],
```

```
'edward': ['rust', 'go'],
'phil': ['python', 'haskell'],
}

❶ for name, languages in favorite_languages.items():
    print(f"\n{name.title()}s favorite languages are:")
❷ for language in languages:
    print(f"\t{language.title()}"
```

El valor asociado con cada nombre en `favorite_languages` es ahora una lista. Observe que algunas personas tienen solo un lenguaje favorito mientras que otras tienen varios. Al pasar en bucle por el diccionario ❶, usamos el nombre de variable `languages` para alojar cada valor del diccionario porque sabemos que cada valor será una lista. Dentro del bucle principal del diccionario, usamos otro bucle `for` ❷ para pasar por la lista de lenguajes favoritos de cada persona. Ahora cada persona puede mencionar tantos lenguajes como quiera:

Jen's favorite languages are:

```
Python
Rust
```

Sarah's favorite languages are:

```
C
```

Edward's favorite languages are:

```
Rust
Go
```

Phil's favorite languages are:

```
Python
Haskell
```

Para refinar este programa, podríamos incluir una sentencia `if` al principio del bucle `for` del diccionario para ver si cada persona tiene más de un lenguaje favorito examinando el valor de `len(languages)`. Si alguien tiene más de un lenguaje, la salida debería ser igual; en cambio, si solo tiene uno, podríamos cambiar las palabras para reflejarlo. Por ejemplo, podríamos decir `Sarah's favorite language is C,` usando el singular.

Nota: No es recomendable anidar demasiadas listas y diccionarios. Si anida elementos en muchos más niveles que los ejemplos anteriores o trabaja con el código de otra persona con muchos niveles de anidación, es bastante probable que exista una forma más sencilla de resolver el problema.

Un diccionario en un diccionario

Podemos anidar un diccionario dentro de otro, pero si lo hacemos el código puede complicarse rápidamente. Por ejemplo, si tenemos varios usuarios para un sitio web, cada uno con un nombre de usuario único, podemos usar esos nombres como claves para un diccionario. Después, podemos guardar información sobre cada usuario usando un diccionario como valor asociado con su nombre de usuario. En el siguiente listado, guardamos tres datos sobre cada usuario: nombre, apellido y ubicación. Accederemos a esta información pasando en bucle por los nombres de usuario y el diccionario de información asociado con cada nombre de usuario:

`many_users.py`

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
    },
}
```

```
'mcurie': {  
    'first': 'marie',  
    'last': 'curie',  
    'location': 'paris',  
},  
}  
  
❶ for username, user_info in users.items():  
❷     print(f"\nUsername: {username}")  
❸     full_name = f"{user_info['first']} {user_info['last']}"  
     location = user_info['location']  
❹     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, el apellido y la ubicación del usuario. A continuación, pasamos en bucle por el diccionario `users` ❶. Python asigna cada clave a la variable `username` y el diccionario asociado con cada clave se asigna a la variable `user_info`. Una vez dentro del bucle del diccionario principal, imprimimos el nombre de usuario ❷.

A continuación, comenzamos a acceder al diccionario interior ❸. La variable `user_info`, que contiene el diccionario de información del usuario, tiene tres claves: `'first'`, `'last'` y `'location'`. Usamos cada clave para generar un nombre completo y una ubicación bien formateados para cada persona y luego imprimimos un resumen de lo que sabemos sobre cada usuario ❹:

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

Username: mcurie

Full name: Marie Curie

Location: Paris

Fíjese en que la estructura del diccionario de cada usuario es idéntica. Aunque Python no lo requiere así, esta estructura facilita el trabajo con diccionarios anidados. Si el diccionario de cada usuario tuviese claves diferentes, el código dentro del bucle `for` sería más complicado.

PRUÉBELO

- **6-7. Personas:** Empiece con el programa que ha escrito para el ejercicio 6-1. Cree dos diccionarios nuevos que representen a distintas personas y guarde los tres diccionarios en una lista llamada `personas`. Pase en bucle por la lista. Al hacerlo, imprima todo lo que sabe sobre cada persona.
- **6-8. Mascotas:** Cree varios diccionarios, cada uno representando una mascota diferente. En cada diccionario, incluya el tipo de animal y el nombre del dueño. Guarde estos diccionarios en una lista llamada `mascotas`. A continuación, pase en bucle por la lista y, al hacerlo, imprima todo lo que sabe sobre cada mascota.
- **6-9. Lugares favoritos:** Cree un diccionario llamado `lugares_favoritos`. Piense en tres nombres para usar como claves en el diccionario y guarde entre uno y tres lugares favoritos para cada persona. Para hacer este ejercicio un poco más interesante, pregunte a algunos amigos por sus sitios preferidos. Pase en bucle por el diccionario e imprima el nombre y el lugar favorito de cada persona.
- **6-10. Números favoritos:** Modifique el programa del ejercicio 6-2 para que cada persona pueda tener más de un número favorito. Luego, imprima el nombre de cada persona junto con su(s) número(s) favorito(s).
- **6-11. Ciudades:** Cree un diccionario llamado `ciudades`. Use los nombres de tres ciudades como claves en su diccionario. Cree un diccionario de información sobre cada ciudad e incluya el país en el que se encuentra, su población aproximada y alguna curiosidad sobre la ciudad. Las claves para

cada ciudad serían `país`, población y curiosidad. Imprima el nombre de cada ciudad y toda la información que tenga guardada sobre ella.

- **6-12. Extensiones:** Ahora estamos trabajando con ejemplos lo suficientemente complejos como para ser ampliados de distintas maneras. Elija uno de los programas de ejemplo de este capítulo y amplíelo añadiendo claves y valores nuevos, cambiando el contexto del programa o mejorando el formato de la salida.

Resumen

En este capítulo, hemos visto cómo definir un diccionario y cómo trabajar con la información que contiene. Ha aprendido a acceder y modificar los elementos individuales de un diccionario y a pasar en bucle por todos ellos. También sabe ya cómo pasar en bucle por los pares clave-valor de un diccionario, sus claves y sus valores y cómo anidar varios diccionarios en una lista, varias listas en un diccionario y un diccionario en otro diccionario.

En el próximo capítulo, descubriremos los bucles `while` y cómo aceptar entrada de los usuarios que usan nuestros programas. Será un capítulo emocionante porque aprenderá a hacer sus programas interactivos: podrán responder a la entrada del usuario.

7

ENTRADA DEL USUARIO Y BUCLES WHILE



Muchos programas se escriben para resolver un problema relacionado con el usuario final. Para ello, es necesario obtener cierta información de dicho usuario. A modo de ejemplo, supongamos que alguien quiere saber si tiene edad suficiente para votar. Si escribimos un programa para responder esa pregunta, tendremos que saber la edad del usuario antes de poder darle una respuesta. El programa tendrá que pedir al usuario que introduzca su edad; cuando tenga esta entrada, podrá compararla con la edad para votar para determinar si el usuario es lo bastante mayor e informarle del resultado.

En este capítulo, veremos cómo aceptar la información introducida por un usuario para que nuestro programa pueda trabajar con ella. Cuando el programa necesite un nombre, podremos solicitárselo al usuario. Cuando necesite una lista de nombres, podremos pedirle al usuario una serie de nombres. Para ello, necesitamos la función `input()`.

También veremos cómo hacer que los programas sigan ejecutándose mientras el usuario lo deseé, con el fin de que pueda introducir toda la información necesaria; después, el programa podrá trabajar con esa información. Usaremos el bucle `while` de Python para que los programas se mantengan en ejecución mientras sigan dándose ciertas condiciones.

Cuando pueda trabajar con entrada de usuario y controlar cuánto tiempo se ejecuta el programa, será capaz de escribir programas

completamente interactivos.

Cómo funciona la función `input()`

La función `input()` pausa el programa y espera a que el usuario introduzca texto. Una vez que Python recibe la entrada del usuario, la asigna a una variable con la que podemos trabajar. Por ejemplo, el siguiente programa pide al usuario que introduzca texto y luego le muestra ese mensaje:

parrot.py

```
message = input("Tell me something, and I will repeat it back to you: ")
print(message)
```

La función `input()` admite un argumento: las instrucciones que queremos mostrar al usuario para que sepa qué información debe introducir. En este ejemplo, cuando Python ejecuta la primera línea de código, el usuario ve las indicaciones `Tell me something, and I will repeat it back to you:` ("Dime algo y te lo repetiré:"). El programa espera mientras el usuario introduce su respuesta y sigue cuando pulsa **Intro**. La respuesta se asigna a la variable `message` y luego muestra la entrada de nuevo al usuario:

```
Tell me something, and I will repeat it back to you: Hello everyone!
Hello everyone!
```

Nota: Algunos editores de texto no ejecutan programas que requieran entrada de usuario. Podemos usarlos para escribir programas que soliciten entrada, pero tendremos que ejecutarlos desde un terminal. Consulte el apartado "Ejecutar programas de Python desde un terminal", en el capítulo 1.

Escribir indicaciones claras

Siempre que use la función `input()`, debería incluir unas indicaciones claras y fáciles de seguir que digan al usuario exactamente qué tipo de información busca. Servirá cualquier frase que le indique al usuario qué información debe introducir. Por ejemplo:

`greeter.py`

```
name = input("Please enter your name: ")  
print(f"\nHello, {name}!")
```

Añada un espacio al final de sus indicaciones (después de los dos puntos en el ejemplo anterior) para separar las instrucciones de la respuesta del usuario y para dejar claro dónde debe introducir este su texto. Por ejemplo:

```
Please enter your name: Eric  
Hello, Eric!
```

A veces, necesitará escribir unas instrucciones de más de una línea. Por ejemplo, puede que le interese explicarle al usuario por qué le pide cierta información. Puede asignar las indicaciones a una variable para pasarla a la función `input()`. Esto permite hacer instrucciones de varias líneas y escribir luego una sentencia `input()` limpia.

`greeter.py`

```
prompt = "If you share your name, we can personalize the messages you see."  
prompt += "\nWhat is your first name? "  
  
name = input(prompt)  
print(f"\nHello, {name}!")
```

Este ejemplo ilustra una forma de crear una cadena multilínea. La primera línea asigna la primera parte del mensaje a la variable

`prompt`. En la segunda línea, el operador `+=` coge la cadena asignada `prompt` y añade la cadena nueva al final.

Ahora las indicaciones ocupan dos líneas y, de nuevo, hay un espacio después de la pregunta para mayor claridad:

```
If you share your name, we can personalize the messages you see.
```

```
What is your first name? Eric
```

```
Hello, Eric!
```

Usar `int()` para aceptar entrada numérica

Cuando usamos la función `input()`, Python interpreta todo lo que introduce el usuario como una cadena. Observe esta sesión del intérprete, que pide la edad del usuario:

```
>>> age = input("How old are you? ")  
How old are you? 21  
>>> age  
'21'
```

El usuario escribe el número 21, pero, cuando pedimos a Python el valor de `age`, devuelve '`21`', la representación en una cadena del valor numérico introducido. Sabemos que Python ha interpretado la entrada como una cadena porque ha puesto el número entre comillas. Si solo queremos imprimir la entrada, nos vale así; pero, si intentamos usar la entrada como un número, obtendremos un error:

```
>>> age = input("How old are you? ")  
How old are you? 21  
1 >>> age >= 18  
  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
2     TypeError: '>=' not supported between instances of 'str' and 'int'
```

Cuando intentamos usar la entrada para una comparación numérica ❶, Python produce un error porque no puede comparar una cadena con un entero: la cadena `'21'` asignada a `age` no puede compararse con el valor numérico `18` ❷.

Podemos resolver este problema con la función `int()`, que transforma la entrada en un valor numérico. Esto permite que la comparación se ejecute con éxito:

```
>>> age = input("How old are you? ")  
How old are you? 21  
❶ >>> age = int(age)  
>>> age >= 18  
True
```

En este ejemplo, cuando escribimos `'21'` en el indicador, Python interpreta el número como una cadena, pero `int()` lo convierte en una representación numérica ❶. Ahora, Python puede ejecutar la prueba condicional: compara `age` (que ahora es el valor numérico `21`) y `18` para ver si la edad es mayor o igual que 18. Esta prueba se evalúa como `True`.

¿Cómo se usa la función `int()` en un programa real? Imagine un programa que determina si la gente es lo bastante alta para subirse a una montaña rusa:

rollercoaster.py

```
height = input("How tall are you, in inches? ")  
height = int(height)  
  
if height >= 48:  
    print("\nYou're tall enough to ride!")  
else:  
    print("\nYou'll be able to ride when you're a little older.")
```

El programa puede comparar `height` con `48` porque `height = int(height)` convierte el valor de la entrada en una representación numérica antes de la comparación. Si el número introducido es mayor o igual que `48`, le diremos al usuario que es lo bastante alto:

```
How tall are you, in inches? 71
```

```
You're tall enough to ride!
```

Cuando use entrada numérica para hacer cálculos y comparaciones, asegúrese de convertir primero el valor de la entrada en una representación numérica.

El operador módulo

El operador módulo (`%`) es una herramienta útil para trabajar con valores numéricos. Lo que hace es dividir un número entre otro y devolver el resto:

```
>>> 4 % 3  
1  
>>> 5 % 3  
2  
>>> 6 % 3  
0  
>>> 7 % 3  
1
```

Este operador no nos da el resultado de la división de los dos números; únicamente nos dice cuál es el resto.

Cuando un número es divisible por otro, el resto es `0`, así que el operador módulo siempre devuelve `0`. Podemos usar esto para determinar si un número es par o impar:

```
even_or_odd.py
```

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")
number = int(number)

if number % 2 == 0:
    print(f"\nThe number {number} is even.")
else:
    print(f"\nThe number {number} is odd.")
```

Los números pares son siempre divisibles entre dos, así que, si el módulo de un número y dos es cero (aquí, `if number % 2 == 0`), el número es par. De lo contrario, es impar.

Enter a number, and I'll tell you if it's even or odd: **42**

The number 42 is even.

PRUÉBELO

- **7-1. Coche de alquiler:** Escriba un programa que pregunte al usuario qué tipo de coche desea alquilar. Imprima un mensaje sobre el coche, como "Veamos si tenemos un Subaru para usted".
- **7-2. Mesa en un restaurante:** Escriba un programa que pregunte al usuario cuántos vienen a cenar. Si la respuesta es más de ocho, imprima un mensaje diciendo al usuario que tendrán que esperar mesa. De lo contrario, dígale que su mesa está lista.
- **7-3. Múltiplos de diez:** Pida al usuario un número y luego infórmele de si ese número es múltiplo de 10 o no.

Introducción a los bucles while

El bucle `for` coge una colección de elementos y ejecuta un bloque de código una vez por cada elemento de la colección. En cambio, el bucle `while` se ejecuta mientras se cumpla una condición.

El bucle while en acción

Podemos usar un bucle `while` para contar una serie de números. Por ejemplo, el siguiente bucle cuenta de 1 a 5:

counting.py

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

En la primera línea empezamos a contar en 1 asignando a `current_number` el valor `1`. Luego configuramos el bucle `while` para que siga ejecutándose mientras el valor de `current_number` sea menor o igual que `5`. El código dentro del bucle imprime el valor de `current_number` y añade 1 a ese valor con `current_number += 1`. (El operador `+=` es una abreviación de `current_number = current_number + 1`).

Python repetirá el bucle mientras se cumpla la condición `current_number <= 5`. Como 1 es menor que 5, Python imprime `1` y suma 1, convirtiendo el número actual en `2`. Como 2 es menor que 5, Python imprime `2` y vuelve a sumar 1, convirtiendo el número actual en `3`, y así sucesivamente. Cuando el valor de `current_number` es mayor que 5, el bucle deja de ejecutarse y finaliza el programa:

1
2
3
4
5

Es muy probable que los programas que usamos a diario contengan bucles `while`. Por ejemplo, un juego necesita uno para seguir ejecutándose mientras queramos seguir jugando y para detenerse cuando queramos salir. No tendría gracia que un

programa se detuviese antes de que se lo ordenemos, ni que siga ejecutándose después de que le digamos que pare, así que los bucles `while` son bastante útiles.

Dejar que el usuario elija cuándo salir

Podemos hacer que el programa `parrot.py` se ejecute mientras el usuario quiera poniendo la mayor parte del programa dentro de un bucle `while`. Definiremos un valor para salir y haremos que el programa se ejecute hasta que el usuario introduzca ese valor:

`parrot.py`

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "

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

En primer lugar definimos unas instrucciones que indiquen al usuario sus dos opciones: introducir un mensaje o introducir el valor para salir (en este caso, `'quit'`). A continuación, configuramos una variable `message` para hacer un seguimiento del valor introducido por el usuario. Definimos `message` como una cadena vacía, `""`, así Python tiene algo que comprobar la primera vez que llegue a la línea `while`. La primera vez que el programa se ejecuta, cuando Python llegue a la sentencia `while`, necesita comparar el valor de `message` con `'quit'`, pero todavía no hay entrada del usuario. Si Python no tiene nada para comparar, no podrá seguir ejecutando el programa. Para solucionar este problema, nos aseguraremos de dar a `message` un valor inicial. Aunque es solo una cadena vacía, para Python tiene sentido y le permite hacer la comparación que hace que funcione el

bucle `while`. Este bucle se ejecutará mientras el valor de `message` no sea '`quit`'.

En el primer paso por el bucle, `message` es solo una cadena vacía, así que Python entra en el bucle. En `message = input(prompt)`, Python muestra las instrucciones y espera a que el usuario escriba algo. Lo que introduzca se asignará a `message` y se imprimirá; luego, Python reevaluará la condición de la sentencia `while`. Siempre y cuando el usuario no haya escrito la palabra '`quit`', se volverán a mostrar las instrucciones y Python esperará otra entrada. Cuando por fin el usuario introduzca '`quit`', Python detendrá la ejecución del bucle y el programa terminará:

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello everyone!  
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello again.  
Hello again.
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. quit  
quit
```

Este programa funciona bien, pero imprime la palabra '`quit`' como si fuese un mensaje real. Podemos arreglarlo con una simple prueba `if`:

```
prompt = "\nTell me something, and I will repeat it back to you:"  
prompt += "\nEnter 'quit' to end the program. "  
  
message = ""  
while message != 'quit':  
    message = input(prompt)  
  
    if message != 'quit':
```

```
print(message)
```

Ahora el programa hace una comprobación rápida antes de mostrar el mensaje y solo lo imprime si no coincide con el valor para salir:

```
Tell me something, and I will repeat it back to you:
```

```
Enter 'quit' to end the program. Hello everyone!
```

```
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:
```

```
Enter 'quit' to end the program. Hello again.
```

```
Hello again.
```

```
Tell me something, and I will repeat it back to you:
```

```
Enter 'quit' to end the program. quit
```

Usar una bandera

En el ejemplo anterior, hemos hecho que el programa realice varias tareas mientras se cumpliese una condición. Pero ¿qué pasa con programas más complejos en los que distintos eventos pueden hacer que el programa deje de ejecutarse?

Por ejemplo, en un juego son varios los eventos que pueden hacer que este finalice. Cuando el jugador se queda sin vidas, o sin tiempo, o se han destruido las ciudades que tenía que proteger, el juego debería terminar. Si sucede cualquiera de esos eventos, el juego debe terminar. Si pueden producirse distintos eventos que detengan el programa, intentar probar todas esas condiciones en una sentencia `while` es complicado y difícil.

Para un programa que debería ejecutarse mientras se cumplen varias condiciones, podemos definir una variable que determine si todo el programa está activo o no. Esta variable, llamada "bandera", actúa como una señal para el programa. Podemos escribir el programa para que se ejecute mientras la bandera sea `True` y se

detenga cuando cualquiera de varios eventos cambie el valor de la bandera a `False`. Como resultado, nuestra sentencia `while` necesita comprobar solo una condición: si la bandera está ahora en `True`. Entonces, todas las demás pruebas (para ver si se ha producido un evento que debería poner la bandera en `False`) se pueden organizar bien en el resto del programa.

Vamos a añadir una bandera a `parrot.py`, del apartado anterior. Esta bandera, que llamaremos `active` (aunque puede ponerle cualquier nombre), vigilará si el programa debería seguir ejecutándose o no:

```
prompt = "\nTell me something, and I will repeat it back to you:"  
prompt += "\nEnter 'quit' to end the program. "  
  
active = True  
❶ while active:  
  
    message = input(prompt)  
  
    if message == 'quit':  
        active = False  
    else:  
        print(message)
```

Configuramos la variable `active` como `True` para que el programa se inicie en estado activo. Esto simplifica la sentencia `while` porque no se hace ninguna comparación en la propia sentencia; otras dos partes del programa se ocupan de la lógica. Mientras la variable `active` siga siendo `True`, el bucle seguirá ejecutándose ❶.

En la sentencia `if` dentro del bucle `while`, comprobamos el valor de `message` cuando el usuario introduce su entrada. Si el usuario escribe '`quit`', configuramos `active` como `False` y se detiene el bucle `while`. Si el usuario escribe cualquier otra cosa, imprimimos su entrada como un mensaje.

Este programa tiene la misma salida que el ejemplo anterior, donde hemos puesto la prueba condicional directamente en la sentencia `while`. Pero, ahora que tenemos una bandera que indica si el programa general está en estado activo, sería más fácil añadir más pruebas (como sentencias `elif`) para eventos que deberían hacer que `active` fuese `False`. Esto resulta útil en programas complicados como los juegos, en los que puede haber varios eventos que hagan que el programa deje de ejecutarse. Cuando cualquiera de estos eventos ponga la bandera `active` en `False`, el bucle del juego principal terminará; entonces se puede mostrar un mensaje de *Game Over* y dar al jugador la opción de volver a jugar.

Usar `break` para salir de un bucle

Para salir de un bucle `while` inmediatamente sin ejecutar ningún código dentro del bucle, independientemente de los resultados de cualquier prueba condicional, podemos usar la sentencia `break`. Lo que hace esta sentencia es dirigir el flujo del programa; podemos usarla para controlar qué líneas de código se ejecutan y cuáles no para que el programa ejecute solo el código que queramos cuando queramos.

Pongamos como ejemplo un programa que pregunte al usuario por lugares que ha visitado. Podemos detener el bucle `while` de este programa llamando a `break` en cuanto el usuario introduzca el valor `'quit'`:

`cities.py`

```
prompt = "\nPlease enter the name of a city you have visited:  
prompt += "\n(Enter 'quit' when you are finished.) "  
  
❶ while True:  
    city = input(prompt)  
  
    if city == 'quit':
```

```
        break  
  
    else:  
  
        print(f"I'd love to go to {city.title()}!")
```

Un bucle que empieza con `while True` ❶ se ejecutará eternamente, a menos que llegue a una sentencia `break`. El bucle de este programa sigue pidiendo al usuario que introduzca los nombres de ciudades en las que ha estado hasta que escriba `'quit'`. Cuando lo haga, la sentencia `break` se ejecutará haciendo que Python salga del bucle:

```
Please enter the name of a city you have visited:
```

```
(Enter 'quit' when you are finished.) New York
```

```
I'd love to go to New York!
```

```
Please enter the name of a city you have visited:
```

```
(Enter 'quit' when you are finished.) San Francisco
```

```
I'd love to go to San Francisco!
```

```
Please enter the name of a city you have visited:
```

```
(Enter 'quit' when you are finished.) quit
```

Nota: Puede usar la sentencia `break` en cualquier bucle de Python. Por ejemplo, podría usarla para salir de un bucle `for` que trabaja con una lista o diccionario.

Usar `continue` en un bucle

En lugar de interrumpir un bucle por completo sin ejecutar el resto de su código, podemos usar la sentencia `continue` para volver al principio del bucle en función del resultado de una prueba condicional. Por ejemplo, piense en un bucle que cuente del 1 al 10, pero imprima solo los números impares en ese rango:

counting.py

```
current_number = 0

while current_number < 10:

    current_number += 1

❶    if current_number % 2 == 0:

        continue

    print(current_number)
```

Primero, configuramos `current_number` como 0. Puesto que es menor que 10, Python entra en el bucle `while`. Una vez dentro, incrementamos la cuenta en 1 ❶, de modo que `current_number` es 1. La sentencia `if` comprueba entonces el módulo de `current_number` y 2. Si es 0 (lo que significa que `current_number` es divisible entre 2), la sentencia `continue` dice a Python que ignore el resto del bucle y vuelve al principio. Si el número actual no es divisible entre 2, se ejecuta el resto del bucle y Python imprime el número actual:

1
3
5
7
9

Evitar bucles infinitos

Todo bucle `while` necesita una forma de parar para no seguir ejecutándose eternamente. Por ejemplo, este bucle debería contar del 1 al 5:

counting.py

```
x = 1

while x <= 5:
```

```
print(x)
x += 1
```

Sin embargo, si omitimos sin querer la línea `x += 1` (como se muestra aquí), el bucle no dejará de ejecutarse:

```
# ¡Este bucle se ejecuta eternamente!
x = 1
while x <= 5:
    print(x)
```

Ahora el valor de `x` empezará en `1`, pero nunca cambiará. En consecuencia, la prueba condicional `x <= 5` se evaluará siempre como `True` y el bucle `while` se ejecutará eternamente, imprimiendo una serie de números `1`:

```
1
1
1
1
--fragmento omitido--
```

Todos los programadores escriben de vez en cuando por accidente un bucle `while` infinito, sobre todo cuando los bucles de un programa tienen condiciones de salida sutiles. Si su programa se queda atascado en un bucle infinito, pulse **Control-C** o cierre la ventana del terminal que muestra la salida del programa.

Para evitar escribir bucles infinitos, pruebe todos los bucles `while` y asegúrese de que se detienen cuando espera que lo hagan. Si quiere que un programa termine cuando el usuario introduzca un valor para salir concreto, ejecútelo e introduzca ese valor. Si el programa no se detiene, analice la forma en la que gestiona el valor que debería haber provocado el final del bucle. Asegúrese de que al menos una parte del programa puede hacer que la condición del bucle sea `False` o haga que llegue a una sentencia `break`.

Nota: Como muchos otros editores, VS Code muestra una ventana de salida incrustada que hace que sea difícil detener un bucle infinito. Para cancelar el bucle, haga clic en el área de salida del editor antes de pulsar **Control-C**.

PRUÉBELO

- **7-4. Ingredientes de pizza:** Escriba un bucle que pida al usuario que introduzca una serie de ingredientes de pizza y termine escribiendo el valor 'quit'. Cuando introduzca cada ingrediente, imprima un mensaje diciendo que lo añadirá a su pizza.
- **7-5. Entradas de cine:** Un cine cobra las entradas a distinto precio en función de la edad del espectador. Los menores de 3 años entran gratis, los niños de entre 3 y 12 años pagan 8 euros y la entrada para mayores de 12 años cuesta 12 euros. Escriba un bucle para preguntar a los usuarios su edad y luego dígales el precio de su entrada.
- **7-6. Tres salidas:** Escriba distintas versiones de los ejercicios 7-4 o 7-5, haciendo cada uno de los siguientes como mínimo una vez:
 - Use una prueba condicional en la sentencia `while` para detener el bucle.
 - Use una variable `active` para controlar cuánto tiempo se ejecuta el bucle.
 - Use una sentencia `break` para salir del bucle cuando el usuario introduzca el valor 'quit'.
- **7-7. Infinidad:** Escriba un bucle que no termine nunca y ejecútelo. (Para detenerlo, pulse **Control-C** o cierre la ventana que muestra la salida).

Usar un bucle `while` con listas y diccionarios

Hasta ahora, hemos trabajado solo con un dato del usuario cada vez. Hemos recibido la entrada del usuario y hemos imprimido la entrada o una respuesta. En el siguiente paso por el bucle `while`, recibimos otra entrada y respondemos. Pero, para manejar varios elementos y varios datos, tendremos que usar listas y diccionarios con nuestros bucles `while`.

Un bucle `for` es efectivo para pasar por una lista, pero no conviene modificar una lista dentro de un bucle `for` porque Python tendrá problemas para hacer el seguimiento de los elementos de la lista. Si hay que modificar una lista cuando se trabaja con ella, es mejor usar un bucle `while`. Emplear bucles `while` con listas y diccionarios nos permite recoger, almacenar y organizar muchas entradas para examinarlas e informar luego.

Pasar elementos de una lista a otra

Piense en una lista de usuarios recién registrados en un sitio web pero todavía sin verificar. Una vez verificados, ¿cómo podemos moverlos a una lista aparte de usuarios confirmados? Una forma sería usar un bucle `while` para sacar elementos de la lista de usuarios no confirmados a medida que los verifiquemos y añadirlos a la lista de usuarios confirmados. Así es como quedaría el código:

confirmed_users.py

```
# Empieza con usuarios que hay que verificar,  
# y una lista vacía para alojar a los usuarios confirmados.  
1 unconfirmed_users = ['alice', 'brian', 'candace']  
confirmed_users = []  
  
# Verifica cada usuario hasta que ya no hay usuarios sin confirmar.  
# Mueve a cada usuario verificado a la lista de usuarios  
# confirmados.  
2 while unconfirmed_users:  
    current_user = unconfirmed_users.pop()  
  
    print(f"Verifying user: {current_user.title()}")  
4    confirmed_users.append(current_user)  
  
# Muestra todos los usuarios confirmados.
```

```
print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

Empezamos con una lista de usuarios sin confirmar ❶ (Alice, Brian y Candace) y una lista vacía para meter a los usuarios confirmados. El bucle `while` se ejecutará mientras la lista `unconfirmed_users` no esté vacía ❷. Dentro de este bucle, el método `pop()` elimina a los usuarios sin verificar de uno en uno del final de `unconfirmed_users` ❸. Aquí, como Candace es la última de la lista `unconfirmed_users`, su nombre será el primero en eliminarse, asignarse a `current_user` y añadirse a la lista `confirmed_users` ❹. Después va Brian y, por último, Alice.

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

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice

The following users have been confirmed:
Candace
Brian
Alice
```

Eliminar todos los casos de valores específicos de una lista

En el capítulo 3, utilizamos `remove()` para eliminar un valor específico de una lista. La función `remove()` funcionaba porque el valor

que nos interesaba aparecía solo una vez en la lista, pero ¿qué pasa si queremos eliminar todas las apariciones de un valor en una lista?

Supongamos que tenemos una lista de mascotas en la que se repite varias veces el valor 'cat'. Para eliminar todos los casos de ese valor, podemos ejecutar un bucle `while` hasta que 'cat' ya no salga en la lista, como aquí:

pets.py

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

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

print(pets)
```

Empezamos con una lista en la que 'cat' aparece varias veces. Tras imprimirla, Python entra en el bucle `while` porque encuentra el valor 'cat' en la lista al menos una vez. Ya dentro del bucle, Python elimina el primer caso de 'cat', regresa a la línea `while` y vuelve a entrar en el bucle al descubrir que 'cat' sigue en la lista. Elimina cada aparición de 'cat' hasta que ese valor ya no está en la lista. Entonces, Python sale del bucle y vuelve a imprimir la lista:

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

Rellenar un diccionario con entrada del usuario

Podemos pedir a los usuarios toda la información que necesitemos en cada paso por un bucle `while`. Vamos a hacer un programa de sondeo en el que cada paso por el bucle pida el nombre del participante y una respuesta. Guardaremos los datos

recogidos en un diccionario para poder conectar cada respuesta con el correspondiente usuario:

mountain_poll.py

```
responses = {}

# Configura una bandera para indicar que la encuesta está activa.
polling_active = True

while polling_active:
    # Pide el nombre y la respuesta de la persona.
    ❶ name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb someday?
                     ")

    # Guarda la respuesta en el diccionario.
    ❷ responses[name] = response

    # Averigua si alguien más va a hacer la encuesta.
    repeat = input("Would you like to let another person respond?
                   (yes/no) ")
    if repeat == 'no':
        polling_active = False

    # La encuesta está completa. Muestra los resultados.
    print("\n--- Poll Results ---")
    ❸ for name, response in responses.items():
        print(f"{name} would like to climb {response}.")
```

El programa define primero un diccionario vacío (`responses`) y configura una bandera (`polling_active`) para indicar que la encuesta

está activa. Mientras `polling_active` sea `True`, Python ejecutará el código del bucle `while`.

Dentro del bucle, se pide al usuario que escriba su nombre y el de una montaña que le gustaría escalar ❶. Esa información se guarda en el diccionario `responses` ❷, y se pregunta al usuario si quiere que la encuesta siga ejecutándose ❸. Si escribe `yes`, el programa vuelve a entrar en el bucle `while`. Si escribe `no`, la bandera `polling_active` se pone en `False`, el bucle `while` deja de ejecutarse y el último bloque de código ❹ muestra el resultado de la encuesta.

Si ejecuta este programa introduciendo respuestas de muestra, debería ver una salida como esta:

```
What is your name? Eric
```

```
Which mountain would you like to climb someday? Denali
```

```
Would you like to let another person respond? (yes/ no) yes
```

```
What is your name? Lynn
```

```
Which mountain would you like to climb someday? Devil's Thumb
```

```
Would you like to let another person respond? (yes/ no) no
```

```
--- Poll Results ---
```

```
Lynn would like to climb Devil's Thumb.
```

```
Eric would like to climb Denali.
```

PRUÉBELO

- **7-8. Bocatería:** Haga una lista llamada `pedidos_bocadillos` y rellénela con los nombres de varios bocadillos. A continuación, haga una lista vacía llamada `bocadillos_terminados`. Pase en bucle por la lista de pedidos de bocadillos e imprima un mensaje para cada pedido, como "Su bocadillo de atún está listo". A medida que se hagan los bocadillos, páselos a la lista de terminados. Cuando todos los bocadillos estén hechos, imprima un mensaje que los enumere todos.

- **7-9. Ya no hay pastrami:** Usando la lista `pedidos_bocadillos` del ejercicio 7-8, asegúrese de que el bocadillo de 'pastrami' aparezca en la lista al menos tres veces. Añada código al principio del programa para imprimir un mensaje diciendo que no queda pastrami y use un bucle `while` para eliminar todas las apariciones de 'pastrami' en `pedidos_bocadillos`. Asegúrese de que no pasa ningún bocadillo de pastrami a la lista `bocadillos_terminados`.
- **7-10. Vacaciones de ensueño:** Escriba un programa que pregunte a los usuarios por las vacaciones de sus sueños. Escriba unas instrucciones como "Si pudieras visitar cualquier lugar del mundo, ¿dónde irías?". Incluya un bloque de código que imprima el resultado de la encuesta.

Resumen

En este capítulo ha aprendido a usar `input()` para permitir que los usuarios introduzcan información en sus programas. Ahora sabe cómo trabajar con entradas de texto y numéricas, y cómo usar bucles `while` para asegurarse de que el programa se ejecute mientras el usuario así lo desee. Hemos visto varias formas de controlar el flujo de un bucle `while`: configurando una bandera `active`, usando la sentencia `break` y usando la sentencia `continue`. Ha aprendido a utilizar un bucle `while` para pasar elementos de una lista a otra y también sabe ya cómo eliminar todas las apariciones de un valor en una lista. Además, hemos visto cómo usar bucles `while` con diccionarios.

En el capítulo 8 aprenderá a usar funciones. Las funciones nos permiten descomponer un programa en partes pequeñas, cada una con una misión específica. Podemos llamar a una función todas las veces que queramos y guardarlas en un archivo aparte. Usando funciones, podrá escribir un código más eficiente, donde sea más fácil detectar y solucionar problemas, que sea más fácil de mantener y que se pueda reutilizar en muchos programas diferentes.

8

FUNCIONES



En este capítulo aprenderá a escribir funciones, que son bloques de código con nombre diseñados para hacer una tarea específica. Para realizar una tarea concreta definida en una función, bastará con llamar a la función correspondiente. Si tiene que realizar esa tarea varias veces a lo largo del programa, no es necesario escribir todo el código para la misma tarea una y otra vez; solo con llamar a la función de esa tarea, la llamada le indicará a Python que ejecute el código de esa función. El uso de funciones hace que los programas sean más fáciles de escribir, leer, probar y corregir.

En este capítulo también aprenderá diversas maneras de pasar información a las funciones. Descubrirá cómo escribir algunas funciones cuya misión principal consiste en mostrar información y otras diseñadas para procesar datos y devolver un valor o un conjunto de valores. Por último, veremos cómo guardar las funciones en archivos separados, denominados "módulos", para tener organizados los archivos de programa principales.

Definir una función

Aquí tenemos una función sencilla, llamada `greet_user()`, que imprime un saludo:

`greeter.py`

```
def greet_user():
    """Muestra un simple saludo."""
    print("Hello!")

greet_user()
```

Este ejemplo muestra la estructura más sencilla de una función. La primera línea usa la palabra clave `def` para informar a Python de que estamos definiendo una función. Esto es la definición de la función, que le indica a Python el nombre de la función y, si procede, el tipo de información que necesitará 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 su tarea, de manera que los paréntesis quedan vacíos (aun así, es preciso incluirlos). Por último, la definición termina con dos puntos. Las líneas que aparecen sangradas debajo de `def greet_user():` conforman el cuerpo de la función. El texto de la segunda línea es un comentario conocido como "cadena de documentación", que describe lo que hace la función. Cuando Python genera documentación para las funciones de nuestros programas, busca una cadena inmediatamente después de la definición de la función. Por lo general estas cadenas van entre triples comillas, que nos permiten escribir múltiples líneas de código.

La línea `print("Hello!")` es la única línea de código real de la función, así que `greet_user()` tiene una sola tarea: `print("Hello!")`.

Cuando queramos usar esta función, tendremos que llamarla. Una llamada a una función le indica a Python que ejecute el código de dicha función. Para llamar a una función, escribimos su nombre seguido de cualquier información necesaria entre paréntesis. Como no se necesita información aquí, llamar a nuestra función es tan sencillo como escribir `greet_user()`. Como era de esperar, imprime `Hello!`:

Hello!

Pasar información a una función

Si modificamos ligeramente la función `greet_user()`, saludará al usuario por su nombre. Para que así suceda, escribimos `username` entre los paréntesis de la definición en `def greet_user()`. Al añadir aquí `username`, permitimos que la función acepte cualquier valor que especifiquemos como nombre de usuario. Ahora la función espera que proporcionemos un valor para `username` cada vez que la llamemos. Al llamar a `greet_user()`, podemos pasarle un nombre, como '`jesse`', dentro de los paréntesis:

```
def greet_user(username):  
    """Muestra un simple saludo."""  
    print(f"Hello, {username.title()}!")  
  
greet_user('jesse')
```

Introducir `greet_user('jesse')` llama a `greet_user()` y da a la función la información que necesita para ejecutar la llamada a `print()`. La función acepta el nombre que le hemos pasado y muestra el saludo para el mismo:

```
Hello, Jesse!
```

Del mismo modo, introducir `greet_user('sarah')` llama a `greet_user()`, le pasa '`sarah`' e imprime `Hello, Sarah!`. Podemos llamar a `greet_user()` con la frecuencia que queramos y pasarle cualquier nombre para producir siempre una salida predecible.

Argumentos y parámetros

En la función `greet_user()` anterior, hemos definido `greet_user()` para que requiera un valor para la variable `username`. Una vez que hemos llamado a la función y le hemos dado la información (un nombre de persona), imprime el saludo adecuado.

La variable `username` en la definición de `greet_user()` es un ejemplo de parámetro, una información que necesita la función para hacer su trabajo. El valor `'jesse'` en `greet_user('jesse')` es un ejemplo de argumento. Un argumento es una información que se pasa desde una función para llamar a una función. Cuando llamamos a la función, colocamos el valor con el que queremos que trabaje entre paréntesis. En este caso, pasamos el argumento `'jesse'` a la función `greet_user()` y asignamos el valor al parámetro `username`.

Nota: En ocasiones, los términos "argumentos" y "parámetros" se usan indistintamente. No se sorprenda si las variables de una definición de función reciben el nombre de argumento, o si las variables en una llamada a una función reciben el nombre de parámetros.

PRUÉBELO

- **8-1. Mensaje:** Escriba una función llamada `mostrar_mensaje()` que imprima una oración diciendo a todos lo que está aprendiendo en este capítulo. Llame a la función y asegúrese de que el mensaje se muestra correctamente.
- **8-2. Libro favorito:** Escriba una función llamada `libro_favorito()` que acepte un parámetro `título`. La función debería imprimir un mensaje como `"Uno de mis libros favoritos es Alicia en el País de las Maravillas"`. Llame a la función, asegurándose de incluir el título de un libro como argumento en la llamada a la función.

Pasar argumentos

Igual que la definición de una función puede tener varios parámetros, la llamada a una función puede tener varios argumentos. Podemos pasar argumentos a las funciones de distintas maneras. Podemos pasar argumentos posicionales, que tienen que estar en el mismo orden en el que se escribieron; parámetros de palabra clave, donde cada argumento consta de un nombre de

variable y un valor; y listas y diccionarios de valores. Veámoslos de uno en uno.

Argumentos posicionales

Cuando llamamos a una función, Python debe asociar cada argumento de la llamada con un parámetro de la definición. La forma más fácil de hacerlo se basa en el orden de los argumentos proporcionados. Los valores asociados así se denominan "argumentos posicionales".

Para ver cómo funcionan, consideraremos una función que muestre información sobre mascotas. La función nos dice qué tipo de animal es cada mascota y cómo se llama, como aquí:

pets.py

```
❶ def describe_pet(animal_type, pet_name):
    """Muestra información sobre una mascota."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")  
  
❷ describe_pet('hamster', 'harry')
```

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

La salida describe un hámster llamado Harry:

I have a hamster.

My hamster's name is Harry.

Múltiples llamadas a una función

Podemos llamar a una función todas las veces que haga falta. Describir una segunda mascota, requerirá otra llamada a `describe_pet()`:

```
def describe_pet(animal_type, pet_name):  
    """Muestra información sobre una mascota."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}.")  
  
describe_pet('hamster', 'harry')  
describe_pet('dog', 'willie')
```

En esta segunda llamada, pasamos a `describe_pet()` los argumentos `'dog'` y `'willie'`. Al igual que con el primer conjunto de argumentos que hemos usado, Python asocia `'dog'` con el parámetro `animal_type` y `'willie'` con `pet_name`. Como antes, la función hace su trabajo, pero ahora imprime valores para un perro llamado Willie. Ahora tenemos un hámster llamado Harry y un perro llamado Willie:

```
I have a hamster.  
My hamster's name is Harry.
```

```
I have a dog.  
My dog's name is Willie.
```

Llamar a una función varias veces es una forma muy eficiente de trabajar. El código que describe a la mascota se escribe solo una vez en la función. Después, cada vez que queramos describir una mascota nueva, llamaremos a la función con la nueva información. Si el código que utilizásemos para describir una mascota tuviera 10 líneas, podríamos seguir describiendo a una mascota nueva en una sola línea llamando de nuevo a la función.

El orden es importante en los argumentos posicionales

Si mezclamos el orden de los argumentos en una llamada a una función usando argumentos posicionales, podemos obtener resultados inesperados:

```
def describe_pet(animal_type, pet_name):  
    """Muestra información sobre una mascota."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}.")  
  
describe_pet('harry', 'hamster')
```

En esta llamada de función, hemos puesto primero el nombre y a continuación el tipo de animal. Dado que el argumento `'harry'` aparece primero, ese valor se asigna al parámetro `animal_type`. Del mismo modo, `'hamster'` se asigna a `pet_name`. Ahora tenemos un "harry" llamado "Hámster":

```
I have a harry.  
My harry's name is Hamster.
```

Si obtiene resultados raros como este, compruebe que el orden de los argumentos en la llamada a la función coincide con el orden de los parámetros de la definición.

Argumentos de palabra clave

Un argumento de palabra clave es un par nombre-valor que pasamos a una función. Se asocia directamente el nombre y el valor dentro del argumento, así que, cuando pasamos el argumento a la función, no hay confusiones (no acabaremos con un harry llamado Hámster). Los argumentos de palabra clave nos evitan tener que preocuparnos por el orden correcto de los argumentos en la llamada a la función y aclaran el papel de cada valor en la llamada.

Vamos a reescribir `pets.py` usando argumentos de palabra clave para llamar a `describe_pet()`:

```
def describe_pet(animal_type, pet_name):  
    """Muestra información sobre una mascota."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}.")  
  
describe_pet(animal_type='hamster', pet_name='harry')
```

La función `describe_pet()` no ha cambiado, pero, cuando la llamamos, decimos explícitamente a Python con qué parámetro debe asociarse cada argumento. Cuando Python lee la llamada a la función, sabe que debe asignar el argumento `'hamster'` al parámetro `animal_type` y `'harry'` a `pet_name`. La salida muestra correctamente que tenemos un hámster llamado Harry.

El orden de los argumentos de palabra clave no importa porque Python sabe dónde debe ir cada valor. Las dos siguientes llamadas son equivalentes:

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

Nota: Cuando utilice argumentos de palabra clave, asegúrese de usar los nombres exactos de los parámetros en la definición de la función.

Valores predeterminados

Cuando escribimos una función, podemos definir un valor para cada parámetro. Si se proporciona un argumento para un parámetro en la llamada a la función, Python usa el valor del argumento. De lo contrario, utiliza el valor predeterminado del parámetro. Así, al definir un valor predeterminado para un parámetro, podemos excluir el argumento correspondiente que normalmente escribiríamos en la llamada a la función. Usar valores predeterminados permite simplificar las llamadas a funciones y aclarar de qué forma se suelen utilizar nuestras funciones.

Por ejemplo, si observa que la mayoría de las llamadas a `describe_pet()` se usan para describir perros, puede establecer el valor predeterminado de `animal_type` en `'dog'`. Entonces, cualquiera que llame a `describe_pet()` para describir un perro podrá omitir esa información:

```
def describe_pet(pet_name, animal_type='dog'):  
    """Muestra información sobre una mascota."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}")  
  
describe_pet(pet_name='willie')
```

Hemos cambiado la definición de `describe_pet()` para incluir un valor predeterminado, `'dog'`, para `animal_type`. Ahora, cuando se llama a la función sin especificar `animal_type`, Python sabe que tiene que usar el valor `'dog'` para ese parámetro:

```
I have a dog.  
My dog's name is Willie.
```

Observe que ha habido que cambiar el orden de los parámetros en la definición de la función. Dado que el valor predeterminado hace innecesario especificar un tipo de animal como argumento, el único argumento que queda en la llamada a la función es el nombre de la mascota. Python sigue interpretando esto como un argumento posicional, así que, si se llama a la función solo con un nombre de mascota, ese argumento se asociará con el primer parámetro de la definición de la función. Por eso, el primer parámetro tiene que ser `pet_name`.

La forma más sencilla de usar esta función ahora es dar solo el nombre de un perro en la llamada:

```
describe_pet('willie')
```

Esta llamada tendría la misma salida que el ejemplo anterior. El único argumento proporcionado es `'willie'`, así que se asocia con el primer parámetro de la definición, `pet_name`. Como no se da un argumento para `animal_type`, Python usa el valor predeterminado, `'dog'`.

Para describir un animal que no sea un perro, podemos usar una llamada a la función como esta:

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

Como se da un argumento explícito para `animal_type`, Python ignora el valor predeterminado del parámetro.

Nota: Cuando use valores predeterminados, debe colocar cualquier parámetro con un valor predeterminado después de todos los parámetros que no tienen valores predeterminados. De este modo, Python podrá seguir interpretando argumentos posicionales correctamente.

Llamadas a funciones equivalentes

Dado que los argumentos posicionales, los argumentos de palabra clave y los valores predeterminados pueden emplearse juntos, a menudo tendrá varias formas equivalentes de llamar a una función. Considere la siguiente definición para `describe_pet()` con un valor predeterminado:

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

Con esta definición, siempre hay que proporcionar un argumento para `pet_name`, que puede darse con formato posicional o de palabra clave. Si el animal descrito no es un perro, hay que incluir un argumento para `animal_type` en la llamada, y este también puede especificarse en formato posicional o de palabra clave.

Todas estas llamadas funcionarían con esta función:

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

Cada una de estas llamadas tendría la misma salida que los ejemplos anteriores.

En realidad, no importa el estilo de llamada que use. En tanto las llamadas a la función produzcan la salida deseada, puede usar la estrategia que le resulte más fácil de entender.

Evitar errores con argumentos

Cuando empiece a usar funciones, no se sorprenda si encuentra errores sobre argumentos no coincidentes. Estos errores se producen cuando damos menos argumentos, o más, de los que necesita una función para hacer su trabajo. Por ejemplo, esto es lo que pasaría si intentásemos llamar a `describe_pet()` sin argumentos:

```
def describe_pet(animal_type, pet_name):  
    """Muestra información sobre una mascota."""  
    print(f"\nI have a {animal_type}.")  
    print(f"My {animal_type}'s name is {pet_name.title()}.")  
  
describe_pet()
```

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

```
Traceback (most recent call last):  
❶  File "pets.py", line 6, in <module>
```

```
❷     describe_pet()  
~~~~~  
❸  TypeError: describe_pet() missing 2 required positional arguments:  
    'animal_type' and 'pet_name'
```

El rastreo nos indica la localización del problema ❶, lo que nos permite retroceder para ver qué está mal en la llamada a la función. A continuación, vemos la llamada a la función incorrecta ❷. Por último, el rastreo nos indica que a la llamada le faltan dos argumentos y nos indica los nombres de los argumentos que faltan ❸. Si esta función estuviese en un archivo aparte, podríamos reescribir bien la llamada sin tener que abrir ese archivo para leer el código de la función.

Python nos ayuda leyendo el código de la función por nosotros y dándonos los nombres de los argumentos que tenemos que proporcionar. Este es otro de los motivos por los que conviene poner nombres descriptivos a variables y funciones. Si lo hace, el mensaje de error de Python será más útil para usted y para cualquiera que use su código.

Si da demasiados argumentos, debería obtener un rastreo similar que le ayudará a asociar correctamente la llamada a la función con la definición de la función.

PRUÉBELO

- **8-3. Camiseta:** Escriba una función llamada `hacer_camiseta()` que acepte una talla y un texto para un mensaje que habría que imprimir en la camiseta. La función debería imprimir una frase resumiendo la talla de la camiseta y el mensaje impreso.

Llame a la función una vez con argumentos posicionales para hacer una camiseta. Llame a la función por segunda vez usando argumentos de palabra clave.

- **8-4. Camisetas grandes:** Modifique la función `hacer_camiseta()` para que las camisetas sean grandes por defecto con un mensaje que diga "Me

encanta Python". Haga una camiseta grande y una mediana con el mensaje predeterminado y una de cualquier talla con un mensaje diferente.

- **8-5. Ciudades:** Escriba una función llamada `describir_ciudad()` que acepte el nombre de una ciudad y su país. La función debería imprimir una oración simple, como "Reikiavik está en Islandia". Dé al parámetro del país un valor predeterminado. Llame a la función para tres ciudades diferentes, con al menos una que no esté en el país predeterminado.

Valores de retorno

Una función no siempre tiene por qué mostrar su salida directamente. En vez de eso, puede procesar datos y devolver un valor o un conjunto de valores. El valor que devuelve la función se denomina "valor de retorno". La sentencia `return` coge un valor dentro de una función y lo envía de vuelta a la línea que ha llamado a la función. Los valores de retorno nos permiten pasar buena parte del trabajo monótono de un programa a funciones, lo que puede simplificar el cuerpo del programa.

Devolver un solo valor

Vamos a ver una función que coge un nombre y un apellido y devuelve un nombre completo con un formato limpio:

`formatted_name.py`

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

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

La definición de `get_formatted_name()` toma como parámetros un nombre y un apellido. La función combina estos dos nombres, añade un espacio entre ellos y asigna el resultado a `full_name` ❶. El valor de `full_name` se convierte para que las iniciales sean mayúsculas y luego se devuelve a la línea de la llamada ❷. Cuando llamamos a una función que devuelve un valor, tenemos que proporcionar una variable a la que se pueda asignar el valor de retorno. En este caso, el valor devuelto se asigna a la variable `musician` en ❸. La salida muestra un nombre con un formato adecuado, compuesto por las partes de un nombre completo:

```
Jimi Hendrix
```

Puede parecer mucho trabajo para conseguir un nombre con buen formato, pudiendo simplemente haber escrito:

```
print("Jimi Hendrix")
```

No obstante, cuando se piensa trabajar con un programa grande que requiere almacenar muchos nombres y apellidos por separado, funciones como `get_formatted_name()` pueden ser de gran utilidad. Guardamos los nombres y los apellidos por separado y posteriormente llamamos a esta función cuando necesitemos mostrar el nombre completo.

Hacer un argumento opcional

A veces, tiene sentido hacer que un argumento sea opcional para que la gente que usa la función pueda decidir si incluir información adicional o no. Podemos usar valores predeterminados para hacer un argumento opcional. Por ejemplo, supongamos que ampliamos `get_formatted_name()` para manejar también nombres compuestos. Un primer intento para incluir segundos nombres quedaría así:

```
def get_formatted_name(first_name, middle_name, last_name):
```

```
"""Devuelve un nombre completo, con un formato adecuado."""
full_name = f"{first_name} {middle_name} {last_name}"
return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Esta función funciona cuando se le da un nombre compuesto y un apellido. Coge las tres partes del nombre y crea una cadena con ellas. Luego añade espacios donde corresponda y pone cada nombre con mayúscula inicial:

John Lee Hooker

Pero el segundo nombre no es siempre necesario y esta función, tal cual está escrita, no funcionaría si intentásemos llamarla solo con un nombre y un apellido. Para convertir el segundo nombre en opcional, podemos dar al argumento `middle_name` un valor predeterminado vacío para ignorarlo a menos que el usuario introduzca un valor. Para que `get_formatted_name()` funcione sin segundo nombre, configuraremos el valor predeterminado de `middle_name` como una cadena vacía y lo moveremos al final de la lista de parámetros:

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Devuelve un nombre completo, con un formato adecuado."""

❶    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    ❷    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

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

```
❸ musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

En este ejemplo, el nombre se compone de tres partes posibles. Como siempre hay un nombre y un apellido, estos parámetros son los primeros que aparecen en la definición de la función. El segundo nombre es opcional, así que aparece al final y su valor predeterminado es una cadena vacía.

En el cuerpo de la función, comprobamos si se ha suministrado un segundo nombre. Python interpreta las cadenas no vacías como `True`, de manera que la prueba condicional `if middle_name` evalúa como `True` si hay un argumento para un segundo nombre en la llamada a la función ❶. Si hay un segundo nombre, el nombre compuesto y el apellido se combinan para formar un nombre completo. Entonces ese nombre se pone en mayúscula inicial y se devuelve a la línea de la llamada, donde se asigna a la variable `musician` y se imprime. Si no hay un segundo nombre, la cadena vacía no pasa la prueba `if` y se ejecuta el bloque ❷. El nombre completo se compone solo de un nombre y un apellido, se le aplica el formato y se devuelve a la línea de la llamada, donde se asigna a `musician` y se imprime.

Llamar a esta función con un nombre y un apellido es fácil. Sin embargo, si usamos un nombre compuesto, tenemos que asegurarnos de que el segundo nombre sea el último argumento pasado para que Python haga coincidir los argumentos posicionales correctamente ❸.

Esta versión modificada de nuestra función va bien para personas con un solo nombre y un apellido y también para personas con un nombre compuesto:

Jimi Hendrix

John Lee Hooker

Los valores opcionales permiten a las funciones manejar una amplia variedad de casos de uso al tiempo que la función se

mantiene lo más sencilla posible.

Devolver un diccionario

Una función puede devolver cualquier tipo de valor que necesitemos, incluidas estructuras de datos más complejas, como listas y diccionarios. Por ejemplo, la siguiente función coge partes de un nombre y devuelve un diccionario que representa a una persona:

person.py

```
def build_person(first_name, last_name):
    """Devuelve un diccionario de información sobre una persona."""
❶    person = {'first': first_name, 'last': last_name}
❷    return person

musician = build_person('jimi', 'hendrix')
❸    print(musician)
```

La función `build_person()` coge un nombre y un apellido y pone estos valores en un diccionario ❶. El valor de `first_name` se guarda con la clave `'first'` y el de `last_name`, con la clave `'last'`. El diccionario completo que representa a la persona se devuelve ❷. El valor de retorno se imprime ❸ con las dos informaciones textuales originales almacenadas ahora en un diccionario:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Esta función coge información textual simple y la pone en una estructura de datos más significativa que nos permite trabajar con la información para hacer algo más que imprimirla. Las cadenas `'jimi'` y `'hendrix'` están ahora etiquetadas como un nombre y un apellido. Podemos ampliar fácilmente esta función para que acepte valores opcionales, como un segundo nombre, una edad, una profesión o

cualquier otra información que queramos guardar sobre una persona. Por ejemplo, el siguiente cambio nos permite guardar también la edad de la persona:

```
def build_person(first_name, last_name, age=None):
    """Devuelve un diccionario de información sobre 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 un nuevo parámetro opcional `age` a la definición de la función y le asignamos el valor especial `None`, que se usa cuando una variable no tiene un valor específico asignado. Puede considerar `None` como un marcador de posición. En las pruebas condicionales, `None` se evalúa como `False`. Si la llamada a la función incluye un valor para `age`, ese valor se guarda en el diccionario. Esta función siempre guarda un nombre de persona, pero puede modificarse para almacenar cualquier otra información personal que queramos.

Usar una función con un bucle while

Podemos usar funciones con todas las estructuras de Python que hemos visto hasta ahora. Por ejemplo, vamos a usar la función `get_formatted_name()` con un bucle `while` para saludar a los usuarios de una manera más formal. Este es un primer intento de saludar a la gente con su nombre completo:

`greeter.py`

```
def get_formatted_name(first_name, last_name):
    """Devuelve un nombre completo, con un formato adecuado."""

```

```

full_name = f"{first_name} {last_name}"

return full_name.title()

# ¡Esto es un bucle infinito!

while True:
    ❶    print("\nPlease tell me your name:")

    f_name = input("First name: ")

    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)

    print(f"\nHello, {formatted_name}!")

```

Para este ejemplo, usamos una versión sencilla de `get_formatted_name()` que no incluye la posibilidad de nombres compuestos. El bucle `while` pide al usuario que introduzca su nombre y se le indica que escriba nombre y apellido por separado ❶.

Pero hay un problema con este bucle `while`: no hemos definido una condición. ¿Dónde ponemos una condición para salir cuando pedimos una serie de entradas? Queremos que el usuario pueda salir del bucle de la manera más sencilla posible, por lo que cada indicación debería ofrecerle esa posibilidad. La sentencia `break` ofrece una forma sencilla de salir del bucle en cada indicación:

```

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

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    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"\nHello, {formatted_name}!")
```

Añadimos un mensaje que informe al usuario de cómo salir y terminamos el bucle si el usuario introduce el valor para salir en cualquier indicación. Ahora el programa seguirá saludando a la gente hasta que alguien escriba '`q`' para cualquier nombre:

```
Please tell me your name:  
(enter 'q' at any time to quit)  
First name: eric  
Last name: matthes
```

```
Hello, Eric Matthes!
```

```
Please tell me your name:  
(enter 'q' at any time to quit)  
First name: q
```

PRUÉBELO

- **8-6. Nombres de ciudad:** Escriba una función llamada `ciudad_pais()` que admita el nombre de una ciudad y su país. La función debería devolver una cadena con formato, similar a esta:

```
"Santiago, Chile"
```

Llame a su función con al menos tres pares ciudad-país e imprima los valores devueltos.

- **8-7. Álbum:** Escriba una función llamada `hacer_album()` que cree un diccionario para describir un álbum musical. La función debería aceptar un

nombre de artista y un título de álbum y debería devolver un diccionario con estas dos informaciones. Use la función para hacer tres diccionarios que representen distintos álbumes. Imprima cada valor devuelto para comprobar que los diccionarios están almacenando bien la información.

Use `None` para añadir un parámetro opcional a `hacer_album()` que permita guardar el número de canciones que contiene un álbum. Si la línea de llamada incluye un valor para el número de canciones, añádalo al diccionario del álbum. Haga al menos una nueva llamada a la función que incluya este dato.

- **8-8. Álbumes de usuarios:** Empiece con el programa del ejercicio 8-7. Escriba un bucle `while` que permita a los usuarios introducir el artista y el título de un álbum. Una vez que disponga de esa información, llame a `hacer_album()` con la entrada de usuario e imprima el diccionario que se ha creado. Asegúrese de incluir un valor para salir en el bucle `while`.

Pasar una lista

Con frecuencia, le resultará útil pasar una lista a una función, ya se trate de una lista de nombres, de números o de objetos más complejos, como diccionarios. Cuando pasamos una lista a una función, la función obtiene acceso directo al contenido de la lista. Vamos a usar funciones para hacer más eficiente el trabajo con listas.

Supongamos que tenemos una lista de usuarios y queremos imprimir un mensaje para saludar 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:

`greet_users.py`

```
def greet_users(names):
    """Imprime un saludo sencillo para cada usuario de la lista."""
    for name in names:
        msg = f"Hello, {name.title()}!"
```

```
print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Definimos `greet_users()` para que espere una lista de nombres, que asignará al parámetro `names`. La función pasará en bucle por la lista recibida e imprimirá un saludo para cada usuario. Fuera de la función, definimos una lista de usuarios y pasamos la lista `usernames` a `greet_users()` en la llamada a la función.

Hello, Hannah!

Hello, Ty!

Hello, Margot!

Esta es la salida que queríamos. Cada usuario ve un saludo personalizado y podemos llamar a la función siempre que queramos saludar a un grupo concreto de usuarios.

Modificar una lista en una función

Cuando pasamos una lista a una función, la función puede modificarla. Cualquier cambio que se haga en la lista dentro del cuerpo de la función es permanente, lo que nos permite trabajar con eficiencia cuando tratamos con grandes cantidades de datos.

Piense en una empresa que crea modelos en impresión 3D de los diseños que envían los usuarios. Los diseños que hay que imprimir se guardan en una lista y, una vez impresos, pasan a una lista aparte. El siguiente código hace esto sin usar funciones:

printing_models.py

```
# Empieza con unos diseños que hay que imprimir.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []
```

```
# Simula la impresión de cada diseño hasta que no queda ninguno.  
# Mueve cada diseño a completed_models después de la impresión.  
  
while unprinted_designs:  
    current_design = unprinted_designs.pop()  
    print(f"Printing model: {current_design}")  
    completed_models.append(current_design)  
  
# Muestra todos los modelos completados.  
print("\nThe following models have been printed:")  
for completed_model in completed_models:  
    print(completed_model)
```

Este programa empieza con una lista de diseños que hay que imprimir y una lista llamada `completed_models` a la que se trasladará cada diseño una vez impreso. Mientras haya diseños en `unprinted_designs`, el bucle `while` simulará la impresión de cada diseño quitándolo del final de la lista, almacenándolo en `current_design` y mostrando un mensaje diciendo que el diseño actual se está imprimiendo. Luego añade el diseño a la lista de modelos completos. Cuando el bucle termina de ejecutarse, se muestra una lista de los diseños impresos:

```
Printing model: dodecahedron  
Printing model: robot pendant  
Printing model: phone case  
  
The following models have been printed:  
dodecahedron  
robot pendant  
phone case
```

Podemos reorganizar este código escribiendo dos funciones, cada una con una tarea específica. La mayor parte del código no cambiará; simplemente lo estamos estructurando con más cuidado.

La primera función se ocupará de la impresión de los diseños y la segunda resumirá las impresiones que se han hecho:

```
❶ def print_models(unprinted_designs, completed_models):
    """
    Simula imprimir cada diseño, hasta que no queda ninguno.
    Mueve cada diseño a completed_models después de la impresión.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)

❷ def show_completed_models(completed_models):
    """Muestra todos los modelos que se han imprimido."""
    print("\nThe following models have been printed:")
    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 los diseños que hay que imprimir y una lista de los modelos completados ❶. Con estas dos listas, la función simula la impresión de cada diseño vaciando la lista de diseños sin imprimir y rellenando la de modelos completados. A continuación, definimos la función `show_completed_models()` con un parámetro: la lista de modelos completados ❷. Con esta lista, `show_completed_models()` muestra el nombre de cada modelo que se ha imprimido.

Este programa tiene la misma salida que la versión sin funciones, pero el código está mucho más organizado. El código que hace la mayor parte del trabajo se ha pasado a dos funciones separadas, lo que hace la parte principal del programa más fácil de entender. Eche un vistazo al cuerpo del programa y fíjese en lo fácil que es seguir lo que ocurre:

```
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 sin imprimir y una lista vacía que alojará los modelos completados. Luego, como ya hemos definido nuestras dos funciones, lo único que tenemos que hacer es llamarlas y pasarles los argumentos adecuados. Llamamos a `print_models()` y le pasamos las dos listas que necesita; como cabía esperar, `print_models()` simula la impresión de los diseños. A continuación llamamos a `show_completed_models()` y le pasamos la lista de modelos completados para que pueda informar de los modelos que se han imprimido. Los nombres descriptivos de las funciones permiten entender el código a otros lectores, aunque no haya comentarios.

Este programa es más fácil de ampliar y mantener que la versión sin funciones. Si más adelante necesitamos imprimir más diseños, podemos volver a llamar a `print_models()`. Si nos damos cuenta de que hay que modificar el código de impresión, podemos cambiarlo una sola vez y los cambios se aplicarán siempre que se llame a la función. Esta técnica es más eficiente que tener que actualizar el código por separado en varios puntos del programa.

Este ejemplo también ilustra la idea de que cada función debería tener una tarea específica. La primera función imprime cada diseño y la segunda muestra los modelos completados. Es mejor que usar una función para ambas misiones. Si está escribiendo una función y observa que hace demasiadas tareas diferentes, pruebe a repartir el

código en dos funciones. Recuerde que siempre puede llamar a una función desde otra, lo cual puede ser muy útil cuando se descompone una tarea compleja en una serie de pasos.

Evitar que una función modifique una lista

A veces nos interesa evitar que una función pueda modificar una lista. Por ejemplo, supongamos que empezamos con una lista de diseños sin imprimir y escribimos una función para que los mueva a una lista de modelos completados, como en el ejemplo anterior. Podríamos decidir que, aunque todos los diseños estén impresos, preferimos mantenerlos en la lista original de diseños sin imprimir para nuestros registros. Sin embargo, dado que hemos sacado todos los nombres de los diseños de `unprinted_designs`, la lista está ahora vacía y esa versión es la única que tenemos: la original ha desaparecido. En este caso, podemos resolver el problema pasando a la función una copia de la lista en lugar de la lista original. De ese modo, cualquier cambio que haga la función en la lista solo afectará a la copia, quedando la lista original intacta.

Así es como se envía una copia de una lista a una función:

```
nombre_función(nombre_lista[:])
```

La notación para partir `[:] hace una copia de la lista y se la envía a la función. Si no quisiéramos vaciar la lista de diseños sin imprimir de printing_models.py, podríamos llamar a print_models() así:`

```
print_models(unprinted_designs[:], completed_models)
```

La función `print_models()` puede hacer su trabajo porque sigue recibiendo los nombres de los diseños sin imprimir. Pero, en esta ocasión, usa una copia de la lista original, no la lista `unprinted_designs` real. La lista `completed_models` se llenará con los nombres de los modelos impresos igual que antes, pero la lista original de diseños sin imprimir no se verá afectada por la función.

Aunque podemos preservar el contenido de una lista pasando una copia a las funciones, conviene pasar la lista original, salvo que haya una razón específica para mandar la copia. Es más eficiente para una función trabajar con una lista existente, ya que con ello se evita usar el tiempo y la memoria que requiere hacer una copia por separado. Esto es especialmente cierto cuando se trabaja con listas grandes.

PRUÉBELO

- **8-9. Mensajes:** Haga una lista con una serie de mensajes de texto cortos. Pásela a una función llamada `mostrar_mensajes()` que imprima cada mensaje.
- **8-10. Enviar mensajes:** Empiece con una copia del programa del ejercicio 8-9. Escriba una función llamada `enviar_mensajes()` que imprima cada mensaje de texto y lo mueva a una nueva lista denominada `mensajes_enviados` a medida que imprime. Después de llamar a la función, imprima ambas listas para asegurarse de que los mensajes se han movido correctamente.
- **8-11. Mensajes archivados:** A partir del trabajo realizado para el ejercicio 8-10, llame a la función `enviar_mensajes()` con una copia de la lista de mensajes. Después, imprima ambas listas para confirmar que la lista original conserva sus mensajes.

Pasar un número arbitrario de argumentos

A veces, no sabemos de antemano cuántos argumentos necesita aceptar una función. Por suerte, Python permite a una función recoger un número arbitrario de argumentos de la sentencia de llamada.

Por ejemplo, considere una función para crear una pizza. Tiene que aceptar una serie de ingredientes, pero no sabemos cuántos querrá el usuario. La función del siguiente ejemplo tiene un

parámetro, `*toppings`, que recoge todos los argumentos que le proporcione la línea de la llamada:

pizza.py

```
def make_pizza(*toppings):
    """Imprime la lista de ingredientes solicitados."""
    print(toppings)

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

El asterisco en el nombre del parámetro `*toppings` le indica a Python que cree una tupla vacía llamada `toppings`, que contenga todos los valores que reciba esta función. La llamada a `print()` en el cuerpo de la función produce una salida que demuestra que Python puede administrar una llamada a una función con un valor y una llamada con tres valores. Trata las distintas llamadas de una forma similar. Observe que Python mete los argumentos en una tupla, incluso aunque la función reciba solo un valor:

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Ahora podemos reemplazar la llamada a `print()` con un bucle que se ejecute por la lista de ingredientes y describa la pizza que se está pidiendo:

```
def make_pizza(*toppings):
    """Resume la pizza que estamos a punto de hacer."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

La función responde adecuadamente, tanto si recibe un valor como si recibe tres:

Making a pizza with the following toppings:

- pepperoni

Making a pizza with the following toppings:

- mushrooms
 - green peppers
 - extra cheese
-

Esta sintaxis funciona independientemente de cuántos argumentos reciba la función.

Mezclar argumentos posicionales y arbitrarios

Si quiere que una función acepte distintos tipos de argumentos, debe colocar el parámetro que acepta un número arbitrario de argumentos al final de la definición de la función. Python asocia primero los argumentos posicionales y de palabra clave y luego recoge los restantes en el parámetro final.

Por ejemplo, si la función tiene que admitir un tamaño para la pizza, tendrá que ir antes de `*toppings`:

```
def make_pizza(size, *toppings):  
    """Resume la pizza que estamos a punto de hacer."""  
    print(f"\nMaking a {size}-inch pizza with the following toppings:")  
    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 demás valores que vengan

después irán a la tupla `toppings`. Las llamadas a la función incluyen un argumento para el tamaño primero, seguido de tantos ingredientes como sean necesarios.

Ahora cada pizza tiene un tamaño y un número de ingredientes y cada dato se imprime en el lugar adecuado, mostrando primero el tamaño y después los ingredientes:

Making a 16-inch pizza with the following toppings:

- pepperoni

Making a 12-inch pizza with the following toppings:

- mushrooms
 - green peppers
 - extra cheese
-

Nota: Con frecuencia verá el nombre de parámetro genérico `*args`, que recoge argumentos posicionales arbitrarios como este.

Usar argumentos de palabra clave arbitrarios

En ocasiones puede interesarnos aceptar un número de argumentos arbitrario, pero no sabemos de antemano qué tipo de información se pasará a la función. En ese caso, podemos escribir funciones que acepten tantos pares clave-valor como ofrezca la sentencia de llamada. Veamos un ejemplo con perfiles de usuario: sabemos que obtendremos información sobre un usuario, pero no estamos seguros de qué tipo de información será. La función `build_profile()` del siguiente ejemplo toma siempre un nombre y un apellido y acepta además un número arbitrario de argumentos de palabra clave.

`user_profile.py`

```
def build_profile(first, last, **user_info):
```

```
"""Crea un diccionario con todo lo que sabemos sobre un
usuario."""
❶ 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 deja que el usuario pase después todos los pares clave-valor que quiera. El doble asterisco antes del parámetro `**user_info` hace que Python cree un diccionario vacío llamado `user_info` que contenga todos los pares nombre-valor adicionales que la función reciba. Dentro de la función, podemos acceder a los pares clave-valor en `user_info` igual que en cualquier otro diccionario.

En el cuerpo de `build_profile()`, añadimos el nombre y el apellido al diccionario `user_info` porque siempre vamos a recibir estos dos datos del usuario ❶, y que todavía no se han incluido en el diccionario. Después devolvemos el diccionario `user_info` a la línea de llamada a la función.

Llamamos a `build_profile()`, pasándole el nombre `'albert'`, el apellido `'einstein'` y los dos pares clave-valor `location='princeton'` y `field='physics'`. Asignamos el perfil devuelto a `user_profile` y lo imprimimos:

```
{'location': 'princeton', 'field': 'physics',
'first_name': 'albert', 'last_name': 'einstein'}
```

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 funcionaría independientemente de cuántos pares clave-valor adicionales se proporcionen en la llamada a la función.

Al escribir sus propias funciones, puede mezclar argumentos posicionales, de palabra clave y valores arbitrarios de muchas formas. Es útil saber que existen todos estos tipos de argumentos porque los verá con frecuencia cuando empiece a leer el código de otros programadores. Lleva práctica aprender a usar correctamente los distintos tipos de funciones y saber cuándo utilizar cada uno. Por ahora, recuerde usar el enfoque más sencillo que consiga el resultado adecuado. A medida que avance, aprenderá a usar el enfoque más eficiente para cada situación.

Nota: Con frecuencia encontrará el nombre de parámetro `**kwargs`, que sirve para recoger argumentos de palabra clave no específicos.

PRUÉBELO

- **8-12. Sándwiches:** Escriba una función que acepte una lista de elementos que quiere una persona para un sándwich. La función debería tener un parámetro que recoja todos los argumentos que le dé la llamada e imprimir un resumen del sándwich que se está pidiendo. Llame a la función tres veces, usando un número distinto de argumentos cada vez.
- **8-13. Perfil de usuario:** Empiece con una copia de `user_profile.py`. Haga un perfil suyo llamando a `build_profile()`, usando su nombre y apellido y otros tres pares clave-valor que le describan.
- **8-14. Coche:** Escriba una función que guarde información sobre un coche en un diccionario. Debería recibir siempre un fabricante y un nombre de modelo y aceptar después un número arbitrario de argumentos de palabra clave. Llame a la función con la información requerida y otros dos pares nombre-valor, como un color o una prestación opcional. Su función debería funcionar para una llamada como esta:

```
car = make_car('subaru', 'outback', color='blue', tow_package=True)
```

Imprima el diccionario devuelto y asegúrese de que se ha almacenado bien toda la información.

Guardar las funciones en módulos

Una ventaja de las funciones es la forma en la que separan bloques de código del programa principal. Cuando empleamos nombres descriptivos para las funciones, los programas son mucho más fáciles de seguir. Puede ir un paso más allá almacenando sus funciones en un archivo aparte llamado "módulo", importando después ese módulo al programa principal. Una sentencia `import` le dice a Python que ponga el código de un módulo a disposición del archivo de programa en ejecución.

Guardar las funciones en un archivo aparte nos permite ocultar los detalles del código del programa para concentrarnos en su lógica de nivel superior. También nos permite reutilizar las funciones en distintos programas. Y, cuando guardamos funciones en archivos independientes, podemos compartirlos con otros programadores sin necesidad de compartir el programa completo. Saber cómo importar funciones también permite usar bibliotecas de funciones que han escrito otros programadores.

Hay varias formas de importar un módulo y vamos a verlas todas brevemente.

Importar un módulo completo

Para empezar a importar funciones, primero tenemos que crear un módulo. Un módulo es un archivo con extensión `.py` que contiene el código que queremos importar a nuestro programa.

Vamos a crear un módulo que contenga la función `make_pizza()`. Para ello, quitaremos del archivo `pizza.py` todo menos la función que nos interesa:

`pizza.py`

```
def make_pizza(size, *toppings):
    """Resume la pizza que estamos a punto de hacer."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

Ahora haremos un archivo independiente llamado `making_pizzas.py` en el mismo directorio que `pizza.py`. Este archivo importa el módulo que hemos creado y hace dos llamadas nuevas a `make_pizza()`:

making_pizzas.py

```
import pizza

❶ pizza.make_pizza(16, 'pepperoni')
pizza.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 en este programa. No vemos realmente código copiándose entre archivos porque Python realiza la copia entre bambalinas, justo antes de ejecutar el programa. Lo único que hay que saber es que cualquier función definida en `pizza.py` estará ahora disponible en `making_pizzas.py`.

Para llamar a una función desde un módulo importado, escribiremos el nombre del módulo, `pizza`, seguido del nombre de la función, `make_pizza()`, separado por un punto ❶. Este código produce la misma salida que el programa original donde no se importaba ningún módulo:

```
Making a 16-inch pizza with the following toppings:
- pepperoni
```

```
Making a 12-inch pizza with the following toppings:
- mushrooms
```

- green peppers
 - extra cheese
-

Esta primera aproximación a la importación, en la que solo escribimos `import` seguido del nombre del módulo, hace que todas las funciones del módulo estén disponibles en un programa. Si usa este tipo de sentencia `import` para importar un módulo completo llamado `nombre_módulo.py`, todas las funciones del módulo estarán disponibles a través de esta sintaxis:

```
nombre_módulo.nombre_función()
```

Importar funciones específicas

También podemos importar una función específica desde un módulo. Esta es la sintaxis general para esta técnica:

```
from nombre_módulo import nombre_función
```

Podemos importar tantas funciones como queramos desde un módulo separando el nombre de cada una con una coma:

```
from nombre_módulo import función_0, función _1, función _2
```

El ejemplo de `making_pizzas.py` quedaría así si quisieramos importar solo la función que vamos a usar:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Con esta sintaxis no necesitamos la notación de punto al llamar a una función. Como hemos importado explícitamente la función

`make_pizza()` en la sentencia `import`, podemos llamarla por su nombre cuando la usemos.

Usar `as` para dar un alias a una función

Si el nombre de una función que vamos a importar puede generar un conflicto con un nombre que ya existe en el programa o es demasiado largo, podemos usar un alias corto y único, un nombre alternativo, similar a un apodo para la función. Daremos a la función este alias especial al importarla.

Aquí vamos a dar a la función `make_pizza()` el alias `mp()`, importando `make_pizza` como `mp`. La palabra clave `as` renombra una función con el alias que le demos:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

La sentencia `import` que se muestra aquí cambia el nombre de la función `make_pizza()` por `mp()` en este programa. Siempre que queramos llamar a `make_pizza()`, bastará con escribir `mp()` y Python ejecutará el código de `make_pizza()`, evitando confusiones con cualquier otra función `make_pizza()` que podamos haber escrito en este archivo de programa.

Esta es la sintaxis general para poner un alias:

```
from nombre_módulo import nombre_función as nf
```

Usar `as` para dar un alias a un módulo

También podemos proporcionar un alias para un nombre de módulo. Dar un alias corto a un módulo, como `p` para `pizza`, nos

permite llamar a las funciones de ese módulo más deprisa. Llamar a `p.make_pizza()` es más conciso que llamar a `pizza.make_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

El módulo `pizza` recibe el alias `p` en la sentencia `import`, pero todas las funciones del módulo conservan sus nombres originales. Llamar a las funciones escribiendo `p.make_pizza()` no solo es más conciso que escribir `pizza.make_pizza()`, sino que además aleja nuestra atención del nombre del módulo y nos permite concentrarnos en los nombres descriptivos de sus funciones. Estos nombres, que nos dicen claramente lo que hace cada función, son más importantes para la legibilidad del código que usar el nombre del módulo completo.

Esta es la sintaxis general para esta aproximación:

```
import nombre_módulo as nm
```

Importar todas las funciones de un módulo

Podemos decirle a Python que importe todas las funciones de un módulo con el operador asterisco (*):

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

El asterisco de la sentencia `import` le dice a Python que copie todas las funciones del módulo `pizza` en este archivo de programa. Como se importan todas, podemos llamarlas por su nombre sin usar la notación de punto. Sin embargo, es mejor no usar esta técnica cuando trabaje con módulos grandes que no haya escrito usted: si el

módulo tiene un nombre de función que coincide con otro de su proyecto, puede obtener resultados inesperados. Python puede ver varias funciones o variables con el mismo nombre y, en vez de importar todas por separado, sobrescribirá las funciones.

Lo más recomendable es importar la función o funciones que le interesen, o el módulo completo, y usar la notación de punto. Así el código queda más claro y es más fácil de leer y entender. He incluido esta sección para que reconozca sentencias `import` como esta cuando las vea en el código de otros programadores:

```
from nombre_módulo import *
```

Dar estilo a las funciones

Debe tener algunos detalles en mente a la hora de aplicar estilo a sus funciones. Las funciones deberían tener siempre nombres descriptivos, que deberían ir en minúscula y con guiones. Los nombres descriptivos nos ayudan a todos a entender lo que intenta hacer el código. Los nombres de módulo deberían seguir las mismas convenciones.

Cada función debería llevar un comentario que explique con concisión lo que hace. El comentario debería aparecer inmediatamente después de la definición de la función y emplear el formato de la cadena de documentación. Con una función bien documentada, otros programadores pueden usar la función con tan solo leer la descripción del comentario. Deberían poder confiar en que el código funciona como se describe y, siempre y cuando conozcan el nombre de la función, los argumentos que necesita y el tipo de valor que devuelve, deberían poder usarlo en sus programas.

Si especifica un valor para un parámetro, no debería usar espacios a los lados del signo de igualdad:

```
def nombre_función(parámetro_0, parámetro_1='valor predeterminado')
```

La misma convención se aplica a los argumentos de palabra clave en las llamadas a una función:

```
nombre_función(valor_0, parámetro_1='valor')
```

La PEP 8 (<https://www.python.org/dev/peps/pep-0008/>) recomienda limitar las líneas de código a 79 caracteres para que se vean todas las líneas en una ventana de editor de tamaño razonable. Si un conjunto de parámetros hace que la definición de una función tenga más de 79 caracteres, pulse **Intro** después de abrir paréntesis en la línea de definición. En la siguiente línea, pulse **Tab** dos veces para separar la lista de argumentos del cuerpo de la función, que solo irá sangrado un nivel.

La mayoría de los editores alinean automáticamente las líneas de parámetros adicionales para hacerlas coincidir con la sangría establecida en la primera línea:

```
def nombre_función(  
    parámetro_0, parámetro_1, parámetro_2,  
    parámetro_3, parámetro_4, parámetro_5):  
    cuerpo de la función...
```

Si su programa o módulo incluye más de una función, puede separarlas con dos líneas en blanco para dejar más claro dónde acaba una y empieza otra.

Todas las sentencias `import` deberían escribirse al principio de un archivo. La única excepción es si hay comentarios iniciales para describir el programa completo.

PRUÉBELO

- **8-15. Imprimir modelos:** Ponga las funciones del ejemplo `printing_models.py` en un archivo aparte llamado `printing_functions.py`. Escriba una sentencia `import` en la parte superior de `printing_models.py` y modifique el archivo para usar las funciones importadas.

- **8-16. Importaciones:** Use uno de los programas que haya escrito que contenga una función y guarde esa función en un archivo aparte. Importe la función al archivo de programa principal y llámela con cada una de estas técnicas:

```
import nombre_módulo
from nombre_módulo import nombre_función
from nombre_módulo import nombre_función as nf
import nombre_módulo as nm
from nombre_módulo import *
```

- **8-17. Dar estilo a funciones:** Elija tres programas que haya escrito para este capítulo y asegúrese de que siguen las directrices de estilo descritas en el último apartado.

Resumen

En este capítulo hemos visto cómo escribir funciones y pasarles argumentos para que puedan acceder a la información que necesitan para hacer su trabajo. Ha aprendido a usar argumentos posicionales y de palabra clave, y cómo aceptar un número arbitrario de argumentos. Hemos visto funciones que muestran una salida y otras que devuelven valores. Ahora sabe usar funciones con listas, diccionarios, sentencias `if` y bucles `while`. También ha aprendido a guardar sus funciones en archivos independientes llamados módulos para que sus archivos de programa sean más sencillos y fáciles de entender. Por último, hemos visto cómo aplicar estilo a las funciones para que sus programas sigan estando bien estructurados y sean lo más fáciles de leer posible para todos.

Uno de nuestros objetivos como programadores debería ser escribir código sencillo que haga lo que nosotros queremos. Las funciones deberían ayudarnos a conseguirlo. Nos permiten escribir bloques de código y no volver a tocarlos una vez hemos comprobado que funcionan. Cuando sabemos que una función hace bien su

trabajo, podemos confiar en que seguirá funcionando y pasar a la siguiente tarea de programación.

Las funciones nos permiten escribir código una vez y reutilizarlo tantas veces como queramos. Cuando tengamos que ejecutar el código de una función, únicamente tenemos que escribir una llamada de una línea y la función cumplirá su misión. Cuando necesitemos modificar el comportamiento de una función, solo tendremos que modificar un bloque de código y los efectos se notarán en todas las llamadas a esa función.

Las funciones hacen que los programas sean más fáciles de leer, y un buen nombre de función resume adecuadamente lo que hace cada parte del programa. Leer una serie de llamadas a funciones nos da una idea más rápida de lo que hace un programa que leer una serie larga de bloques de código.

Las funciones también hacen el código más fácil de probar y depurar. Cuando el grueso del trabajo de un programa lo hace un conjunto de funciones, cada una con su tarea específica, es mucho más fácil probar y mantener el código que hemos escrito. Podemos escribir un programa aparte que llame a cada función y pruebe si funcionan en todas las situaciones que pueden encontrarse. Así, podremos tener la tranquilidad de que nuestras funciones funcionarán bien siempre que las llamemos.

En el capítulo 9, veremos cómo escribir clases. Las clases combinan funciones y datos en un paquete limpio que se puede usar de formas flexibles y eficientes.

9

CLASES



La programación orientada a objetos (POO) es uno de los enfoques más eficaces a la hora de escribir software. En programación orientada a objetos escribimos "clases" que representan cosas y situaciones del mundo real, y creamos "objetos" basados en esas clases. Cuando escribimos una clase, definimos el comportamiento general que puede tener una categoría completa de

objetos.

Cuando creamos objetos individuales de una clase, cada objeto asume automáticamente el comportamiento general; después, podemos dar a cada objeto los rasgos únicos que queramos. Le sorprenderá lo bien que se puede modelar situaciones del mundo real con la programación orientada a objetos.

La creación de un objeto a partir de una clase recibe el nombre de "instanciación". Trabajamos con "instancias" de una clase. En este capítulo, escribiremos clases y crearemos instancias de esas clases. Especificaremos el tipo de información que puede albergar cada instancia y definiremos las acciones que se puede hacer con ellas. También escribiremos clases que amplíen la funcionalidad de clases existentes, de modo que clases similares puedan compartir una funcionalidad común y podamos hacer más con menos código. Guardaremos las clases en módulos e importaremos clases escritas por otros programadores a nuestros propios archivos de programa.

Aprender programación orientada a objetos le ayudará a ver el mundo como lo ven los programadores. Le ayudará a entender su código: no solo lo que pasa en cada línea, sino los conceptos generales subyacentes al conjunto. Conocer la lógica subyacente a

las clases le entrenará para pensar lógicamente a la hora de escribir programas que resuelvan con eficacia casi cualquier problema.

Al igual que les ocurre a todos los programadores con los que colabore, las clases le facilitarán las cosas a medida que deba enfrentarse a desafíos cada vez más complejos. Cuando usted y otros programadores escriban código basado en el mismo tipo de lógica, todos serán capaz de entender el trabajo de los demás. Sus programas tendrán sentido para sus colaboradores y todo el mundo saldrá ganando.

Crear y usar una clase

Podemos modelar prácticamente cualquier cosa usando clases. Empezaremos escribiendo una clase sencilla, `Dog`, que representa un perro (no uno en particular, sino cualquier perro). ¿Qué sabemos sobre la mayoría de los perros que la gente tiene como mascota? Bueno, todos tienen nombre y edad. También sabemos que muchos saben sentarse y hacer la croqueta. Esos dos datos (nombre y edad) y los dos comportamientos (sentarse y hacer la croqueta) irán en nuestra clase `Dog`, porque son comunes a la mayoría de los perros. Esta clase le dirá a Python cómo crear un objeto que represente un perro. Una vez escrita la clase, la usaremos para hacer instancias individuales que representen perros específicos.

Creación de la clase Dog

Cada instancia creada a partir de la clase `Dog` contendrá un nombre y una edad y daremos a cada perro la capacidad de sentarse (`sit()`) y hacer la croqueta (`roll_over()`):

`dog.py`

```
❶ class Dog:  
    """Un intento sencillo de modelar un perro."""
```

```
❷ def __init__(self, name, age):
    """Inicializa los atributos de nombre y edad."""
❸     self.name = name
     self.age = age

❹ def sit(self):
    """Simula un perro sentándose en respuesta a una orden."""
    print(f"{self.name} is now sitting.")

def roll_over(self):
    """Simula hacer la croqueta en respuesta a una orden."""
    print(f"{self.name} rolled over!")
```

En este código deberá prestar atención a muchas cosas, pero no se preocupe. A lo largo del capítulo verá esta estructura y tendrá mucho tiempo para acostumbrarse a ella. En primer lugar, definimos una clase llamada `dog` ❶. Por convención, los nombres con inicial mayúscula hacen referencia a clases en Python. No hay paréntesis en la definición de la clase porque la estamos creando desde cero. A continuación, escribimos una cadena de documentación que describe lo que hace esta clase.

El método `__init__()`

Una función que forma parte de una clase es un "método". Todo lo que hemos visto sobre las funciones es también aplicable a los métodos; la única diferencia práctica en este momento es la forma en la que llamaremos a los métodos. El método `__init__()` ❷ es un método especial que Python ejecutará automáticamente siempre que creamos una nueva instancia basada en la clase `Dog`. Este método tiene dos guiones bajos a cada lado, una convención que ayuda a evitar que los nombres de métodos predeterminados de Python entren en conflicto con los nuestros. Asegúrese de usar dos

guiones a los lados `_init_()`. Si solo pone uno a cada lado, no se llamará al método automáticamente cuando use la clase, lo que producirá errores difíciles de identificar. Definimos el método `_init_()` con tres parámetros: `self`, `name` y `age`. El parámetro `self` es necesario en la definición del método y debe ir antes que los otros. Debe incluirse en la definición porque, cuando Python llame a este método más adelante (para crear una instancia de `Dog`), la llamada pasará automáticamente el argumento `self`. Cada llamada al método asociada con una instancia pasa automáticamente `self`, que es una referencia a la propia instancia; da a la instancia individual acceso a los atributos y métodos de la clase. Cuando hagamos una instancia de `Dog`, Python llamará al método `_init_()` desde la clase `Dog`. Pasaremos a `Dog()` un nombre y una edad como argumentos; `self` pasa automáticamente, así que no tenemos que hacer nada. Cuando queramos hacer una instancia de la clase `Dog`, solo suministraremos valores para los dos últimos parámetros, `name` y `age`.

Las dos variables definidas en el cuerpo del método `_init_()` tienen, cada una de ellas, el prefijo `self` ❸. Cualquier variable prefijada con `self` estará disponible para todos los métodos de la clase; podremos acceder a estas variables a través de cualquier instancia creada desde la clase. La línea `self.name = name` coge el valor asociado con el parámetro `name` y se lo asigna a la variable `name`, que entonces se une a la instancia que se está creando. El mismo proceso ocurre con `self.age = age`. Las variables a las que se accede mediante instancias como esta se denominan "atributos".

La clase `Dog` tiene otros dos métodos definidos: `sit()` y `roll_over()` ❹. Como estos métodos no necesitan información adicional para ejecutarse, simplemente los definiremos para que tengan un parámetro, `self`. Las instancias que creamos después tendrán acceso a estos métodos; en otras palabras, serán capaces de sentarse y hacer la croqueta. De momento, `sit()` y `roll_over()` no hacen gran cosa. Simplemente imprimen un mensaje diciendo que el perro se está sentando o haciendo la croqueta. Pero el concepto puede extenderse a situaciones realistas: si esta clase formase parte de un videojuego real, estos métodos contendrían código para hacer que

un perro animado se siente y haga la croqueta. Si esta clase se escribiese para controlar un robot, estos métodos dirigirían movimientos que harían que el perro-robot se sentase e hiciese la croqueta.

Hacer una instancia de una clase

Piense en una clase como un conjunto de instrucciones para crear una instancia. La clase `Dog` es una serie de instrucciones que dice a Python cómo crear instancias individuales para representar perros específicos.

Vamos a hacer una instancia que represente un perro concreto:

```
class Dog:  
    --fragmento omitido--  
1     my_dog = Dog('Willie', 6)  
  
2     print(f"My dog's name is {my_dog.name}.")  
3     print(f"My dog is {my_dog.age} years old.")
```

La clase `Dog` que estamos usando aquí es la que hemos escrito en el ejemplo anterior. Aquí le decimos a Python que cree un perro llamado `'Willie'` cuya edad es `6` **1**. Cuando Python lee esta línea, llama al método `__init__()` de `Dog` con los argumentos `'Willie'` y `6`. El método `__init__()` crea una instancia que representa a este perro en particular y configura los atributos `name` y `age` con los valores proporcionados. Después, Python devuelve una instancia que representa a este perro. Asignamos esa instancia a la variable `my_dog`. La convención de nomenclatura es útil aquí: podemos asumir que un nombre con inicial mayúscula como `Dog` se refiere a una clase y un nombre en minúsculas como `my_dog` se refiere a una sola instancia creada de una clase.

Acceder a los atributos

Para acceder a los atributos de una instancia, usamos la notación de punto. Accedemos al valor del atributo `name` de `my_dog` ❷ escribiendo:

```
my_dog.name
```

La notación de punto se usa con frecuencia en Python. Esta sintaxis muestra cómo Python busca el valor de un atributo. Aquí, se fija en la instancia `my_dog` y busca el atributo `name` asociado a `my_dog`. Es el mismo atributo al que se hace referencia como `self.name` en la clase `Dog`. Utilizamos el mismo enfoque para trabajar con el atributo `age` ❸.

La salida es un resumen de lo que sabemos sobre `my_dog`:

```
My dog's name is Willie.
```

```
My dog is 6 years old.
```

Llamadas a métodos

Después de crear una instancia a partir de la clase `Dog`, podemos usar la notación de punto para llamar a cualquier otro método definido en `Dog`. Vamos a hacer que nuestro perro se siente y haga la croqueta:

```
class Dog:  
    --fragmento omitido--  
  
my_dog = Dog('Willie', 6)  
my_dog.sit()  
my_dog.roll_over()
```

Para llamar a un método, damos el nombre de la instancia (en este caso, `my_dog`) y el método al que queremos llamar separados por un punto. Cuando Python lee `my_dog.sit()`, busca el método `sit()` en la clase `Dog` y ejecuta ese código. Python interpreta la línea `my_dog.roll_over()` de la misma manera.

Ahora Willie hace lo que le mandamos:

Willie is now sitting.

Willie rolled over!

Esta sintaxis es bastante útil. Cuando los atributos y métodos tienen nombres descriptivos, como `name`, `age`, `sit()` y `roll_over()`, podemos inferir fácilmente lo que se supone que hace un bloque de código, incluso aunque no lo hayamos visto nunca.

Crear múltiples instancias

Podemos crear todas las instancias que necesitemos a partir de una clase. Vamos a crear un segundo perro llamado `your_dog`:

```
class Dog:  
    --fragmento omitido--  
  
    my_dog = Dog('Willie', 6)  
    your_dog = Dog('Lucy', 3)  
  
    print(f"My dog's name is {my_dog.name}.")  
    print(f"My dog is {my_dog.age} years old.")  
    my_dog.sit()  
  
    print(f"\nYour dog's name is {your_dog.name}.")  
    print(f"Your dog is {your_dog.age} years old.")  
    your_dog.sit()
```

En este ejemplo, hemos creado un perro llamado Willie y una perra llamada Lucy. Cada uno es una instancia separada, con su propio conjunto de atributos y capaz de hacer el mismo conjunto de acciones:

My dog's name is Willie.

My dog is 6 years old.

Willie is now sitting.

Your dog's name is Lucy.

Your dog is 3 years old.

Lucy is now sitting.

Incluso aunque usásemos el mismo nombre y edad para el mismo perro, Python crearía una instancia independiente a partir de la clase `Dog`. Podemos hacer tantas instancias como queramos a partir de una clase, siempre y cuando le demos un nombre de variable único u ocupe un lugar único en una lista o un diccionario.

PRUÉBELO

- **9-1. Restaurante:** Haga una clase llamada `Restaurante`. El método `__init__()` para `Restaurante` debería albergar dos atributos: `nombre_restaurante` y `tipo_cocina`. Cree un método llamado `describir_restaurante()` que imprima estos dos datos, y un método llamado `abrir_restaurante()` que imprima un mensaje indicando que el restaurante está abierto.

Haga una instancia llamada `restaurante` a partir de su clase. Imprima los dos atributos por separado y luego llame a ambos métodos.

- **9-2. Tres restaurantes:** Empiece con la clase del ejercicio 9-1. Cree tres instancias diferentes a partir de ella y llame a `describir_restaurante()` para cada instancia.

- **9-3. Usuarios:** Cree una clase llamada `Usuario`. Cree dos atributos llamados `nombre` y `apellido` y otros que suelen guardarse en un perfil de usuario. Cree un método llamado `describir_usuario()` que imprima un resumen de la información del usuario. Cree otro método llamado `saludar_usuario()` que imprima un saludo personalizado para el usuario. Cree varias instancias que representen a distintos usuarios y llame a ambos métodos para cada usuario.

Trabajar con clases e instancias

Podemos usar clases para representar muchas situaciones del mundo real. Una vez que escriba una clase, pasará la mayor parte del tiempo trabajando con instancias creadas a partir de esa clase. Una de las primeras tareas que le conviene aprender es modificar los atributos asociados con una instancia en particular. Se pueden modificar directamente o escribiendo métodos que actualicen los atributos de formas específicas.

La clase Car

Vamos a escribir otra clase que represente un coche. Esta clase guardará información sobre el tipo de coche con el que vamos a trabajar y tendrá un método que resuma esa información:

car.py

```
class Car:  
    """Un simple intento de representar un coche."""  
  
    ❶ def __init__(self, make, model, year):  
        """Inicializa atributos para describir un coche."""  
        self.make = make  
        self.model = model  
        self.year = year  
  
    ❷ def get_descriptive_name(self):  
        """Devuelve un nombre descriptivo con el formato adecuado."""  
        long_name = f'{self.year} {self.make} {self.model}'  
        return long_name.title()  
  
    ❸ my_new_car = Car('audi', 'a4', 2019)  
    print(my_new_car.get_descriptive_name())
```

En la clase `Car`, definimos el método `__init__()` con el parámetro `self` primero ❶, igual que hemos hecho antes con la clase `Dog`. Le damos además otros tres parámetros: `make`, `model` y `year`, que representan la marca, el modelo y el año. El método `__init__()` coge estos parámetros y los asigna a los atributos que irán asociados con instancias creadas a partir de esta clase. Cuando creemos una nueva instancia de `Car`, tendremos que especificar marca, modelo y año para nuestra instancia.

Definimos un método llamado `get_descriptive_name()` ❷ que ponga el año, la marca y el modelo de un coche en una cadena que describa el coche con precisión. Esto nos evitará tener que imprimir el valor de cada atributo individualmente. Para trabajar con los valores de los atributos de este método, usamos `self.make`, `self.model` y `self.year`. Fuera de la clase, creamos instancia de la clase `Car` y se la asignamos a la variable `my_new_car` ❸. A continuación, llamamos a `get_descriptive_name()` para ver qué tipo de coche tenemos:

2024 Audi A4

Para que esta clase sea más interesante, vamos a añadir un atributo que cambie con el tiempo. Agregaremos un atributo que albergue el kilometraje total del coche.

Establecer un valor predeterminado para un atributo

Cuando creamos una instancia, podemos definir atributos sin pasarlos como parámetros. Estos atributos se pueden definir en el método `__init__()`, donde se les asigna un valor predeterminado.

Vamos a añadir un atributo llamado `odometer_reading` que siempre empiece con un valor de 0. También añadiremos un método `read_odometer()` que nos ayude a leer el cuentakilómetros de cada coche:

```
class Car:  
    def __init__(self, make, model, year):
```

```

    """Inicializa atributos para describir un coche."""

    self.make = make
    self.model = model
    self.year = year
❶    self.odometer_reading = 0

    def get_descriptive_name(self):
        --fragmento omitido--

❷    def read_odometer(self):
        """Imprime una oración que indica el kilometraje del coche."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()

```

Esta vez, cuando Python llama al método `__init__()` para crear una nueva instancia, guarda la marca, el modelo y el año como atributos, igual que en el ejemplo anterior. Luego crea un atributo llamado `odometer_reading` y establece su valor inicial en ❶. También tenemos un nuevo método llamado `read_odometer()` en ❷ que facilita la lectura del kilometraje del coche. Nuestro coche empieza con 0 millas:

```

2024 Audi A4
This car has 0 miles on it.

```

No hay muchos coches que se vendan con 0 millas en el cuentakilómetros, así que necesitamos una forma de cambiar el valor de este atributo.

Modificar el valor de un atributo

Podemos cambiar el valor de un atributo de tres maneras: directamente a través de una instancia, estableciendo el valor con un método o incrementando el valor (sumándole una cantidad dada) a través de un método. Veamos cada una de estas técnicas.

Modificar el valor de un atributo directamente

La forma más fácil de cambiar el valor de un atributo consiste en acceder directamente al atributo a través de una instancia. Aquí vamos a poner el cuentakilómetros a 23 directamente:

```
class Car:  
    --fragmento omitido--  
  
my_new_car = Car('audi', 'a4', 2024)  
print(my_new_car.get_descriptive_name())  
  
my_new_car.odometer_reading = 23  
my_new_car.read_odometer()
```

Usamos la notación de punto para acceder al atributo `odometer_reading` del coche y establecer su valor directamente. Esta línea dice a Python que coja la instancia `my_new_car`, busque el atributo `odometer_reading` asociado con ella y configure el valor de ese atributo como 23:

```
2024 Audi A4  
This car has 23 miles on it.
```

A veces, nos interesa acceder directamente a los atributos así, pero otras nos conviene escribir un método que actualice el valor por nosotros.

Modificar el valor de un atributo a través de un método

Puede ser útil tener métodos que actualicen ciertos atributos por nosotros. En lugar de acceder directamente al atributo, pasamos el

nuevo valor a un método que gestiona la actualización internamente.

Veamos un ejemplo con un método llamado `update_odometer()`:

```
class Car:  
    --fragmento omitido--  
  
    def update_odometer(self, mileage):  
        """Configura el kilometraje con el valor dado."""  
        self.odometer_reading = mileage  
  
    my_new_car = Car('audi', 'a4', 2024)  
    print(my_new_car.get_descriptive_name())  
  
❶ my_new_car.update_odometer(23)  
my_new_car.read_odometer()
```

La única modificación de `Car` es la adición de `update_odometer()`. Este método coge un valor de kilometraje y se lo asigna a `self.odometer_reading`. Usando la instancia `my_new_car`, llamamos a `update_odometer()` con `23` como argumento ❶. Con ello se establece el kilometraje en `23` y `read_odometer()` lo imprime:

2024 Audi A4

This car has 23 miles on it.

Podemos ampliar el método `update_odometer()` para que trabaje más cada vez que se modifique el cuentakilómetros. Vamos a añadir una lógica para asegurarnos de que nadie intenta manipular la lectura del cuentakilómetros:

```
class Car:  
    --fragmento omitido--  
  
    def update_odometer(self, mileage):  
        """
```

Configura el cuentakilómetros con el valor dado.

Rechaza el cambio si se intenta hacer retroceder el cuentakilómetros.

"""

```
❶ if mileage >= self.odometer_reading:  
    self.odometer_reading = mileage  
else:  
❷     print("You can't roll back an odometer!")
```

Ahora `update_odometer()` comprueba que la nueva lectura tenga sentido antes de modificar el atributo. Si el valor proporcionado para `mileage` es mayor o igual que el existente, `self.odometer_reading`, podemos actualizar el cuentakilómetros al nuevo kilometraje ❶. Si es inferior al existente, recibiremos un aviso de que no podemos hacer retroceder un cuentakilómetros ❷.

Aumentar el valor de un atributo a través de un método

A veces necesitamos incrementar el valor de un atributo en una cantidad determinada, en lugar de darle un valor completamente nuevo. Supongamos que compramos un coche usado y hacemos 100 millas con él entre que lo compramos y lo registramos. Aquí tenemos un método que nos permite pasar esta cantidad incremental y sumarla al valor de la lectura del cuentakilómetros:

```
class Car:  
  
    --fragmento omitido--  
  
    def update_odometer(self, mileage):  
        --fragmento omitido--  
  
    def increment_odometer(self, miles):  
        """Añade la cantidad dada a la lectura del cuentakilómetros."""
```

```
    self.odometer_reading += miles

❶ my_used_car = Car('subaru', 'outback', 2019)
print(my_used_car.get_descriptive_name())

❷ my_used_car.update_odometer(23_500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()
```

El nuevo método de `increment_odometer()` coge un número de millas y se lo suma a `self.odometer_reading`. En primer lugar, creamos un coche usado, `my_used_car` ❶. Configuramos su cuentakilómetros en 23.500 llamando a `update_odometer()` y pasándole `23_500` ❷. Por último, llamamos a `increment_odometer()` y le pasamos `100` para sumar las 100 millas que hemos recorrido desde que compramos el coche hasta que lo registramos:

```
2019 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

Podemos modificar fácilmente este método para que rechace incrementos negativos, de modo que nadie pueda usar esta función para manipular un cuentakilómetros.

Nota: Puede usar métodos como este para controlar cómo los usuarios de su programa actualizan valores como la lectura de un cuentakilómetros, pero cualquiera con acceso al programa puede poner el cuentakilómetros en el valor que quiera accediendo directamente al atributo. Una seguridad eficaz debe prestar extrema atención al detalle, además de hacer comprobaciones básicas como esta.

PRUÉBELO

- **9-4. Número servido:** Empiece con el programa del ejercicio 9-1. Añada un atributo llamado `número_servido` con un valor predeterminado de 0. Cree una instancia llamada `restaurante` a partir de esta clase. Imprima el número de clientes a los que ha servido el restaurante, cambie ese valor y vuelva a imprimirllo.

Añada un método llamado `establecer_número_servido()` que le permita configurar el número de clientes a los que se ha servido. Llámelo con un número nuevo y vuelva a imprimir el valor.

Añada un método llamado `incrementar_número_servido()` que le permita incrementar el número de clientes atendidos. Llámelo con cualquier número que pueda representar a cuántos clientes se ha servido en un día laborable normal.

- **9-5. Intentos de inicio de sesión:** Añada un atributo `intentos_inicio` a la clase `Usuario` del ejercicio 9-3. Escriba un método llamado `incrementar_intentos_inicio()` que aumente el valor de `intentos_inicio` en 1. Escriba otro método llamado `restablecer_intentos_inicio()` que restablezca el valor de `intentos_inicio` a 0.

Haga una instancia de la clase `Usuario` y llame varias veces a `incrementar_intentos_inicio()`. Imprima el valor de `intentos_inicio` para asegurarse de que se ha incrementado correctamente y luego llame a `restablecer_intentos_inicio()`. Vuelva a imprimir `intentos_inicio` para asegurarse de que se ha restablecido a 0.

Herencia

No siempre hace falta empezar desde cero para crear una clase. Si la clase que queremos hacer es una versión especializada de otra que hemos escrito, podemos usar la "herencia". Cuando una clase hereda de otra, coge los atributos y métodos de la primera. La clase original se denomina "clase base" y la nueva es la "clase derivada". La clase derivada puede heredar algunos o todos los atributos y

métodos de su clase base, pero también tiene libertad para definir nuevos atributos y métodos propios.

El método `__init__()` para una clase derivada

Cuando escribimos una nueva clase basada en otra existente, con frecuencia necesitaremos llamar al método `__init__()` de la clase base. Esto inicializará todos los atributos definidos en el método `__init__()` de la base y hará que estén disponibles para la derivada.

A modo de ejemplo, vamos a modelar un coche eléctrico. Se trata simplemente de un tipo concreto de coche, así que podemos basar nuestra nueva clase `ElectricCar` en la clase `Car` que ya tenemos. Así solo tendremos que escribir código para los atributos y comportamientos específicos de los coches eléctricos.

Empezaremos haciendo una versión sencilla de la clase `ElectricCar` que haga todo lo que hace `Car`:

`electric_car.py`

```
❶ class Car:

    """Un simple intento de representar un coche."""

    def __init__(self, make, model, year):
        """Inicializa atributos para describir un coche."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo correctamente formateado."""
        long_name = f'{self.year} {self.make} {self.model}'
        return long_name.title()
```

```

def read_odometer(self):
    """Imprimir una declaración mostrando el kilometraje del
    vehículo."""
    print(f"This car has {self.odometer_reading} miles on it.")

def update_odometer(self, mileage):
    """Configura la lectura del cuentakilómetros para el valor
    dado."""
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("You can't roll back an odometer!")

def increment_odometer(self, miles):
    self.odometer_reading += miles

❷ class ElectricCar(Car):
    """Representa aspectos de un coche propios de los vehículos
    eléctricos."""

❸ def __init__(self, make, model, year):
    """Inicializa los atributos de la clase base."""
❹     super().__init__(make, model, year)
❺     my_leaf = ElectricCar('nissan', 'leaf', 2024)
     print(my_leaf.get_descriptive_name())

```

Comenzaremos con `Car` ❶. Al crear una clase derivada, la base debe formar parte del archivo actual y aparecer antes que la derivada. A continuación, definimos la clase derivada `ElectricCar` ❷. El nombre de la clase base debe incluirse entre paréntesis en la definición de la clase derivada. El método `__init__()` coge la información necesaria para crear una instancia de `Car` ❸. La función

`super()` ❸ es una función especial que nos permite llamar a un método de la clase base. Esta línea dice a Python que llame al método `__init__()` de `Car`, que da a una instancia `ElectricCar` todos los atributos definidos en ese método. El nombre "super" viene de una convención de llamar a la clase base "superclase" y a la derivada "subclase".

Probamos si la herencia funciona bien intentando crear un coche eléctrico con el mismo tipo de información que facilitaríamos para crear un coche normal. Creamos una instancia de la clase `ElectricCar` y la asignamos a `my_leaf` ❹. Esta línea llama al método `__init__()` definido en `ElectricCar`, que a su vez dice a Python que llame al método `__init__()` definido en la clase base `Car`. Proporcionamos los argumentos `'nissan'`, `'leaf'` y `2024`.

Al margen de `__init__()`, no hay todavía atributos ni métodos que sean propios de un coche eléctrico. En este punto, solo nos estamos asegurando de que el coche eléctrico tiene los comportamientos apropiados de `Car`:

2024 Nissan Leaf

La instancia `ElectricCar` funciona igual que una instancia de `Car`, así que ya podemos empezar a definir atributos y métodos específicos para un coche eléctrico.

Definir atributos y métodos para la clase derivada

Cuando ya tenemos una clase derivada que hereda de una base, podemos añadir todos los atributos y métodos nuevos necesarios para diferenciar la clase derivada de la base.

Vamos a añadir un atributo propio de los coches eléctricos (una batería, por ejemplo) y un método que informe de ese atributo. Recogeremos el tamaño de la batería y escribiremos un método que escriba una descripción de la misma:

```
class Car:
```

--fragmento omitido--

```
class ElectricCar(Car):  
    """Representa aspectos de un coche propios de los vehículos  
    eléctricos."""  
  
    def __init__(self, make, model, year):  
        """  
        Inicializa atributos de la clase base.  
        Luego inicializa atributos propios de un coche eléctrico.  
        """  
        super().__init__(make, model, year)  
1        self.battery_size = 40  
  
2    def describe_battery(self):  
        """Imprime una frase que describe el tamaño de la batería."""  
        print(f"This car has a {self.battery_size}-kWh battery.")  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())  
my_leaf.describe_battery()
```

Añadimos un nuevo atributo `self.battery_size` y establecemos su valor inicial en `40` **1**. Este atributo se asociará con todas las instancias creadas a partir de la clase `ElectricCar`, pero no se asociará con ninguna de `Car`. También añadimos un método llamado `describe_battery()` que imprime información sobre la batería **2**. Cuando llamamos a este método, obtenemos una descripción claramente propia de un coche eléctrico:

```
2024 Nissan leaf  
This car has a 40-kWh battery.
```

Podemos especializar la clase `ElectricCar` sin límites. Podemos añadir todos los atributos y métodos que necesitemos para modelar un coche eléctrico hasta el nivel de precisión deseado. Un atributo o método que podría pertenecer a cualquier coche, y no ser específico de un eléctrico, debería ir en la clase `Car` y no en `ElectricCar`. Así, cualquiera que use la clase `Car` tendrá también esa funcionalidad disponible, y la clase `ElectricCar` solo contendrá código para la información y el comportamiento específicos de los vehículos eléctricos.

Anular métodos de la clase base

Podemos anular cualquier método de la clase base que no se ajuste a lo que intentamos modelar con la clase derivada. Para ello, definimos un método en la derivada con el mismo nombre que el método de la base que queremos anular. Python ignorará al método de la clase base y solo prestará atención al que hemos definido en la derivada.

Supongamos que la clase `Car` tuviera un método llamado `fill_gas_tank()`. Este método no tiene sentido para un vehículo eléctrico porque no requiere combustible, así que vamos a anularlo. Esta es una de las técnicas para hacerlo:

```
class ElectricCar(Car):
    --fragmento omitido--

    def fill_gas_tank(self):
        """Los coches eléctricos no tienen depósito de combustible."""
        print("This car doesn't have a gas tank!")
```

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 usamos la herencia, podemos hacer que las clases derivadas conserven lo que necesitamos de la base y anular todo lo demás.

Instancias como atributos

Cuando modele algo del mundo real en código, es posible que vaya añadiendo cada vez más detalles a una clase. Acabarás con una lista creciente de atributos y métodos que hacen que los archivos se alarguen. En estas situaciones, conviene saber que se puede escribir parte de una clase como una clase aparte. Podemos descomponer una clase grande en clases más pequeñas que funcionen juntas; este enfoque se conoce como "composición". Por ejemplo, si seguimos añadiendo detalles a la clase `ElectricCar`, podríamos descubrir que estamos añadiendo varios atributos y métodos específicos para la batería del coche. Cuando esto ocurre, podemos parar y mandar todos esos atributos y métodos a una clase nueva llamada `Battery`. Posteriormente podremos usar una instancia de `Battery` como atributo de la clase `ElectricCar`:

```
class Car:  
    --fragmento omitido--  
  
class Battery:  
    """Un simple intento de modelar una batería para un coche  
    eléctrico."""  
  
❶ def __init__(self, battery_size=40):  
    """Inicializa los atributos de la batería."""  
    self.battery_size = battery_size  
  
❷ def describe_battery(self):  
    """Imprime una frase que describe el tamaño de la batería."""  
    print(f"This car has a {self.battery_size}-kWh battery.")  
  
class ElectricCar(Car):  
    """Representa aspectos de un coche propios de los vehículos  
    eléctricos."""
```

```
def __init__(self, make, model, year):  
    """  
        Inicializa los atributos de la clase base.  
        Luego inicializa los atributos específicos de un coche  
        eléctrico.  
    """  
    super().__init__(make, model, year)  
③    self.battery = Battery()  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())  
my_leaf.battery.describe_battery()
```

Definimos una clase nueva llamada `Battery` que no herede de ninguna otra clase. El método `__init__()` **①** tiene un parámetro, `battery_size`, además de `self`. Se trata de un parámetro opcional que establece el tamaño de la batería en 75 si no se proporciona ningún valor. El método `describe_battery()` también se ha pasado a esta clase **②**.

En la clase `ElectricCar`, añadimos un atributo `self.battery` **③**. Esta línea le indica a Python que cree una instancia nueva de `Battery` (con un tamaño predeterminado de 40, ya que no especificamos un valor) y se la asigne al atributo `self.battery`. Esto ocurrirá cada vez que se llame al método `__init__()`; para cualquier instancia de `ElectricCar` se creará ahora una instancia de `Battery` automáticamente.

Creamos un coche eléctrico y lo asignamos a la variable `my_leaf`. Cuando queramos describir la batería, tendremos que usar el atributo `battery` del coche:

```
my_leaf.battery.describe_battery()
```

Esta línea dice a Python que mire la instancia `my_leaf`, busque su atributo `battery` y llame al método `describe_battery()` asociado con la

instancia de `Battery` asignada al atributo.

La salida es idéntica a la que hemos visto antes:

```
2024 Nissan Leaf  
This car has a 40-kWh battery.
```

Parece mucho trabajo extra, pero ahora podemos describir la batería con todo el detalle que queramos sin saturar la clase `ElectricCar`. Vamos a añadir a `Battery` otro método que indique la autonomía del coche basándose en el tamaño de la batería:

```
class Car:  
    --fragmento omitido--  
  
class Battery:  
    --fragmento omitido--  
  
    def get_range(self):  
        """Imprime una frase sobre la autonomía que ofrece esta  
        batería."""  
        if self.battery_size == 40:  
            range = 150  
        elif self.battery_size == 65:  
            range = 225  
  
        print(f"This car can go about {range} miles on a full charge.")  
  
class ElectricCar(Car):  
    --fragmento omitido--  
  
    my_leaf = ElectricCar('nissan', 'leaf', 2024)  
    print(my_leaf.get_descriptive_name())  
    my_leaf.battery.describe_battery()  
❶    my_leaf.battery.get_range()
```

El nuevo método `get_range()` realiza un análisis sencillo. Si la capacidad de la batería es 40 kWh, `get_range()` establece la autonomía en 150 millas; si la capacidad es de 65 kWh, establece una autonomía de 225. Posteriormente, informa de este valor. Cuando queramos usar este método, tenemos que llamarlo a través del atributo `battery` del coche ❶.

La salida nos informa de la autonomía del coche basándose en el tamaño de su batería:

```
2024 Nissan Leaf
This car has a 40-kWh battery.
This car can go about 150 miles on a full charge.
```

Modelar objetos del mundo real

Cuando se empieza a modelar cosas más complejas, como coches eléctricos, se plantean cuestiones interesantes. ¿Es la autonomía una propiedad de la batería o del coche? Si solo vamos a describir un coche, seguramente está bien mantener la asociación del método `get_range()` con la clase `Battery`. Pero, si vamos a describir la línea entera de un fabricante, seguramente nos conviene mover `get_range()` a la clase `ElectricCar` class. El método `get_range()` seguiría comprobando el tamaño de la batería para determinar la autonomía, pero informaría de una autonomía específica para el tipo de coche con el que está asociado. Otra opción sería mantener la asociación de `get_range()` con la batería, pero pasarle un parámetro como `car_model`. El método `get_range()` informaría entonces de la autonomía basándose en el tamaño de la batería y el modelo del coche.

Esto le lleva a un punto interesante en su crecimiento como programador. Al enfrentarse a preguntas como estas, tendrá que pensar a un nivel lógico más alto, en lugar de centrarse en la sintaxis. No estará pensando en Python, sino en cómo representar el mundo real con código. Cuando llegue a este punto, se dará cuenta de que, a menudo, no hay enfoques ni buenos ni malos para

modelar situaciones del mundo real. Algunos enfoques son más eficaces que otros, pero requiere práctica encontrar las representaciones más eficientes. Si su código funciona como quiere, ¡lo está haciendo bien! No se desanime si tiene que desmontar y reescribir sus clases varias veces usando distintos enfoques. Todo el mundo pasa por ese proceso hasta conseguir escribir código preciso y eficiente.

PRUÉBELO

- **9-6. Carrito de helados:** Un carrito de helados es, en cierto modo, parecido a un restaurante. Escriba una clase llamada `CarritoDeHelados` que herede de la clase `Restaurante` del ejercicio 9-1 o del 9-4. Servirá cualquiera de las dos versiones, así que coja la que más le guste. Añada un atributo llamado `sabores` que almacene una lista de sabores de helado. Escriba un método que muestre los sabores. Cree una instancia de `CarritoDeHelados` y llame a ese método.
- **9-7. Admin:** Un administrador es un tipo especial de usuario. Escriba una clase llamada `Admin` que herede de la clase `Usuario` del ejercicio 9-3 o del 9-5. Añada un atributo `privilegios` que acoja una lista de cadenas como "puede añadir comentario", "puede borrar comentario", "puede vetar usuarios", etc. Escriba un método llamado `show_privileges()` que enumere el conjunto de privilegios del administrador. Cree una instancia de `Admin` y llame al método.
- **9-8. Privilegios:** Escriba una clase `Privilegios` aparte. Esta clase debería tener un atributo, `privilegios`, que almacene una lista de cadenas como la descrita en el ejercicio anterior. Mueva el método `show_privileges()` a esta clase. Haga una instancia de `Privilegios` como atributo de la clase `Admin`. Cree una nueva instancia de `Admin` y use su método para mostrar los privilegios.
- **9-9. Mejora de Battery:** Use la última versión de `electric_car.py`. Añada un método a la clase `Battery` llamado `upgrade_battery()`. Este método debería comprobar el tamaño de la batería y establecer la capacidad en 65 si no está ya así. Haga un coche eléctrico con un tamaño

de batería predeterminado, llame a `get_range()` una vez y vuelva a llamarlo para mejorar la batería. Debería ver un incremento en la autonomía del coche.

Importar clases

A medida que añadimos funcionalidad a las clases, los archivos se pueden hacer muy largos, incluso si utilizamos adecuadamente la herencia y composición. En consonancia con la filosofía de Python, nos interesa que nuestros archivos estén lo menos abarrotados posible. Para ayudarnos, Python nos permite almacenar clases en módulos para luego importar las que necesitemos a nuestro programa principal.

Importar una sola clase

Vamos a crear un módulo que contenga solo la clase `Car`. Esto provoca un problemilla de nomenclatura: ya tenemos un archivo llamado `car.py` en este capítulo, pero este módulo debería llamarse `car.py` porque contiene código que representa un coche. Resolveremos este problema guardando la clase `Car` en un módulo `car.py`, reemplazando el archivo `car.py` que usábamos antes. A partir de ahora, cualquier programa que use este módulo necesitará un nombre de archivo más específico, como `my_car.py`. Aquí está `car.py` con el código de la clase `Car` únicamente:

`car.py`

① `"""Una clase que se puede usar para representar un coche."""`

```
class Car:  
    """Un simple intento de representar un coche."  
  
    def __init__(self, make, model, year):
```

```

"""Inicializa atributos para describir un coche."""

self.make = make
self.model = model
self.year = year
self.odometer_reading = 0

def get_descriptive_name(self):
    """Devuelve un nombre descriptivo con un formato adecuado."""
    long_name = f"{self.year} {self.make} {self.model}"
    return long_name.title()

def read_odometer(self):
    """Imprime una frase que indica el kilometraje del coche."""
    print(f"This car has {self.odometer_reading} miles on it.")

def update_odometer(self, mileage):
    """
    Establece la lectura del cuentakilómetros en el valor dado.
    Rechaza el cambio si intenta hacer retroceder el
    cuentakilómetros.
    """
    if mileage >= self.odometer_reading:
        self.odometer_reading = mileage
    else:
        print("You can't roll back an odometer!")

def increment_odometer(self, miles):
    """Suma la cantidad dada a la lectura del cuentakilómetros."""
    self.odometer_reading += miles

```

Incluimos una cadena de documentación a nivel de módulo que describe brevemente el contenido del módulo ❶. Debería escribir

una para cada módulo que cree.

Ahora vamos a crear un archivo aparte llamado `my_car.py`. Este archivo importará la clase `Car` y creará una instancia a partir de ella:

`my_car.py`

```
❶ from car import Car

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23
my_new_car.read_odometer()
```

La sentencia `import` ❶ le dice a Python que abra el módulo `car` e importe la clase `Car`. Ahora podemos usar esta clase como si estuviese definida en este archivo. La salida es la misma de antes:

```
2024 Audi A4
This car has 23 miles on it.
```

Importar clases es una forma efectiva de programar. Imagine lo largo que sería este archivo de programa si se incluyese toda la clase `Car`. Si en vez de eso movemos la clase a un módulo y lo importamos, tendremos la misma funcionalidad, pero el archivo principal de nuestro programa estará más limpio y será más fácil de leer. También guardamos la mayoría de la lógica en archivos aparte; cuando las clases ya funcionen como queríamos, podemos dejar quietos esos archivos y centrarnos solo en la lógica de alto nivel de nuestro programa,

Almacenar varias clases en un módulo

Podemos guardar todas las clases que necesitemos en un solo módulo, aunque todas las clases de un módulo deberían estar relacionadas de alguna manera. Las clases `Battery` y `ElectricCar` sirven para representar coches, así que podemos añadirlas al módulo `car.py`.

`car.py`

```
"""Un conjunto de clases que sirven para representar coches de gasolina y eléctricos."""
```

```
class Car:  
    --fragmento omitido--  
  
class Battery:  
    """Un simple intento de modelar una batería para un coche eléctrico."""  
  
    def __init__(self, battery_size=40):  
        """Inicializa los atributos de la batería."""  
        self.battery_size = battery_size  
  
    def describe_battery(self):  
        """Imprime una frase que describe el tamaño de la batería."""  
        print(f"This car has a {self.battery_size}-kWh battery.")  
  
    def get_range(self):  
        """ Imprime una frase sobre la autonomía que ofrece esta batería."""  
        if self.battery_size == 40:  
            range = 150  
        elif self.battery_size == 65:  
            range = 225  
  
        print(f"This car can go about {range} miles on a full charge.")  
  
class ElectricCar(Car):
```

```
"""Modela aspectos de un coche que son propios de los vehículos eléctricos."""
```

```
def __init__(self, make, model, year):  
    """  
    Inicializa atributos de la clase base.  
    Luego inicializa atributos específicos de un coche eléctrico.  
    """  
    super().__init__(make, model, year)  
    self.battery = Battery()
```

Ahora podemos crear un nuevo archivo llamado `my_electric_car.py`, importar la clase `ElectricCar` y crear un coche eléctrico:

`my_electric_car.py`

```
from car import ElectricCar  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())  
my_leaf.battery.describe_battery()  
my_leaf.battery.get_range()
```

La salida será la misma que hemos visto antes, aunque la mayor parte de la lógica está oculta en un módulo:

```
2024 Nissan Leaf  
This car has a 40-kWh battery.  
This car can go about 150 miles on a full charge.
```

Importar varias clases desde un módulo

Podemos importar tantas clases como necesitemos a un archivo de programa. Si queremos hacer un coche convencional y uno

eléctrico en un mismo archivo, tendremos que importar las dos clases, `Car` y `ElectricCar`:

my_cars.py

```
❶ from car import Car, ElectricCar

❷ my_mustang = Car('ford', 'mustang', 2024)
    print(my_mustang.get_descriptive_name())

❸ my_leaf= ElectricCar('nissan', 'leaf', 2024)
    print(my_leaf.get_descriptive_name())
```

Para importar varias clases desde un módulo, las separaremos con comas ❶. Una vez importadas las clases que necesitamos, podemos crear todas las instancias que queramos de cada una.

En este ejemplo vamos a crear un Ford Mustang de gasolina ❷ y a continuación un Nissan Leaf eléctrico ❸:

2024 Ford Mustang

2024 Nissan Leaf

Importar un módulo entero

También se puede importar un módulo entero y acceder a las clases necesarias con la notación de punto. Es una aproximación sencilla que tiene como resultado código fácil de leer. Como cada llamada que crea una instancia de una clase incluye el nombre del módulo, no habrá conflictos con los nombres usados en el archivo actual.

Así importaríamos el módulo `car` entero para crear después un coche convencional y otro eléctrico:

my_cars.py

```
❶ import car

❷ my_mustang = car.Car('ford', 'mustang', 2024)
print(my_mustang.get_descriptive_name())

❸ my_leaf = car.ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

Importamos el módulo `car` completo ❶. A continuación, accedemos a las clases que necesitamos con la sintaxis `nombre_módulo.NombreClase`. Volvemos a crear un Ford Mustang ❷ y un Nissan Leaf ❸.

Importar todas las clases de un módulo

Podemos importar todas las clases de un módulo con esta sintaxis:

```
from nombre_módulo import *
```

Este método no es recomendable por dos motivos. Primero, conviene poder leer las sentencias `import` en la parte superior de un archivo para hacernos una mejor idea de las clases que usa un programa. Con este enfoque, no está claro qué clases del módulo estamos usando. Además, puede haber confusiones con los nombres en el archivo. Si importamos accidentalmente una clase con el mismo nombre que otro elemento del programa, crearemos errores difíciles de diagnosticar. Mencione esta técnica aquí, aunque no es recomendable, porque es probable que tarde o temprano la encuentre en el código de otros programadores.

Si necesita importar varias clases de un módulo, es mejor importar todo el módulo y usar la sintaxis `nombre_módulo.NombreClase`. No verá todas las clases usadas en la parte superior del archivo, pero sí verá claramente dónde se usa el módulo en el programa. También

evitará el problema de los conflictos entre nombres que pueden surgir al importar todas las clases del módulo.

Importar un módulo en otro módulo

A veces conviene dividir las clases en varios módulos para evitar que los archivos se hagan demasiado grandes y que queden clases no relacionadas en un mismo módulo. Al guardar las clases en varios módulos, puede ocurrir que una clase en un módulo dependa de otra que esté en otro módulo. En ese caso, podemos importar la clase necesaria al primer módulo.

Por ejemplo, vamos a guardar la clase `Car` en un módulo y las clases `ElectricCar` y `Battery` en otro. Haremos un nuevo módulo llamado `electric_car.py` (que sustituya al archivo `electric_car.py` que hemos creado antes) y copiaremos solo las clases `Battery` y `ElectricCar` en este archivo:

`electric_car.py`

```
"""Un conjunto de clases que sirven para representar coches eléctricos."""
```

```
from car import Car

class Battery:
    --fragmento omitido--

class ElectricCar(Car):
    --fragmento omitido--
```

La clase `ElectricCar` necesita acceder a su clase base, `Car`, así que importamos `Car` directamente al módulo. Si olvidamos esta línea, Python dará error cuando intentemos importar el módulo `electric_car`. También tenemos que actualizar el módulo `car` para que contenga solo la clase `Car`:

`car.py`

```
"""Una clase que sirve para representar un coche."""
```

```
class Car:  
    --fragmento omitido--
```

Ahora podemos importar cada módulo por separado y crear el tipo de coche que necesitemos:

my_cars.py

```
from car import Car  
from electric_car import ElectricCar  
  
my_mustang = Car('ford', 'mustang', 2024)  
print(my_mustang.get_descriptive_name())  
  
my_leaf = ElectricCar('nissan', 'leaf', 2024)  
print(my_leaf.get_descriptive_name())
```

Importamos `Car` desde su módulo y `ElectricCar` desde el suyo. Luego creamos un coche convencional y uno eléctrico. Ambos tipos de coche se crean correctamente:

```
2024 Ford Mustang  
2024 Nissan Leaf
```

Usar alias

Como vimos en el capítulo 8, los alias pueden ser bastante útiles cuando usamos módulos para organizar el código de nuestros proyectos. Podemos usarlos también para importar clases.

A modo de ejemplo, piense en un programa en el que queremos hacer un montón de coches eléctricos. Sería una pesadez escribir (y leer) `ElectricCar` una y otra vez. Podemos dar a `ElectricCar` un alias en la sentencia `import`:

```
from electric_car import ElectricCar as EC
```

Ahora podemos usar este alias siempre que queramos crear un coche eléctrico:

```
my_leaf = EC('nissan', 'leaf', 2024)
```

También podemos dar un alias al módulo. Veamos cómo importar el módulo `electric_car` al completo usando un alias:

```
import electric_car as ec
```

Ahora podemos usar este alias de módulo con el nombre de clase completo:

```
my_leaf = ec.ElectricCar ('nissan', 'leaf', 2024)
```

Encontrar su propio flujo de trabajo

Como ve, Python ofrece muchas opciones a la hora de estructurar el código de un proyecto grande. Es importante conocer estas posibilidades para decidir la mejor forma de organizar sus proyectos y para comprender los proyectos de otras personas. Cuando se empieza, es mejor mantener la estructura del código sencilla. Pruebe a hacer todo en un archivo y mover las clases a módulos aparte cuando todo funcione. Si le gusta cómo interactúan los módulos y los archivos, pruebe a guardar sus clases en módulos cuando empiece un proyecto. Busque un enfoque que le permita escribir código que funcione y siga a partir de ahí.

PRUÉBELO

- **9-10. Restaurante importado:** Use la última clase `Restaurante` y guárdela en un módulo. Cree un archivo aparte que importe `Restaurante`.

Haga una instancia `Restaurante` y llame a uno de los métodos de `Restaurante` para comprobar que la sentencia `import` funciona.

- **9-11. Admin importado:** Empiece con el trabajo del ejercicio 9-8. Guarde las clases `Usuario`, `Privilegios` y `Admin` en un módulo. Cree un archivo aparte, haga una instancia `Admin` y llame a `show_privileges()` para comprobar que todo funciona correctamente.
- **9-12. Múltiples módulos:** Guarde la clase `Usuario` en un módulo y las clases `Privilegios` y `Admin` en otro. En un archivo aparte, cree una instancia `Admin` y llame a `show_privileges()` para ver si todo sigue funcionando.

La biblioteca estándar de Python

La "biblioteca estándar de Python" es un conjunto de módulos incluidos con la instalación de Python. Ahora que tiene un conocimiento básico del funcionamiento de funciones y clases, puede empezar a usar módulos como estos, escritos por otros programadores. Puede usar cualquier función o clase de la biblioteca estándar simplemente incluyendo una sentencia `import` al principio de su archivo. Vamos a echar un vistazo a un módulo, `random`, que puede ser útil para modelar muchas situaciones de la vida real.

Una función interesante de este módulo es `randint()`. Esta función coge dos argumentos enteros y devuelve un entero seleccionado aleatoriamente entre esos números (incluidos ambos). Así es como generaríamos un número aleatorio entre el 1 y el 6:

```
>>> from random import randint  
>>> randint(1, 6)  
3
```

Otra función útil es `choice()`, que admite una lista o tupla y devuelve un elemento seleccionado al azar:

```
>>> from random import choice  
>>> players = ['charles', 'martina', 'michael', 'florence', 'eli']
```

```
>>> first_up = choice(players)
>>> first_up
'florence'
```

El módulo `random` no debería usarse cuando se crean aplicaciones relacionadas con la seguridad, pero es adecuado para muchos proyectos divertidos e interesantes.

Nota: También puede descargar módulos de fuentes externas. Veremos varios ejemplos en la segunda parte del libro, donde necesitaremos módulos externos para completar cada proyecto.

PRUÉBELO

- **9-13. Dados:** Haga una clase `Dados` con un atributo llamado `caras`, que tenga un valor predeterminado de 6. Escriba un método llamado `tirar_dado()` que imprima un número aleatorio entre 1 y el número de caras que tenga el dado. Haga un dado de 6 caras y tirelo 10 veces. Haga un dado de 10 caras y otro de 20. Lance cada dado 10 veces.
- **9-14. Lotería:** Cree una lista o tupla que contenga series de 10 números y 5 letras. Seleccione aleatoriamente cuatro números o letras de la lista e imprima un mensaje diciendo que cualquier boleto con estos cuatro números o letras está premiado.
- **9-15. Análisis de la lotería:** Puede usar un bucle para ver lo difícil que sería ganar el tipo de lotería que acaba de modelar. Haga una lista o tupla llamada `mi_boleto`. Escriba un bucle que saque números hasta que gane su boleto. Imprima un mensaje que indique cuántas veces ha tenido que ejecutarse el bucle hasta que ha salido el número ganador.
- **9-16. Módulo de Python de la semana:** Un recurso excelente para explorar la biblioteca estándar de Python es un sitio web llamado *Python Module of the Week*. Vaya a <https://pymotw.com/> y eche un vistazo a la lista de contenidos. Busque un módulo que le parezca interesante y lea sobre él. Puede empezar por el módulo `random`.

Dar estilo a las clases

Merece la pena aclarar algunas cuestiones estilísticas de las clases, sobre todo para cuando sus programas se vuelvan más complicados. Los nombres de clase deberían escribirse siguiendo la convención de nomenclatura CamelCase, es decir, con mayúscula inicial en cada palabra y sin guiones. Los nombres de instancias y módulos deberían escribirse en minúsculas y con guiones bajos entre palabras.

Cada clase debería tener una cadena de documentación justo después de la definición. Este comentario debería ser una breve descripción de lo que hace la clase y debería seguir las mismas convenciones de formato usadas para las cadenas de documentación de las funciones. Cada módulo debería contar también con una cadena de documentación explicando para qué sirven las clases del módulo.

Puede usar líneas en blanco para organizar el código, pero no abuse. Dentro de una clase, puede usar una línea en blanco entre métodos y, dentro de un módulo, puede usar dos líneas en blanco para separar clases.

Si necesita importar un módulo de la biblioteca estándar y otro suyo, coloque primero la sentencia `import` para el de la biblioteca estándar. Después, añada una línea en blanco y la sentencia `import` para su módulo. En programas con varias sentencias `import`, esta convención hace que sea más fácil ver de dónde vienen los distintos módulos usados en el programa.

Resumen

En este capítulo ha aprendido a escribir sus propias clases. Ya sabe cómo guardar información en una clase usando atributos y cómo escribir métodos que den a sus clases el comportamiento que necesitan. Ha aprendido a escribir métodos `__init__()` que crean instancias para sus clases con los atributos que quiere exactamente. También hemos visto cómo modificar los atributos de una instancia

directamente y a través de métodos. Ha descubierto que la herencia puede simplificar la creación de clases relacionadas entre sí y que podemos usar instancias de una clase como atributos de otra para mantener las clases simples.

Hemos visto cómo guardar clases en módulos e importar las que necesitemos en los archivos en los que se usarán nos ayuda a mantener nuestros proyectos organizados. Ha conocido la biblioteca estándar de Python y ha visto un ejemplo basado en el módulo `random`. Por último, ha aprendido a dar estilo a sus clases con las convenciones de Python.

En el capítulo 10, veremos cómo trabajar con archivos para poder guardar nuestro trabajo en un programa y el trabajo que hemos dejado hacer a otros. También hablaremos de las excepciones, una clase especial de Python diseñada para ayudarnos a responder a errores cuando surjan.

10

ARCHIVOS Y EXCEPCIONES



Ahora que domina las destrezas básicas necesarias para escribir programas organizados que sean fáciles de usar, ha llegado el momento de pensar en programas más relevantes y usables. En este capítulo, aprenderá a trabajar con archivos para que sus programas puedan analizar con rapidez muchos datos.

Aprenderá a manejar errores para que sus programas no fallen al encontrarse con situaciones inesperadas. También aprenderá sobre "excepciones", que son objetos especiales que crea Python para gestionar errores que surjan cuando se está ejecutando un programa. Por último, descubrirá el módulo `json`, que permite guardar datos de usuario para que no se pierdan cuando el programa deje de ejecutarse. Aprender a trabajar con archivos y guardar datos hará que sus programas sean más fáciles de utilizar. Los usuarios podrán elegir qué datos introducir y cuándo hacerlo. Podrán ejecutar su programa, trabajar con él y cerrarlo para continuar más adelante en el punto donde lo dejaron. Aprender a manejar excepciones le ayudará a lidiar con situaciones en las que no existe un archivo y con otros problemas que pueden hacer que un programa falle. Esto hará que sus programas sean más resistentes cuando encuentren datos malos, ya provengan estos de errores inocentes o de intentos maliciosos de estropear el programa. Con las habilidades que adquirirá en este capítulo, hará sus programas más aplicables, usables y estables.

Leer de un archivo

Los archivos de texto ofrecen una cantidad increíble de datos. Estos archivos pueden contener información meteorológica, datos de tráfico, datos socioeconómicos, obras literarias y mucho más. Leer de un archivo es especialmente útil en aplicaciones de análisis de datos, pero también es aplicable a cualquier situación en la que haya que analizar o modificar información guardada en un archivo. Por ejemplo, podemos escribir un programa que lea el contenido de un archivo de texto y reescriba el archivo con un formato que permita que lo muestre un navegador.

Cuando quiera trabajar con la información contenida en un archivo de texto, el primer paso será leer el archivo en memoria. A continuación, podrá leer todos los contenidos del archivo o trabajarlos línea a línea.

Leer los contenidos de un archivo

Para empezar, necesitamos un archivo con unas cuantas líneas de texto. Comenzaremos con uno que contiene el número pi hasta 30 decimales, con 10 posiciones decimales por línea:

pi_digits.txt

3.1415926535
8979323846
2643383279

Para probar este ejemplo, puede escribir estas líneas en un editor y guardar el archivo como `pi_digits.txt` o puede descargarlo de los recursos del libro. Guarde el archivo en el mismo directorio donde guardará los programas de este capítulo.

El siguiente programa abre este archivo, lo lee e imprime el contenido en la pantalla:

file_reader.py

```
from pathlib import Path

❶ path = Path('pi_digits.txt')
❷ contents = path.read.text()

print(contents)
```

Para trabajar con los contenidos de un archivo, debemos indicarle a Python la ruta al archivo. La ruta es la ubicación exacta del archivo o carpeta en el sistema. Python ofrece un módulo llamado `pathlib` que nos hace las cosas más fáciles a la hora de trabajar con archivos y directorios, independientemente del sistema operativo que utilicen usted o los usuarios de sus programas.

Comenzamos importando la clase `Path` desde `pathlib`. Podemos hacer muchas cosas con un objeto `Path` que apunta a un archivo. Por ejemplo, podemos comprobar que el archivo existe antes de trabajar con él, leer los contenidos del archivo o escribir nuevos datos en el mismo. En esta ocasión, vamos a crear un objeto `Path` que represente al archivo `pi_digits.txt`, que asignaremos a la variable `path` ❶. Dado que este archivo se guarda en el mismo directorio que el archivo `.py` que estamos escribiendo, el nombre de archivo es todo lo que `Path` necesita para acceder al archivo.

Nota: VS Code buscará los archivos en la última carpeta abierta. Si utiliza VS Code, abra la carpeta donde se almacenan los programas de este capítulo. Por ejemplo, si guarda sus archivos de programa en una carpeta llamada `capítulo_10`, pulse **Control-O** o **Comando-O** (en MacOS) para abrir dicha carpeta.

Una vez tengamos un objeto `Path` que representa `pi_digits.txt`, usamos el método `read()` para leer todos los contenidos del archivo ❷. Los contenidos del archivo se devuelven como una cadena simple que asignaremos a la variable `contents`. Al imprimir el valor de `contents`, obtenemos el archivo de texto completo:

8979323846

2643383279

La única diferencia entre esta salida y el archivo original es la línea en blanco extra al final de la salida. Esta línea aparece porque `read_text()` devuelve una cadena vacía cuando llega al final del archivo; esta cadena vacía se muestra como una línea en blanco.

Podemos eliminar esta línea en blanco de más usando `rstrip()` en la cadena `contents`:

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
contents = contents.rstrip()
print(contents)
```

Recuerde, como vimos en el capítulo 2, que el método `rstrip()` de Python elimina cualquier carácter en blanco a la derecha de una cadena. Ahora la salida coincide exactamente con el contenido del archivo original.

3.1415926535

8979323846

2643383279

Podemos eliminar el carácter del final de la cadena al leer los contenidos del archivo aplicando el método `rstrip()` inmediatamente después de llamar a `read_text()`:

```
contents = path.read_text().rstrip()
```

Esta línea de código le dice a Python que llame al método `read_text()` del archivo con el que estamos trabajando. A continuación aplica el método `rstrip()` a la cadena que devuelve `read_text()`. La cadena resultante, ya limpia, es entonces asignada a la variable

`contents`. Este enfoque recibe el nombre de encadenamiento de métodos, y lo verá frecuentemente en programación.

Rutas de archivo relativas y absolutas

Cuando pasamos un nombre de archivo simple como `pi_digits.txt` a `Path`, Python busca en el directorio dónde se almacena el archivo en ejecución (es decir, el archivo de programa `.py`).

En ocasiones, dependiendo de cómo organicemos el trabajo, el archivo que queremos abrir no estará en el mismo directorio que el archivo de programa. Por ejemplo, podríamos guardar los archivos de programa en una carpeta llamada `python_work` y, dentro de ella, tener una subcarpeta `text_files` para distinguir los archivos de programa de los archivos de texto que van a manipular. Aunque `text_files` esté en `python_work`, pasar a `open()` solo el nombre de un archivo de `text_files` no funcionará, ya que Python buscará en `python_work` y se detendrá ahí; no seguirá buscando dentro de `text_files`. Para que Python abra archivos de un directorio distinto a aquel donde se aloja nuestro archivo, tendremos que proporcionar la ruta correcta. Existen dos maneras principales para especificar las rutas en programación. Una ruta de archivo relativa le indica a Python que busque una ubicación dada en relación al directorio donde se almacena el programa que se está ejecutando en ese momento. Dado que `text_files` está dentro de `python_work`, tendremos que crear una ruta que comience con el directorio `text_files` y termine con el nombre de archivo. Veamos cómo crear esta ruta:

```
path = Path('text_files/filename.text')
```

Además, podemos indicarle a Python la ubicación exacta del archivo en nuestro ordenador, independientemente de dónde se ubique el programa en ejecución. Esto es lo que se conoce como ruta de archivo absoluta. Usamos una ruta absoluta cuando no funcione una ruta relativa. Por ejemplo, si hemos colocado `text_files` en una carpeta distinta a `python_work`, entonces pasar `Path` a la ruta

'text_files/*filename.text*' no funcionará porque Python únicamente buscará esa ubicación dentro de `python_work`. Tendremos que escribir una ruta completa para aclarar dónde queremos que Python busque. Las rutas absolutas suelen ser más largas que las relativas, puesto que comienzan en la carpeta raíz de nuestro sistema:

```
path = Path('/home/eric/data_files/text_files/filename.txt')
```

Usando rutas absolutas podremos leer archivos de cualquier ubicación del sistema, pero por ahora es más fácil guardar los archivos en el mismo directorio que los archivos del programa, o bien en una subcarpeta dentro del directorio que almacene los archivos, como `text_files`.

Nota: Windows emplea el símbolo de barra invertida (\) en lugar de la barra inclinada (/) a la hora de mostrar las rutas de archivo, pero debería poder utilizar ambos tipos de barras en su código, incluso en Windows. La biblioteca `pathlib` usará automáticamente la representación de la ruta cuando interactúe con su sistema o con cualquier sistema del usuario.

Acceder a las líneas de un archivo

Al trabajar con un archivo, necesitaremos analizarlo línea a línea con cierta frecuencia. Puede que busquemos cierta información o que tengamos que modificar el texto de alguna forma. Por ejemplo, podríamos querer leer un archivo con datos meteorológicos y trabajar con una línea que incluya la palabra "soleado" en la descripción del tiempo de ese día. En una noticia, podríamos buscar cualquier línea con la etiqueta <headline> para reescribirla con un tipo de formato especial.

Podemos usar el método `splitlines()` para convertir una cadena larga en un conjunto de líneas, y a continuación usar el bucle `for` para analizar todas las líneas del archivo, una por una:

`file_reader.py`

```
from pathlib import Path

path = Path('pi_digits.txt')
❶ contents = path.read_text()

❷ lines = contents.splitlines()
for line in lines:
    print(line)
```

Comenzamos leyendo los contenidos del archivo, como ya habíamos hecho anteriormente ❶. Si tiene pensado trabajar con un archivo línea por línea, no es necesario eliminar los espacios en blanco al leer dicho archivo. El método `splitlines()` devuelve una lista con todas las líneas del archivo. Podemos asignar esta lista a las variables `lines` ❷. Después pasamos en bucle por estas líneas e imprimimos cada una de ellas:

```
3.1415926535
8979323846
2643383279
```

Dado que no hemos modificado ninguna de las líneas, la salida coincide exactamente con el archivo de texto original.

Trabajar con el contenido de un archivo

Una vez leído un archivo en memoria, podemos hacer lo que queramos con esos datos, así que vamos a explorar brevemente los dígitos de pi. Primero, intentaremos crear una sola cadena que contenga todos los números del archivo, sin espacios en blanco:

pi_string.py

```
from pathlib import Path
```

```
path = Path('pi_digits.txt')

contents = path.read_text()

lines = contents.splitlines()

pi_string = ''

❶ for line in lines:

    pi_string += line

print(pi_string)

print(len(pi_string))
```

Comenzamos leyendo el archivo y almacenando cada línea de números en una lista, al igual que en el ejemplo anterior. A continuación, creamos una variable, `pi_string`, para guardar los dígitos de pi ❶. Imprimimos esta cadena y mostramos lo larga que es:

3.1415926535 8979323846 2643383279

36

La variable `pi_string` contiene el espacio en blanco que había a la izquierda de los números de cada línea, pero podemos deshacernos de él usando `rstrip()` en cada línea:

```
--fragmento omitido--

for line in lines:

    pi_string += line.lstrip()

print(pi_string)

print(len(pi_string))
```

Ahora tenemos una cadena que contiene pi hasta 30 decimales. La cadena tiene 32 caracteres porque también incluye el 3 y el punto decimal:

Nota: Cuando Python lee de un archivo de texto, interpreta todo el archivo como una cadena. Si leemos en un número y queremos trabajar con ese valor en un contexto numérico, tendremos que convertirlo en entero con la función `int()` o en flotante con la función `float()`.

Archivos grandes: un millón de números

Hasta ahora nos hemos centrado en analizar un archivo de texto que contiene solo tres líneas, pero el código de estos ejemplos también funcionaría con archivos mucho más largos. Si empezamos con un archivo de texto que contenga pi hasta un millón de decimales en vez de solo 30, podemos crear una sola cadena que contenga todos esos números. No es necesario cambiar nuestro programa en absoluto, más que para pasarle un archivo diferente. También imprimimos los 50 primeros decimales para no tener que ver un millón de números en terminal:

pi_string.py

```
from pathlib import Path

path = Path('pi_million_digits.txt')
contents = path.read_text()

lines = contents.splitlines()
pi_string = ''
for line in lines:
    pi_string += line.lstrip()

print(f"{pi_string[:52]}...")
print(len(pi_string))
```

La salida muestra que realmente tenemos una cadena que contiene pi hasta un millón de decimales:

```
3.14159265358979323846264338327950288419716939937510...
```

```
1000002
```

Python no tiene límites intrínsecos que afecten a la cantidad de datos con la que se puede trabajar; puede trabajar con todos los que la memoria de su sistema pueda manejar.

Nota: Para ejecutar este programa (así como varios de los siguientes ejemplos), necesitará descargar los recursos disponibles del libro.

¿Está su cumpleaños contenido en pi?

Siempre he tenido curiosidad por saber si mi cumpleaños aparece en algún lugar de los números de pi. Vamos a usar el programa que acabamos de escribir para averiguar si la fecha de nacimiento de alguien aparece en alguna parte del primer millón de dígitos de pi. Podemos hacerlo expresando cada cumpleaños como una cadena de números para ver si esa cadena está en `pi_string`:

```
--fragmento omitido--  
for line in lines:  
    pi_string += line.strip()  
  
birthday = input("Enter your birthday, in the form 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.")
```

En primer lugar, le preguntamos al usuario cuándo es su cumpleaños y después comprobamos si esa cadena está en `pi_string`. Vamos a probar:

```
Enter your birthdate, in the form mmddyy: 120372
Your birthday appears in the first million digits of pi!
```

¡Mi cumpleaños aparece entre los dígitos de pi! Una vez hemos leído de un archivo, podemos analizar su contenido de cualquier forma que imaginemos.

PRUÉBELO

- **10-1. Aprender Python:** Abra un archivo nuevo en su editor de texto y escriba unas líneas resumiendo lo que ha aprendido sobre Python hasta ahora. Empiece cada línea con la frase "En Python se puede...". Guarde el archivo como `aprender_python.txt` en el mismo directorio que los ejercicios de este capítulo. Escriba un programa que lea el archivo e imprima dos veces lo que ha escrito: una vez leyendo el archivo completo y otra pasando en bucle por el objeto del archivo.
- **10-2. Aprender C:** Puede usar el método `replace()` para sustituir cualquier palabra de una cadena por otra diferente. Aquí tiene un ejemplo rápido para cambiar '`perro`' por '`gato`' en una oración:

```
>>> message = "Me encantan los perros."
>>> message.replace('perro', 'gato')
'Me encantan los gatos.'
```

Lea cada línea del archivo que acaba de crear, `aprender_python.txt`, y cambie la palabra Python por el nombre de otro lenguaje, como C. Imprima en la pantalla las líneas modificadas.

- **10-3. Un código más sencillo:** El programa `file_reader.py` en esta sección utiliza una variable temporal, `lines`, para mostrar el funcionamiento de `splitlines()`. Puede saltarse la parte de la variable temporal y pasar un bucle directamente por la lista que devuelve `splitlines()`:

```
for line in contents.splitlines():
```

Elimine la variable temporal de cada uno de los programas en esta sección para hacerlos más concisos.

Escribir a un archivo

Una de las formas más sencillas de guardar datos es escribirlos en un archivo. Cuando escribimos texto en un archivo, la salida estará disponible después de cerrar el terminal que contiene la salida del programa. Puede examinar la salida después de que un programa termine de ejecutarse y también compartir los archivos de salida con otros. También puede escribir programas que lean el texto en memoria para trabajar con él otra vez.

Escribir una línea

Una vez definida la ruta, puede escribirla a un archivo usando el método `write_text()`. Para ver cómo funciona, vamos a escribir un sencillo mensaje que guardaremos en un archivo en lugar de imprimirlo en la pantalla:

`write_message.py`

```
from pathlib import Path

path = Path('programming.txt')
path.write_text("I love programming.")
```

El método `write_text()` toma un único argumento: la cadena que queremos escribir en el archivo. Este programa no tiene salida en el terminal, pero, si abrimos el archivo `programming.txt`, veremos una línea:

`programming.txt`

I love programming.

Este archivo se comporta como cualquier otro archivo de nuestro ordenador. Podemos abrirlo, escribir texto nuevo, copiar de él, pegar en él, etc.

Nota: Python solo puede escribir cadenas en un archivo de texto. Si quiere almacenar datos numéricos en un archivo de texto, tendrá que convertir antes los datos a formato de cadena con la función `str()`.

Escribir múltiples líneas

El método `write_text()` hace varias cosas entre bastidores. Si el archivo al que `path` apunta no existe, crea dicho archivo. Además, tras escribir la cadena en el archivo, se asegura de que el archivo se cierra correctamente. Los archivos que no se cierran correctamente pueden conducir a la omisión o corrupción de datos.

Para escribir más de una línea a un archivo, es preciso crear una cadena que contenga todos los contenidos del archivo y, a continuación, hacer una llamada a `write_text()` con dicha cadena. Escribamos ahora varias líneas al archivo `programming.txt`:

```
from pathlib import Path

contents = "I love programming.\n"
contents += "I love creating new games.\n"
contents += "I also love working with data.\n"

path = Path('programming.txt')
path.write_text(contents)
```

Definimos una variable llamada `contents` que aloje la totalidad de los contenidos del archivo. En la siguiente línea, utilizamos el operador `+=` para añadir a esta cadena. Puede hacer esto tantas veces como sea necesario, para crear cadenas de cualquier longitud. En este caso hemos incluido caracteres de línea nueva al final de cada línea para asegurarnos de que cada declaración aparezca en su

propia línea. Si ejecuta este código y a continuación abre `programming.txt`, verá cada una de las líneas siguiente en el archivo de texto:

```
I love programming.  
I love creating new games.  
I also love working with data.
```

También podemos usar espacios, tabulaciones y líneas en blanco para dar formato a la salida, igual que hemos hecho con la salida del terminal. No existen limitaciones a la longitud de las cadenas. Así es como se crean muchos documentos generados por ordenador.

Nota: Tenga cuidado al realizar la llamada `write_text()` sobre un objeto de ruta. Si el archivo ya existe, `write_text()` eliminará los contenidos del archivo y los reescribirá. Más adelante, en este mismo capítulo, aprenderá a comprobar si el archivo existe utilizando `pathlib`.

PRUÉBELO

- **10-4. Invitado:** Escriba un programa que pida al usuario su nombre. Cuando responda, escriba su nombre en un archivo llamado `invitado.txt`.
- **10-5. Libro de invitados:** Escriba un bucle `while` que pida a los usuarios su nombre. Recopile todos los nombres introducidos por los usuarios y a continuación escríbalos en un archivo llamado `libro_invitados.txt`. Asegúrese de que cada entrada aparece en una nueva línea del archivo.

Excepciones

Python utiliza objetos especiales llamados "excepciones" para administrar los errores que surjan durante la ejecución de un programa. Siempre que se produzca un error que haga que Python no sepa exactamente qué hacer a continuación, crea un objeto de excepción. Si escribimos código que maneje la excepción, el

programa seguirá ejecutándose. Si no manejamos la excepción, el programa se detendrá y mostrará un rastreo que incluye un informe de la excepción en cuestión.

Las excepciones se manejan con bloques `try-except`. Un bloque `try-except` pide a Python que haga algo, pero también le dice qué hacer si surge una excepción. Cuando usamos bloques `try-except`, los programas seguirán ejecutándose incluso si algo se tuerce. En lugar de rastreos, que pueden resultar confusos, los usuarios verán los mensajes de error normales que hayamos escrito.

Manejar la excepción `ZeroDivisionError`

Veamos un error muy sencillo que hace que Python lance una excepción. Como sabrá, es imposible dividir un número entre cero, pero vamos a pedirle a Python que lo haga de todos modos:

`division_calculator.py`

```
print(5/0)
```

Python no puede realizar la operación, así que obtenemos un rastreo:

```
Traceback (most recent call last):
  File "division_calculator.py", line 1, in <module>
    print(5/0)
    ~^~
❶ ZeroDivisionError: division by zero
```

El error del que nos informa el rastreo, `ZeroDivisionError`, es un objeto de excepción ❶. Python crea este tipo de objeto en respuesta a una situación en la que no puede hacer lo que le pedimos. Cuando esto sucede, Python detiene el programa y nos informa sobre el tipo de excepción que se ha lanzado. Podemos usar esta información para

modificar nuestro programa. Le diremos a Python lo que tiene que hacer cuando se produzca este tipo de excepción; así, si sucede otra vez, estaremos preparados.

Usar bloques try-except

Cuando crea que puede producirse un error, puede escribir un bloque `try-except` para manejar la excepción que podría lanzarse. Le indicamos a Python que intente ejecutar un código y qué hacer si ese código da como resultado un tipo concreto de excepción.

Aquí tenemos un ejemplo de bloque `try-except` para manejar la excepción `ZeroDivisionError`:

```
try:  
    print(5/0)  
except ZeroDivisionError:  
    print("You can't divide by zero!")
```

Ponemos `print(5/0)`, la línea que ha causado el error, dentro de un bloque `try`. Si el código del bloque `try` funciona, Python omite el bloque `except`. Si el código del bloque `try` provoca un error, Python busca un bloque `except` cuyo error coincide con el que se ha producido y ejecuta el código de ese bloque.

En este ejemplo, el código del bloque `try` produce un `ZeroDivisionError`, así que Python busca un bloque `except` que le diga cómo responder. Luego ejecuta el código en ese bloque y el usuario ve un mensaje de error inteligible en vez de un rastreo:

```
You can't divide by zero!
```

Si hubiese más código después del bloque `try-except`, el programa seguiría ejecutándose porque hemos dicho a Python cómo manejar el error. Veamos un ejemplo en el que capturar un error puede permitir a un programa seguir ejecutándose.

Usar excepciones para evitar fallos

Manejar correctamente los errores es especialmente importante cuando el programa tiene que hacer más cosas después de que se produzca el error. Esto ocurre a menudo en programas que requieren entrada de usuario. Si el programa responde adecuadamente a una entrada no válida, puede solicitar más entrada válida en lugar de fallar.

Vamos a crear una simple calculadora que haga solo divisiones:

division_calculator.py

```
print("Give me two numbers, and I'll divide them.")  
print("Enter 'q' to quit.")  
while True:  
    ❶    first_number = input("\nFirst number: ")  
    if first_number == 'q':  
        break  
    ❷    second_number = input("Second number: ")  
    if second_number == 'q':  
        break  
    ❸    answer = int(first_number) / int(second_number)  
    print(answer)
```

Este programa pide al usuario que introduzca un número, `first_number` ❶ y, si el usuario no escribe `q` para salir, otro número, `second_number` ❷. Luego divide estos dos números para obtener una respuesta ❸. Este programa no hace nada para manejar errores, así que, si se le pide que divida entre cero, fallará:

```
Give me two numbers, and I'll divide them.  
Enter 'q' to quit.
```

```
First number: 5
```



```
    print("You can't divide by 0!")
❸ else:
    print(answer)
```

Pedimos a Python que intente completar la división en un bloque `try` ❶, que incluye solo el código que podría dar lugar a un error. Añadimos cualquier código que dependa del éxito del bloque `try` en el bloque `else`. En este caso, si la división se hace bien, usamos el bloque `else` para imprimir el resultado ❷.

El bloque `except` dice a Python cómo responder cuando se produce un `ZeroDivisionError` ❸. Si el bloque `try` no puede hacer la operación por un error de división entre cero, imprimimos un mensaje que dice al usuario cómo evitar este tipo de error. El programa sigue ejecutándose y el usuario nunca llega a ver un rastreo:

```
Give me two numbers, and I'll divide them.
```

```
Enter 'q' to quit.
```

```
First number: 5
```

```
Second number: 0
```

```
You can't divide by 0!
```

```
First number: 5
```

```
Second number: 2
```

```
2.5
```

```
First number: q
```

El único código que debería ir en un bloque `try` es el que podría lanzar una excepción. A veces, tenemos código adicional que debería ejecutarse solo si el bloque `try` ha tenido éxito; este código va en el bloque `else`. El bloque `except` dice a Python qué hacer si se lanza una excepción determinada cuando ejecute el código del bloque `try`.

Al anticipar posibles fuentes de errores, podemos escribir programas robustos que sigan ejecutándose, aunque encuentren

datos no válidos o falten recursos. Nuestro código será resistente a errores inocentes y ataques maliciosos.

Manejar la excepción `FileNotFoundException`

Un problema habitual cuando se trabaja con archivos es el manejo de archivos que faltan. El archivo que buscamos podría estar en otra ubicación, puede que se haya escrito mal el nombre o puede incluso que no exista el archivo. Podemos manejar todas estas situaciones fácilmente con un bloque `try-except`.

Vamos a intentar leer un archivo que no existe. El siguiente programa intenta leer el contenido de la versión original de *Alicia en el País de las Maravillas (Alice in Wonderland)*, pero no he guardado el archivo `alice.txt` en el mismo directorio que `alice.py`:

`alice.py`

```
from pathlib import Path

path = Path('alice.txt')
contents = path.read_text(encoding='utf-8')
```

Observe que estamos usando `read_text()` de un modo ligeramente distinto a como lo hemos usado anteriormente. El argumento `encoding` es necesario cuando la codificación predeterminada de nuestro sistema no coincide con la codificación del archivo que vamos a leer. Esto ocurrirá con más probabilidad al leer desde un archivo que no se haya creado en nuestro sistema.

Python no puede leer un archivo que no está, así que lanza una excepción:

```
Traceback (most recent call last):
❶  File "alice.py", line 4, in <module>
❷  contents = path.read_text(encoding='utf-8')
^~~~~~
```

```
File ".../pathlib.py", line 1056, in read_text
    with self.open(mode='r', encoding=encoding, errors=errors) as f:
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

File ".../pathlib.py", line 1042, in open
    return io.open(self, mode, buffering, encoding, errors, newline)
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

-
- ③ `FileNotFoundException: [Errno 2] No such file or directory: 'alice.txt'`

Estamos ante un rastreo más largo que los que hemos visto anteriormente. Veamos cómo entender estos rastreos de mayor complejidad. Con frecuencia, es recomendable comenzar por el final del rastreo. En la última línea, vemos que se ha lanzado una excepción `FileNotFoundException` ③. Esto es importante, porque nos indica qué clase de excepción debemos utilizar en el bloque `except` que vamos a escribir.

Si retrocedemos hacia al inicio del rastreo ①, vemos que el error se ha producido en la línea 4 del archivo `alice.py`. La siguiente línea muestra la línea de código que ha provocado el error ②. El resto del rastreo muestra un fragmento de código perteneciente a las bibliotecas implicadas en la apertura y lectura de archivos. Por lo general no es necesario leer o comprender todas estas líneas en un rastreo.

Para gestionar el error que se ha lanzado, el bloque `try` comenzará con la línea identificada como problemática en el rastreo. En nuestro ejemplo, se trata de la línea que contiene `read_text()`:

```
from pathlib import Path

path = Path('alice.txt')
try:
    contents = path.read_text(encoding='utf-8')
① except FileNotFoundError:
    print(f"Sorry, the file {path} does not exist.")
```

En este ejemplo, el código del bloque `try` produce un `FileNotFoundException`, de modo que escribiremos un bloque `except` que coincida con dicho error ❶. Python ejecutará el código de ese bloque cuando no se encuentre el archivo, y el resultado es un mensaje de error inteligible para el usuario en vez de un rastreo:

```
Sorry, the file alice.txt does not exist.
```

El programa no tiene nada que hacer si no existe el archivo, así que esa será toda la salida que veamos. Vamos a ampliar este ejemplo para ver cómo manejar excepciones; puede ser de ayuda cuando se trabaja con más de un archivo.

Analizar texto

Podemos analizar archivos de texto que contengan libros enteros. Muchas obras clásicas de la literatura están disponibles como archivos de texto simple porque pertenecen al dominio público. Los textos que usamos en este apartado proceden de Project Gutenberg (<http://gutenberg.org/>), una colección de obras literarias de dominio público. Es una fuente estupenda si le interesa trabajar con textos literarios en sus proyectos de programación.

Vamos a usar el texto de *Alice in Wonderland* para contar el número de palabras que tiene. Para ello, emplearemos el método de cadena `split()`, que por defecto trocea una cadena siempre que encuentra un espacio:

```
from pathlib import Path

path = Path('alice.txt')

try:
    contents = path.read_text(encoding='utf-8')
except FileNotFoundError:
```

```
print(f"Sorry, the file {path} does not exist.")

else:

    # Cuenta el número aproximado de palabras en el archivo:
❶    words = contents.split()
❷    num_words = len(words)

    print(f"The file {path} has about {num_words} words.")
```

He movido el archivo `alice.txt` al directorio correcto, de manera que el bloque `try` funcionará en esta ocasión. Tomamos la cadena `contents`, que ahora contiene todo el texto de *Alice in Wonderland* como una cadena larga, y utilizamos `split()` para crear un listado de todas las palabras del libro ❶. Utilizando `len()` sobre esta lista ❷, tendremos una aproximación válida al número de palabras del texto original. Por último, imprimimos una declaración que informe del número de palabras encontradas en el archivo. Este código se ubica en el bloque `else` porque únicamente funciona si el código en el bloque `try` se ejecuta con éxito.

La salida nos indica la cantidad de palabras que contiene `alice.txt`:

The file alice.txt has about 29594 words.

El recuento es algo elevado, porque la editorial da información adicional en el archivo de texto, pero es una buena aproximación a la longitud de *Alice in Wonderland*.

Trabajar con múltiples archivos

Vamos a añadir más libros al análisis, pero antes moveremos el grueso del programa a una función llamada `count_words()`. De ese modo, será más fácil analizar varios libros:

`word_count.py`

```
from pathlib import Path

def count_words(path):
    """Contar el número aproximado de palabras en un archivo."""
    try:
        contents = path.read_text(encoding='utf-8')
    except FileNotFoundError:
        print(f"Sorry, the file {path} does not exist.")
    else:
        # Cuenta el número aproximado de palabras en el archivo:
        words = contents.split() num_words = len(words)
        print(f"The file {path} has about {num_words} words.")

path = Path('alice.txt')
count_words(path)
```

La mayor parte de este código no ha cambiado. Únicamente lo hemos sangrado y movido al cuerpo de `count_words()`. Conviene mantener los comentarios actualizados cuando se modifica un programa, así que hemos cambiado el comentario por una cadena de documentación con una redacción ligeramente distinta ❶.

Ahora podemos escribir un bucle corto para contar las palabras de cualquier texto que queramos analizar. Hacemos esto guardando los nombres de los archivos en cuestión en una lista y llamando a `count_words()` para cada elemento de la lista. Intentaremos contar las palabras de las versiones originales de *Alice in Wonderland*, *Siddhartha*, *Moby Dick* y *Little Women*, todas ellas pertenecientes al dominio público. He dejado intencionadamente `siddhartha.txt` fuera del directorio que contiene `word_count.py` para comprobar cómo maneja nuestro programa un archivo desaparecido:

```
from pathlib import Path

def count_words(filename):
```

```
--fragmento omitido--
```

```
filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
'little_women.txt']
for filename in filenames:
❶    path = Path(filename)
    count_words(path)
```

Los nombres de los archivos se almacenan como cadenas simples. Cada una de las cadenas se convierte a un objeto `Path` ❶ antes de la llamada a `count_words()`. La desaparición del archivo `siddhartha.txt` no afecta al resto de la ejecución del programa:

```
The file alice.txt has about 29594 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215864 words.
The file little_women.txt has about 189142 words.
```

El uso del bloque `try-except` en este ejemplo tiene dos ventajas fundamentales: evitamos que nuestros usuarios vean un rastreo y dejamos que el programa siga analizando los textos que pueda encontrar. Si no detectásemos el `FileNotFoundException` de `siddhartha.txt`, el usuario vería un rastreo y el programa dejaría de funcionar después de intentar analizar *Siddhartha*. Nunca analizaría *Moby Dick* ni *Little Women*.

Fallos silenciosos

En el ejemplo anterior, hemos informado a los usuarios de que uno de los archivos no estaba disponible, pero no es necesario informar de todas las excepciones que capturemos. A veces nos interesa que el programa falle en silencio cuando se produzca una excepción y siga después como si no hubiese pasado nada. Para hacer esto, escribiremos un bloque `try` como siempre, pero diremos

a Python explícitamente que no haga nada en el bloque `except`. Python tiene una sentencia `pass` que le dice que no haga nada en un bloque:

```
def count_words(path):  
    """ Cuenta el número aproximado de palabras de un archivo.""""  
    try:  
        --fragmento omitido--  
    except FileNotFoundError:  
        pass  
    else:  
        --fragmento omitido--
```

La única diferencia entre este listado y el anterior es la sentencia `pass` en el bloque `except`. Ahora, cuando se lanza una excepción `FileNotFoundError`, el código del bloque `except` se ejecuta, pero no sucede nada. No se genera un rastreo ni hay salida en respuesta al error. Los usuarios ven el recuento de palabras de los archivos que existen, pero no aparece ningún indicio de que no se haya encontrado un archivo:

```
The file alice.txt has about 29594 words.  
The file moby_dick.txt has about 215864 words.  
The file little_women.txt has about 189142 words.
```

La sentencia `pass` también actúa como marcador de posición. Es un recordatorio de que estamos decidiendo no hacer nada en un punto específico de la ejecución de nuestro programa, aunque es posible que queramos hacer algo ahí más adelante. Por ejemplo, en este caso puede que queramos escribir los nombres de los archivos que faltan en un archivo llamado `missing_files.txt`. Los usuarios no verían este archivo, pero nosotros podríamos leerlo para ocuparnos de los textos desaparecidos.

Decidir qué errores informar

¿Cómo sabemos cuándo informar a los usuarios de un error y cuándo permitir que el programa descarte el error en silencio? Si los usuarios saben qué textos se supone que se van a analizar, apreciarán recibir un mensaje que les informe de por qué algunos no se han analizado. En cambio, si esperan ver unos resultados sin saber qué libros se van a analizar, lo más probable es que no necesiten saber que algunos textos no están disponibles. Proporcionar a los usuarios información que no están buscando puede disminuir la usabilidad de nuestros programas. Las estructuras de manejo de errores de Python nos dan un control minucioso sobre cuánto compartimos con los usuarios cuando algo va mal; depende de nosotros decidir cuánta información les damos.

Un código correctamente escrito y probado no es muy propenso a errores internos, tales como errores sintácticos o lógicos. Sin embargo, cuando el programa depende de factores externos, como la entrada de usuario, la existencia de un archivo o la capacidad de conectarse a una red, cabe la posibilidad de que se produzca una excepción. Con un poco de experiencia sabrá dónde incluir bloques de manejo de excepciones en sus programas y cuánto informar a los usuarios sobre los errores que puedan producirse.

PRUÉBELO

- **10-6. Suma:** Un problema habitual cuando se pide entrada numérica se da cuando la gente proporciona texto en vez de números. Cuando intentamos convertir la entrada en un entero, obtenemos un `ValueError`. Escriba un programa que solicite dos números, súmelos e imprima el resultado. Capture el `ValueError` si cualquiera de los valores de entrada no es un número e imprima un error inteligible para el usuario. Pruebe el programa introduciendo dos números y luego escribiendo texto en vez de un número.
- **10-7. Calculadora de suma:** Incluya el código del ejercicio 10-6 en un bucle `while` para que el usuario pueda seguir introduciendo números, aunque se equivoquen y metan texto en vez de un número.
- **10-8. Perros y gatos:** Cree dos archivos, `gatos.txt` y `perros.txt`. Guarde al menos tres nombres de gato en el primero y tres nombres de perro en el

segundo. Escriba un programa que intente leer estos archivos para imprimir su contenido en la pantalla. Ponga el código en un bloque `try-except` para capturar el error `FileNotFoundException` e imprima un mensaje si falta un archivo. Mueva uno de los archivos a una ubicación distinta en su sistema y asegúrese de que el código del bloque `except` se ejecuta correctamente.

- **10-9. Perros y gatos silenciosos:** Modifique el bloque `except` del ejercicio 10-8 para que falle en silencio si falta un archivo.
- **10-10. Palabras comunes:** Visite Project Gutenberg (<https://gutenberg.org/>) y elija unos cuantos textos en inglés que le gustaría analizar. Descargue los archivos de esas obras o copie el texto del navegador en un archivo de su ordenador.

Puede usar el método `count()` para descubrir cuántas veces aparece una palabra o frase en una cadena. Por ejemplo, el siguiente código cuenta el número veces que aparece '`row`' en una cadena:

```
>>> line = "Row, row, row your boat"
>>> line.count('row')
2
>>> line.lower().count('row')
3
```

Observe que convertir la cadena a minúsculas con `lower()` recoge todas las apariciones de la palabra que buscamos independientemente del formato que tenga.

Escriba un programa que lea los archivos que ha sacado de Project Gutenberg y determine cuántas veces aparece la palabra '`the`' en cada texto. Será una aproximación, ya que también contará palabras como '`then`' y '`there`'. Pruebe a contar '`the`', con un espacio en la cadena, y observe cómo baja el recuento.

Almacenar datos

Muchos de sus programas pedirán a los usuarios que introduzcan algún tipo de información. Podría permitir que los usuarios guarden

preferencias en un juego o aporten datos para una visualización. Sea cual sea el enfoque del programa, guardará la información suministrada por los usuarios en estructuras de datos como listas y diccionarios. Cuando los usuarios cierren el programa, casi siempre le interesará guardar la información que hayan introducido. Una forma sencilla de hacerlo consiste en guardar los datos usando el módulo `json`.

El módulo `json` permite convertir estructuras de datos de Python simples en cadenas JSON, y posteriormente cargar los datos desde ahí la próxima vez que se ejecute el programa. También puede usar `json` para compartir datos entre distintos programas de Python. Es más, el formato de datos JSON no es específico de Python, así que podemos compartir datos en este formato con gente que trabaja con otros lenguajes de programación. Es un formato útil y portátil y es fácil de aprender.

Nota: El formato JSON (*JavaScript Object Notation*, notación de objeto de JavaScript) se desarrolló originalmente para JavaScript. Sin embargo, con el tiempo se ha convertido en un formato utilizado por muchos lenguajes, incluido Python.

Utilizar `json.dumps()` y `json.loads()`

Vamos a escribir un programa corto que almacene una serie de números y otro que los lea de vuelta en la memoria. El primer programa usará `json.dumps()` para almacenar el conjunto de números y el segundo, `json.loads()`.

La función `json.dumps()` toma un argumento: un dato que deberá convertirse a formato JSON. La función devuelve una cadena, que después podemos escribir en un archivo de datos:

`number_writer.py`

```
_writer.py from pathlib import Path  
import json
```

```
numbers = [2, 3, 5, 7, 11, 13]
```

```
❶ path = Path('numbers.json')
❷ contents = json.dumps(numbers)
path.write_text(contents)
```

Primero, importamos el módulo `json` y a continuación creamos una lista de números con la que trabajar. Elegimos un nombre de archivo para guardar la lista de números ❶. Es costumbre usar la extensión `.json` para indicar que los datos de ese archivo están guardados en formato JSON. A continuación utilizamos la función `json.dumps()` ❷ para generar una cadena que contenga la representación JSON de los datos con los que estamos trabajando. Una vez tengamos esta cadena, la escribiremos en el archivo empleando el mismo método `write_text()` que hemos visto con anterioridad.

Este programa no produce ninguna salida, pero vamos a abrir el archivo `numbers.json` para ver qué ocurre. Los datos se almacenan en un formato idéntico a Python:

```
[2, 3, 5, 7, 11, 13]
```

Ahora escribiremos un programa que utilice `json.loads()` para volver a leer la lista en memoria:

number_reader.py

```
from pathlib import Path
import json

❶ path = Path('numbers.json')
❷ contents = path.read_text()
❸ numbers = json.loads(contents)
```

```
print(numbers)
```

Nos aseguramos de leer desde el mismo archivo en el que hemos escrito ❶. Dado que el archivo de datos es simplemente un archivo de texto con un formato específico, podemos leerlo con el método `read_text()` ❷. A continuación pasamos los contenidos del archivo a `json.loads()` ❸. Esta función toma una cadena de formato JSON y devuelve un objeto Python (en este caso, una lista) que asignamos a `numbers`. Por último, imprimimos la lista de números recuperada y vemos que se trata de la misma lista creada en `numbers_writer.py`:

```
[2, 3, 5, 7, 11, 13]
```

Esta es una forma sencilla de compartir datos entre dos programas.

Guardar y leer datos generados por usuarios

Guardar datos con `json` es útil cuando se trabaja con datos generados por los usuarios porque, si no se guardan de alguna manera, se perderán cuando se detenga la ejecución del programa. Veamos un ejemplo en el que se pide al usuario su nombre y apellido la primera vez que ejecuta un programa y luego se recuerdan esos datos en la siguiente ejecución. Empezaremos guardando el nombre de usuario:

```
remember_me.py
```

```
from pathlib import Path  
  
import json  
  
❶ username = input("What is your name? ")  
  
❷ path = Path('username.json')  
  
path.write_text(username)
```

```
contents = json.dumps(username)
path.write_text(contents)

❸ print(f"We'll remember you when you come back, {username}!")
```

En primer lugar pedimos un nombre de usuario para almacenar ❶. A continuación, escribimos los datos que acabamos de recopilar en un archivo llamado `username.json` ❷. Después, imprimimos un mensaje informando al usuario de que hemos guardado sus datos ❸:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

Ahora vamos a escribir un programa nuevo que salude a un usuario cuyo nombre esté ya guardado:

`greet_user.py`

```
from pathlib import Path
import json

❶ path = Path('username.json')
contents = path.read_text()
❷ username = json.loads(contents)

print(f"Welcome back, {username}!")
```

Leemos los contenidos del archivo de datos ❶ y a continuación usamos `json.loads()` para asignar los datos recuperados a la variable `username` ❷. Dado que hemos recuperado el nombre de usuario, podemos dar la bienvenida al usuario con un saludo personalizado:

Welcome back, Eric!

Tenemos que combinar estos dos programas en un único archivo. Cuando alguien ejecute `remember_me.py`, queremos recuperar su nombre de usuario de la memoria, si es posible; de lo contrario, pediremos el nombre de usuario y lo almacenaremos en `username.json` hasta la próxima ocasión. Aquí podríamos utilizar un bloque `try_except` para responder correctamente cuando `username.json` no existe, pero en lugar de eso utilizamos un método muy cómodo del módulo `pathlib`:

`remember_me.py`

```
from pathlib import Path

import json


path = Path('username.json')

❶ if path.exists():

    contents = path.read_text()

    username = json.loads(contents)

    print(f"Welcome back, {username}!")

❷ else:

    username = input("What is your name? ")

    contents = json.dumps(username)

    path.write_text(contents)

    print(f"We'll remember you when you come back, {username}!")
```

Existen muchos métodos de gran ayuda que puede utilizar con objetos `Path`. El método `exists()` devuelve `True` si existe un archivo o carpeta, y `False` si no existe. Aquí utilizamos `path.exists()` para saber si ya se ha guardado un nombre de usuario ❶. Si `username.json` existe, cargamos el nombre de usuario e imprimimos un saludo personalizado para el usuario.

Si el archivo `username.json` no existe ❷, pedimos el nombre de usuario y almacenamos el valor introducido por el usuario.

Asimismo, imprimimos el mensaje de que nos acordaremos de ellos cuando vuelvan.

Sea cual sea el bloque que se ejecute, el resultado es un nombre de usuario y un saludo apropiado. Si es la primera vez que se ejecuta el programa, la salida será esta:

```
What is your name? Eric
We'll remember you when you come back, Eric!
```

De lo contrario, tendremos:

```
Welcome back, Eric!
```

Esta es la salida que vemos si el programa se ha ejecutado al menos una vez antes. Aun cuando los datos almacenados en esta sección no sean más que una cadena simple, el programa funcionaría igualmente con cualquier tipo de datos que puedan ser convertidos a una cadena en formato JSON.

Refactorización

A menudo, llegará un punto en el que su código funcione, pero se dé cuenta de que podría mejorarlo dividiéndolo en una serie de funciones con tareas específicas. Este proceso recibe el nombre de "refactorización". La refactorización hace que nuestro código sea más limpio, fácil de entender y fácil de ampliar.

Podemos refactorizar `remember_me.py` moviendo el grueso de su lógica a una o varias funciones. El foco de `remember_me.py` está en saludar al usuario, así que vamos a mover todo el código existente a una función llamada `greet_user()`:

`remember_me.py`

```
from pathlib import Path
import json
```

```
def greet_user():
    """Saludar al usuario por su nombre."""
    path = Path('username.json')
    if path.exists():
        contents = path.read_text()
        username = json.loads(contents)
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Dado que ahora estamos usando una función, reescribimos los comentarios con una cadena de documentación que refleje cómo funciona el programa ❶. Este archivo está un poco más limpio, pero la función `greet_user()` hace más que simplemente saludar al usuario: también recupera un nombre de usuario almacenado si lo hay y solicita uno nuevo si no lo hay.

Vamos a refactorizar `greet_user()` para que no tenga que hacer tantas tareas diferentes. Empezaremos moviendo el código para recuperar un nombre de usuario guardado a una función aparte:

```
from pathlib import Path
import json

def get_stored_username(path):
    """Obtener el nombre de usuario guardado si está disponible."""

    if path.exists():
```

```

contents = path.read_text()
username = json.loads(contents)
return username

else:
②    return None

def greet_user():
    """Saludar al usuario por su nombre."""
    path = Path('username.json')
    username = get_stored_username(path)
    ③    if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        contents = json.dumps(username)
        path.write_text(contents)
        print(f"We'll remember you when you come back, {username}!")

greet_user()

```

La nueva función `get_stored_username()` tiene un propósito claro ❶, como se indica en la cadena de documentación. Esta función recupera un nombre de usuario guardado y lo devuelve si lo encuentra. Si el archivo pasado a `get_stored_username()` no existe, la función devuelve `None` ❷. Es una buena práctica: una función debería devolver el valor que esperamos o `None`. Esto nos permite realizar una sencilla prueba con el valor de retorno de la función. Imprimimos un mensaje de bienvenida para el usuario si el intento de recuperar un nombre ha tenido éxito ❸; de lo contrario, pedimos un nuevo nombre de usuario.

Debemos factorizar un bloque más de código de `greet_user()`. Si el nombre de usuario no existe, deberíamos mover el código que pide un nuevo nombre a una función dedicada a ese fin:

```
from pathlib import Path
import json

def get_stored_username(path):
    """Obtener el nombre de usuario guardado si está disponible."""
    --fragmento omitido--

def get_new_username(path):
    """Solicitar nuevo nombre de usuario."""
    username = input("What is your name? ")
    contents = json.dumps(username)
    path.write_text(contents)
    return username

def greet_user():
    """Saludar al usuario por su nombre."""
    path = Path('username.json')
    ❶ username = get_stored_username(path)
    if username:
        print(f"Welcome back, {username}!")
    else:
        ❷ username = get_new_username(path)
    print(f"We'll remember you when you come back, {username}!")

greet_user()
```

Cada una de las funciones en esta versión final de `remember_me.py` cumple un propósito claro y definido. Llamamos a `greet_user()` y esa función imprime un mensaje apropiado: o da la bienvenida de vuelta a un usuario existente, o saluda a uno nuevo. Lo hace llamando a `get_new_username()` ❶, que es únicamente responsable de recuperar un nombre de usuario almacenado si lo hay. Por último, en caso de que

sea necesario, `greet_user()` llama a `get_new_username()` ❷, que se ocupa de obtener un nuevo nombre de usuario y guardarlo. Esta división del trabajo es crucial a la hora de escribir un código claro que sea fácil de mantener y de ampliar.

PRUÉBELO

- **10-11. Número favorito:** Escriba un programa que pida al usuario su número favorito. Utilice `json.dump()` para guardar ese número en un archivo. Escriba un programa aparte que lea ese valor e imprima el mensaje "¡Sé cuál es tu número favorito! Es el ____".
- **10-12. Número favorito recordado:** Combine los dos programas del ejercicio anterior en un solo archivo. Si el número ya está guardado, diga al usuario su número favorito. Si no, solicíteselo al usuario y guárdelo en un archivo. Ejecute el programa dos veces para ver si funciona.
- **10-13. Diccionario del usuario:** El ejemplo `remember_me.py` almacena únicamente un dato, el nombre de usuario. Amplíe el ejemplo, pidiendo dos nuevos datos al usuario, y almacene toda la información recopilada en un diccionario. Escriba el diccionario en un archivo utilizando `json.dumps()` y vuelva a leerlo usando `json.loads()`. Imprima un resumen mostrando exactamente lo que el programa recuerda del usuario.
- **10-14. Verificar usuario:** El listado definitivo de `remember_me.py` asume que el usuario ya ha introducido su nombre o que el programa se ejecuta por primera vez. Deberíamos modificarlo por si el usuario actual no es la última persona que usó el programa.

Antes de imprimir un mensaje de bienvenida en `greet_user()`, pregunte al usuario si es el nombre correcto. De lo contrario, llame a `get_new_username()` para obtener el nombre de usuario adecuado.

Resumen

En este capítulo, ha aprendido a trabajar con archivos. Ahora sabe leer la totalidad de los contenidos de un archivo y analizar sus contenidos línea a línea cuando sea necesario. Ha aprendido a

escribir tanto texto como desee en un archivo. Hemos hablado de excepciones y de cómo manejar aquellas excepciones que seguramente encontrará en sus programas. Por último, ha aprendido a almacenar estructuras de datos de Python para poder guardar la información proporcionada por los usuarios, evitando que tengan que empezar desde cero cada vez que ejecuten un programa.

En el capítulo 11, descubriremos técnicas eficientes para probar nuestro código. Esto le ayudará a confiar en que el código que desarrolle es correcto y a identificar los errores que vayan apareciendo a medida que amplíe sus programas.

11

PROBAR EL CÓDIGO



Al escribir una función o una clase, puede además escribir pruebas para su código con el fin de comprobar que el código funciona como debería en respuesta a cualquier tipo de entrada que esté diseñado para recibir. Mediante las pruebas, podemos tener la tranquilidad de que nuestro código funcionará correctamente a medida que más personas empiecen a usar nuestros programas. También

podemos probar código nuevo a medida que lo vayamos añadiendo, para asegurarnos de que los cambios no afectan al comportamiento del programa. Todos los programadores cometén errores, razón por la cual todo programador debe probar su código para localizar un problema antes de que lo encuentre el usuario.

En este capítulo veremos cómo probar nuestro código con `pytest`. La biblioteca `pytest` es una colección de herramientas que le ayudará a escribir sus primeras pruebas de forma rápida y eficaz, al tiempo que ofrece soporte a sus pruebas a medida que crezcan en complejidad con sus proyectos. Python no incluye `pytest` por defecto, de modo que aprenderemos a instalar bibliotecas externas. Aprender a instalar bibliotecas externas le abrirá las puertas a una amplia variedad de código correctamente diseñado. Estas bibliotecas ampliarán considerablemente el tipo de proyectos en los que podrá trabajar.

Aprenderemos a crear una serie de pruebas y comprobar que un conjunto de cada uno de los conjuntos de entrada produce la salida esperada. Descubriremos qué es una prueba que pasa y una prueba fallida, y veremos de qué manera las pruebas fallidas pueden

ayudarnos a mejorar nuestro código. Veremos cómo probar funciones y clases y empezará a entender cuántas pruebas debe escribir para un proyecto.

Instalar pytest con pip

Si bien es cierto que Python incluye numerosas funcionalidades en su biblioteca estándar, los desarrolladores que trabajan con Python dependen en gran medida de paquetes desarrollados por terceros. Un paquete desarrollado por terceros es una biblioteca desarrollada fuera de Python. Algunas bibliotecas de terceros que gozan de gran popularidad terminan incorporándose a la biblioteca estándar de Python y, a partir de ese momento, terminan incorporándose a la mayoría de instalaciones Python. Este suele ser el caso con aquellas bibliotecas que no son susceptibles de sufrir muchos cambios una vez se han solucionado los fallos iniciales. Este tipo de bibliotecas puede evolucionar al mismo ritmo que el propio lenguaje.

Sin embargo, muchos paquetes se mantienen fuera de la biblioteca estándar de Python, por lo que pueden tener un desarrollo en el tiempo independiente del propio lenguaje. Estos paquetes tienden a actualizarse con mucha más frecuencia que si estuvieran vinculados al desarrollo de Python. Es el caso de `pytest` y de la mayor parte de las bibliotecas que utilizaremos en la segunda parte de este libro. No deposite una confianza ciega en cualquier paquete de terceros, pero tampoco desconfíe por principio únicamente por el hecho de que buena parte de la funcionalidad importante de Python se implemente a través de dichos paquetes.

Actualizar pip

Python incorpora una herramienta llamada pip, utilizada para instalar paquetes de terceros. Dado que pip nos ayuda a instalar paquetes procedentes de recursos externos, se actualiza con

frecuencia para resolver cualquier potencial problema de seguridad. Comenzaremos por tanto actualizando pip.

Abra una nueva ventana en su terminal y escriba lo siguiente:

```
$ python -m pip install --upgrade pip
❶ Requirement already satisfied: pip in /.../python3.11/site-packages
(22.0.4)
--fragmento omitido--
❷ Successfully installed pip-22.1.2
```

La primera parte de esta instrucción, `python -m pip`, le indica a Python que debe ejecutar el módulo pip. La segunda parte, `install - upgrade`, le indica a pip que actualice un paquete que ya se ha instalado. La última parte, `pip`, especifica qué paquete de terceros debe ser actualizado. La salida muestra que mi versión de pip, la versión, 22.0.4 ❶, se ha actualizado a la última versión, que en el momento de redactar estas líneas es 22.1.2 ❷.

Puede usar esta instrucción para actualizar cualquier paquete de terceros instalado en su sistema:

```
$ python -m pip install --upgrade package_name
```

Nota: Si utiliza Linux, es posible que pip no esté incluido en su versión de Python. Si recibe un mensaje de error al intentar actualizar pip, consulte las instrucciones del apéndice A.

Instalar pytest

Ahora que hemos actualizado pip, procederemos a instalar `pytest`:

```
$ python -m pip install --user pytest
Collecting pytest
--fragmento omitido--
Successfully installed attrs-21.4.0 iniconfig-1.1.1 ...pytest-7.x.x
```

Seguimos empleando la instrucción `pip install`, pero esta vez sin el indicador `--upgrade`. En lugar de eso, utilizaremos el indicador `--user`, que le dice a Python que instale este paquete únicamente para el usuario actual. La salida muestra que se ha instalado con éxito la última versión de `pytest`, así como varios otros paquetes de los que `pytest` depende.

Puede utilizar la siguiente instrucción para instalar muchos paquetes de terceros:

```
$ python -m pip install --user package_name
```

Nota: Si tiene cualquier dificultad al ejecutar esta instrucción, prueba a ejecutarla sin el indicador `--user`.

Probar una función

Para aprender sobre pruebas, necesitamos código que probar. Aquí tenemos una sencilla función que toma un nombre y un apellido y devuelve un nombre completo con un formato adecuado:

`name_function.py`

```
def get_formatted_name(first, last):
    """Genera un nombre completo con formato adecuado."""
    full_name = f"{first} {last}"
    return full_name.title()
```

La función `get_formatted_name()` combina el nombre y el apellido con un espacio en medio para formar un nombre completo; luego le pone mayúsculas y lo devuelve. Para comprobar que `get_formatted_name()` funciona, vamos a escribir un programa que use esta función. El programa `names.py` permite que los usuarios introduzcan un nombre y un apellido y vean un nombre completo con un formato adecuado:

names.py

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")

while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
        break
    last = input("Please give me a last name: ")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tNeatly formatted name: {formatted_name}.")
```

Este programa importa `get_formatted_name()` de `name_function.py`. El usuario puede introducir una serie de nombres y apellidos y ver los nombres completos con formato que se generan:

Enter 'q' at any time to quit.

```
Please give me a first name: janis
Please give me a last name: joplin
Neatly formatted name: Janis Joplin.
```

```
Please give me a first name: bob
Please give me a last name: dylan
Neatly formatted name: Bob Dylan.
```

```
Please give me a first name: q
```

Vemos que los nombres generados aquí son correctos, pero supongamos que queremos modificar `get_formatted_name()` para que gestione también nombres compuestos. Al hacerlo, tenemos que asegurarnos de no estropear la forma en que la función gestiona los

nombres que constan solo de un nombre y el apellido. Podríamos probar nuestro código ejecutando `names.py` e introduciendo un nombre como `Janis Joplin` cada vez que modifiquemos `get_formatted_name()`, pero sería bastante tedioso. Por suerte, Python ofrece una manera eficiente de automatizar las pruebas de la salida de una función. Si automatizamos las pruebas de `get_formatted_name()`, siempre podremos confiar en que la función funcionará cuando reciba los tipos de nombres para los que hemos escrito pruebas.

Pruebas unitarias y casos de prueba

Existen multitud de enfoques válidos a la hora de probar software. Una de las formas más sencillas de realizar pruebas es pasar una prueba unitaria. Una "prueba unitaria" comprueba que un aspecto específico del comportamiento de una función es correcto. Un "caso de prueba" es un conjunto de pruebas unitarias que, en conjunto, comprueban si la función se comporta como debería, dentro del rango completo de situaciones que esperamos que gestione.

Un buen caso de prueba considera todos los tipos posibles de entrada que podría recibir una función e incluye pruebas para representar cada una de estas situaciones. Un caso de prueba con "cobertura completa" incluye una gama completa de pruebas unitarias que cubren todas las formas posibles en las que se puede usar una función. Conseguir cobertura completa en un proyecto de envergadura puede ser desalentador. A menudo basta con escribir pruebas para los comportamientos clave del código y aspirar a lograr una cobertura completa solo si el proyecto empieza a usarse de forma generalizada.

Una prueba que pasa

Con `pytest`, escribir una prueba unitaria es muy sencillo. Escribiremos una única función de prueba. La función de prueba

llamará a la función que estemos probando, y a continuación lanzaremos una declaración de afirmación sobre el valor devuelto. Si nuestra declaración de afirmación es correcta, la prueba pasará; si es incorrecta, la prueba fallará.

Veamos a continuación la primera prueba de la función

`get_formatted_name()`:

test_name_function.py

```
from name_function import get_formatted_name

❶ def test_first_last_name():
    """¿Funcionan nombres como Janis Joplin?"""
❷     formatted_name = get_formatted_name('janis', 'joplin')
❸     assert formatted_name == 'Janis Joplin'
```

Antes de ejecutar la prueba, veamos con detenimiento esta función. El nombre del archivo de prueba es importante: debe empezar por `test_`. Al pedirle a `pytest` que ejecute las pruebas que hemos escrito, buscará cualquier archivo que empiece por `test_` y ejecutará todas las pruebas que encuentre en dicho archivo.

En el archivo de prueba, importamos en primer lugar la función que queremos probar: `get_formatted_name()`. A continuación definimos una función de prueba: en este caso, `test_first_last_name()` ❶. Se trata de un nombre de función más largo del que hemos venido usando, pero hay una buena razón para ello. En primer lugar, las funciones de prueba deben empezar por `test` seguido de guion bajo. Cualquier función que comience por `test_` será descubierta por `pytest`, y se ejecutará como parte del proceso de prueba.

Además, los nombres de las pruebas deben ser más largos y descriptivos que el nombre de una función típica. Nunca llamamos a la función por nosotros mismos; `pytest` la encontrará y la ejecutará por nosotros. Los nombres de las funciones de prueba deberían ser lo suficientemente largos como para que, si vemos el nombre de la

función en un informe de prueba, sepamos con exactitud cuál es el comportamiento que se está probando. A continuación, llamamos a la función que estamos probando ❷. En este caso, llamamos a `get_formatted_name()` con los argumentos `'janis'` y `'joplin'`, exactamente igual que cuando ejecutamos `names.py`. Asignamos el valor de retorno de esta función a `formatted_name`.

Por último, lanzamos una declaración de afirmación ❸. Una declaración de afirmación se define como una afirmación relativa a una condición. En esta ocasión, estamos afirmando que el valor de `formatted_name` debe ser `'Janis Joplin'`.

Ejecutar una prueba

Si ejecuta directamente el archivo `test_name_funcion.py`, no obtendrá ninguna salida, puesto que no hemos llamado a la función de prueba. En lugar de esto, le pediremos a `pytest` que ejecute por nosotros el archivo de prueba.

Para ello, abra la ventana del terminal y diríjase a la carpeta donde esté ubicado el archivo de prueba. Si utiliza VS Code, puede abrir la carpeta que contiene dicho archivo y utilizar el terminal incrustado en la ventana del editor. En dicha ventana, escriba la instrucción `pytest`. Debería ver lo siguiente:

```
$ pytest
=====
platform darwin -- Python 3.x.x, pytest-7.x.x, pluggy-1.x.x
rootdir: ../../python_work/chapter_11
collected 1 item
test_name_function.py . [100%]
=====
1 passed in 0.00s =====
```

Veamos qué está ocurriendo en esta salida. En primer lugar, vemos información acerca del sistema sobre el que se ejecuta la

prueba ❶. Estoy ejecutando la prueba en un sistema macOS, por lo que puede que usted obtenga una salida diferente. Más importante aún es el hecho de que podemos comprobar qué versiones de Python, `pytest` y otros paquetes se están utilizando para ejecutar la prueba.

A continuación, vemos el directorio desde el cual se ejecuta la prueba ❷; en mi caso, `python_work/chapter_11`. Vemos cómo `pytest` ha encontrado al menos una prueba que pasar ❸, y podemos ver el archivo de prueba que se está ejecutando ❹. El punto que sigue al nombre del archivo nos indica que se ha pasado una única prueba, y el valor de 100 % nos deja claro que se han ejecutado todas las pruebas. Un proyecto de gran envergadura puede tener cientos o miles de pruebas; los puntos y el indicador de porcentajes pueden resultar útiles a la hora de controlar el progreso de las pruebas.

La última línea nos indica que se ha pasado una prueba y que su ejecución ha durado menos de 0,01 segundo.

Esta salida nos indica que la función `get_formatted_name()` siempre funcionará con nombres compuestos por un nombre de pila y un apellido, a menos que modifiquemos la función. Cuando modifiquemos `get_formatted_name()`, podemos volver a pasar esta prueba. Si la prueba pasa, sabemos que la función seguirá siendo válida para nombres como Janis Joplin.

Nota: Si no está seguro de cómo navegar hasta la ubicación correcta en su terminal, consulte el apartado "Ejecutar programas de Python desde un terminal" en el capítulo 1. Si recibe un mensaje que le indica que no se ha encontrado la instrucción `pytest`, utilice en su lugar la instrucción `python -m pytest`.

Una prueba que falla

¿Qué aspecto tiene una prueba que falla? Vamos a modificar `get_formatted_name()` para que pueda manejar nombres compuestos, pero lo haremos de manera que la función falle con nombres que tengan solo un nombre y un apellido, como Janis Joplin.

Esta es la nueva versión de `get_formatted_name()`, que requiere un argumento para el segundo nombre:

`name_function.py`

```
def get_formatted_name(first, middle, last):
    """Genera un nombre completo con formato adecuado."""
    full_name = f"{first} {middle} {last}"
    return full_name.title()
```

Esta versión debería funcionar para personas con nombres compuestos, pero, al probarla, vemos que hemos estropeado la función para la gente con solo un nombre y un apellido. En esta ocasión ejecutar `pytest` dará el siguiente resultado:

```
$ pytest
=====
test session starts =====
--fragmento omitido--
❶ test_name_function.py F [100%]
❷ ===== FAILURES =====
❸ _____ test_first_last_name _____
def test_first_last_name():

    """¿Funcionan nombres como Janis Joplin?"""
❹     > formatted_name = get_formatted_name('janis', 'joplin')
❺ E TypeError: get_formatted_name() missing 1 required positional
argument: 'last'

test_name_function.py:5: TypeError
=====
short test summary info =====
FAILED test_name_function.py::test_first_last_name - TypeError:
get_formatted_name() missing 1 required positional argument: 'last'
=====
1 failed in 0.04s =====
```

Hay mucha información aquí, ya que cuando una prueba falla, son muchos los aspectos a los que debemos prestar atención. El primer elemento que debemos tener en cuenta en la salida es la `F` ❶, que nos indica que una de las pruebas ha fallado. A continuación, vemos una sección que se centra en los errores (`FAILURES`) ❷, puesto que las pruebas fallidas son por regla general lo más importante en lo que debemos fijarnos al pasar una prueba. Después, vemos que `test_first_last_name()` es la función de prueba que ha fallado ❸. El corchete angular ❹ señala la línea del código que ha provocado el fallo de la prueba. La `E` de la siguiente línea ❺ indica el error que ha provocado el fallo: un `TypeError` debido a la ausencia de un argumento posicional obligatorio, `last`. La información más importante se repite en un breve resumen al final, de modo que, cuando ejecutamos muchas pruebas, podemos saber con rapidez qué pruebas han fallado y por qué.

Responder a una prueba fallida

¿Qué hacemos cuando una prueba falla? Suponiendo que estemos comprobando las condiciones adecuadas, una prueba que pasa significa que la función se comporta correctamente y una que falla implica que hay un error en el nuevo código. Así pues, cuando una prueba falla, no hay que cambiar la prueba. De hacerlo, es posible que las pruebas pasen, pero cualquier código que llame a una función igual que lo hace la prueba dejará de funcionar de repente. En lugar de eso, debemos analizar los cambios que acabamos de realizar en la función y reflexionar sobre la manera en que esos cambios han estropeado el comportamiento que queríamos lograr.

En este caso, `get_formatted_name()` requería solo dos parámetros originalmente: un nombre y un apellido. Ahora requiere dos nombres y un apellido. La adición de un segundo nombre obligatorio arruinó el comportamiento original de `get_formatted_name()`. En esta ocasión, la mejor opción es convertir en opcional el argumento del segundo nombre. De este modo, nuestra prueba para nombres como `Janis`

`Joplin` debería pasar otra vez y también deberíamos poder aceptar nombres compuestos. Vamos a modificar `get_formatted_name()` para que el segundo nombre sea opcional y volvemos a ejecutar el caso de prueba. Si pasa, avanzaremos para asegurarnos de que la función gestiona bien los nombres compuestos.

Para que el segundo nombre sea opcional, moveremos el parámetro `middle` al final de la lista de parámetros en la definición de la función y le daremos un valor predeterminado vacío. También añadiremos una prueba `if` que componga bien el nombre completo, dependiendo de si se ha proporcionado o no un segundo nombre:

`name_function.py`

```
def get_formatted_name(first, last, middle=''):
    """Genera un nombre completo con formato adecuado."""
    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 tendrá un nombre compuesto y un apellido. De lo contrario, el nombre completo constará solo de un nombre y un apellido. Ahora la función debería funcionar para los dos tipos de nombres. Para averiguar si la función todavía funciona con nombres como `Janis Joplin`, vamos a ejecutar de nuevo la prueba:

```
$ pytest
=====
 test session starts =====
--fragmento omitido--
test_name_function.py . [100%]
=====
 1 passed in 0.00s =====
```

Ahora el caso de prueba pasa. Es perfecto: significa que la función vuelve a funcionar para nombres como `Janis Joplin` sin necesidad de probar la función manualmente. Arreglar la función ha sido fácil porque la prueba fallida nos ayudó a identificar el código nuevo que estropeaba el comportamiento original.

Añadir pruebas nuevas

Ahora que sabemos que `get_formatted_name()` funciona con nombres simples, vamos a escribir una segunda prueba para nombres compuestos. Lo haremos añadiendo otra función de prueba al archivo `test_name_function.py`:

`test_name_function.py`

```
from name_function import get_formatted_name

def test_first_last_name():
    --fragmento omitido--

def test_first_last_middle_name():
    """¿Funcionan nombres como 'Wolfgang Amadeus Mozart'?"""
❶    formatted_name = get_formatted_name(
        'wolfgang', 'mozart', 'amadeus')
❷    assert formatted_name == 'Wolfgang Amadeus Mozart'
```

Hemos llamado a esta nueva función `test_first_last_middle_name()`. El nombre de la función debe empezar por `test_` para que se ejecute automáticamente al ejecutar `pytest`. Le hemos puesto un nombre que deja claro qué comportamiento de `get_formatted_name()` estamos probando. Como resultado, si la prueba falla, sabremos de inmediato qué tipos de nombre se ven afectados.

Para probar la función, realizamos una llamada a `get_formatted_name()` con un nombre de pila, un apellido y la segunda

parte del nombre compuesto ❶. Después, lanzamos una declaración de afirmación ❷ indicando que el nombre completo devuelto debe coincidir con el nombre completo (nombre de pila, segundo nombre y apellido) que esperamos recibir. Cuando vuelve a ejecutarse `pytest`, ambas pruebas pasan:

```
$ pytest
=====
test session starts =====
--fragmento omitido--
collected 2 items
[100%]
❶ test_name_function.py ..
=====
2 passed in 0.01s =====
```

Los dos puntos ❶ nos indican que ambas pruebas han pasado, algo que queda claro en la última línea de la salida. ¡Genial! Ahora sabemos que la función seguirá funcionando para nombres como `Janis Joplin` y que también funcionará para nombres como `Wolfgang Amadeus Mozart`.

PRUÉBELO

- **11-1. Ciudad, País:** Escriba una función que acepte dos parámetros: un nombre de ciudad y un nombre de país. La función debería devolver una cadena sencilla con la forma *Ciudad, País*, como `Santiago, Chile`. Guarde la función en un módulo llamado `ciudad_funciones.py` y guarde este archivo en una nueva carpeta para que `pytest` no intente ejecutar las pruebas ya escritas.

Cree un archivo `test_ciudades.py` que pruebe la función que acaba de escribir. Escriba una función llamada `test_ciudad_país()` para verificar que llamar a la función con valores como `'santiago'` y `'chile'` produce la cadena correcta. Ejecute `test_ciudades.py` y asegúrese de que `test_ciudad_país()` pasa.

- **11-2. Población:** Modifique la función para que requiera un tercer parámetro, la población. Ahora debería devolver una cadena con la forma *Ciudad, País - habitantes xxx*, como Santiago, Chile - habitantes 5000000. Ejecute `test_ciudades.py` otra vez. Compruebe que `test_ciudad_país()` falla esta vez.

Modifique la función para que el parámetro de la población sea opcional.

Vuelva a ejecutar la prueba y asegúrese de que `test_ciudad_país()` vuelve a pasar.

Escriba una segunda prueba llamada `test_ciudad_país_habitantes()` que compruebe que se puede llamar a la función con los valores '`santiago', 'chile' y habitantes=5000000'. Ejecute las pruebas una vez más, y asegúrese de que la nueva prueba pasa.`

Probar una clase

En la primera parte de este capítulo, escribimos pruebas para una función. Ahora escribiremos pruebas para una clase. Usará clases en muchos de sus programas, por lo que es muy recomendable ser capaz de comprobar que funcionan bien. Si las pruebas de las clases con las que está trabajando pasan, puede tener la tranquilidad de que las mejoras que haga en la clase no estropearán accidentalmente su comportamiento actual.

Varios métodos assert

Hasta ahora, hemos visto un único tipo de método `assert`: una declaración de que una cadena contiene un valor concreto. Al escribir una prueba, puede plantear cualquier declaración que pueda expresarse de forma condicional. Si la condición que esperamos que se cumpla es `True`, nuestra suposición acerca de cómo esa parte del programa se comportará se confirmará, y podremos estar seguros de que no existe error. Si la condición que afirmamos como `True` es en realidad `False`, la prueba no pasará y sabremos que tenemos un problema que resolver. La tabla 11.1 describe algunos de los

métodos `assert` más útiles, que podrá incorporar en sus pruebas iniciales.

Tabla 11.1. Métodos `assert` utilizados frecuentemente en pruebas.

Método	Uso
<code>assert a == b</code>	Verifica que dos valores son idénticos.
<code>assert a != b</code>	Verifica que dos valores no son idénticos.
<code>assert a</code>	Verifica que <code>a</code> es <code>True</code> .
<code>assert not a</code>	Verifica que <code>a</code> es <code>False</code> .
<code>assert element in list</code>	Verifica que un elemento está en una lista.
<code>assert element not in list</code>	Verifica que un elemento no está en una lista.

Estos no son más que algunos ejemplos; cualquier cosa que pueda expresarse en forma de sentencia condicional puede incluirse en una prueba.

Una clase para probar

Probar una clase es similar a probar una función, dado que buena parte del trabajo implica probar el comportamiento de los métodos de la clase. Sin embargo, existen unas cuantas diferencias, así que vamos a escribir una clase para probarla. Imagine una clase que ayude a administrar encuestas anónimas:

`survey.py`

```
class AnonymousSurvey:  
    """Recoge respuestas anónimas a una pregunta de una encuesta."""  
  
    ①     def __init__(self, question):  
        """Guarda una pregunta y se prepara para guardar respuestas."""  
        self.question = question
```

```

        self.responses = []

❷    def show_question(self):
        """Muestra la pregunta del sondeo."""
        print(self.question)

❸    def store_response(self, new_response):
        """Guarda una sola respuesta a la encuesta."""
        self.responses.append(new_response)

❹    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 empieza con una pregunta que nosotros proporcionamos ❶ e incluye una lista vacía para almacenar las respuestas. La clase tiene métodos para imprimir la pregunta del sondeo ❷, añadir una nueva respuesta a la lista de respuestas ❸ e imprimir todas las respuestas guardadas en la lista ❹. Para crear una instancia de esta clase, lo único que tenemos que proporcionar es una pregunta. Una vez que tenemos una instancia que representa una encuesta particular, mostramos la pregunta de la encuesta con `show_question()`, guardamos una respuesta con `store_response()` y mostramos los resultados con `show_results()`.

Para comprobar que la clase `AnonymousSurvey` funciona, vamos a escribir un programa que la utilice:

language_survey.py

```

from survey import AnonymousSurvey

# Define una pregunta y hace una encuesta.

```

```

question = "What language did you first learn to speak?"
language_survey = AnonymousSurvey(question)

# Muestra la pregunta y guarda las respuestas a la pregunta.
language_survey.show_question()
print("Enter 'q' at any time to quit.\n")
while True:
    response = input("Language: ")
    if response == 'q':
        break
    language_survey.store_response(response)

# Muestra los resultados de la encuesta.
print("\nThank you to everyone who participated in the survey!")
language_survey.show_results()

```

Este programa define una pregunta, en este caso sobre el primer idioma que aprendió el usuario ("What language did you first learn to speak?"), y crea un objeto `AnonymousSurvey` con esa pregunta. El programa llama a `show_question()` para mostrar la pregunta y luego pide respuestas. Cada respuesta se guarda según se recibe. Cuando se han introducido todas las respuestas (el usuario escribe `q` para salir), `show_results()` imprime los resultados de la encuesta:

What language did you first learn to speak?

Enter 'q' at any time to quit.

```

Language: English
Language: Spanish
Language: English
Language: Mandarin
Language: q

```

Thank you to everyone who participated in the survey!

Survey results:

- English
 - Spanish
 - English
 - Mandarin
-

Esta clase funciona para una sencilla encuesta anónima, pero supongamos que queremos mejorar `AnonymousSurvey` y el módulo en el que se encuentra, `survey`. Podríamos permitir que cada usuario introduzca más de una respuesta. Podríamos escribir un método que enumere solamente las respuestas únicas e informe de cuántas veces se ha dado cada una, o incluso podríamos escribir otra clase para administrar encuestas no anónimas.

Implementar esos cambios podría afectar al comportamiento actual de la clase `AnonymousSurvey`. Por ejemplo, es posible que, al intentar permitir que cada usuario introduzca respuestas múltiples, cambiemos por accidente la forma en la que se gestionan las respuestas simples. Para asegurarnos de no estropear el comportamiento existente mientras desarrollamos este módulo, podemos escribir pruebas para la clase.

Probar la clase `AnonymousSurvey`

Vamos a escribir una prueba que verifica un aspecto del comportamiento de `AnonymousSurvey`. Escribiremos una prueba para verificar que se ha almacenado correctamente una respuesta única a la pregunta planteada en la encuesta:

`test_survey.py`

```
from survey import AnonymousSurvey

❶ def test_store_single_response():

    """Comprobar que una respuesta única se almacena
    correctamente."""

    question = "What language did you first learn to speak?"
```

```
❷ language_survey = AnonymousSurvey(question)
language_survey.store_response('English')
❸ assert 'English' in language_survey.responses
```

Empezamos importando la clase que queremos probar, `AnonymousSurvey`. La primera función de prueba verifica que, cuando guardemos una respuesta a la pregunta planteada en la respuesta, dicha respuesta terminará en la lista de respuestas de la encuesta. Un buen nombre descriptivo para esta función sería `test_store_single_response()` ❶. Si esta prueba falla, sabremos a partir del nombre de función del resumen de la prueba que ha habido un problema al guardar una respuesta única a la encuesta.

Para probar el comportamiento de una clase, tenemos que crear una instancia de dicha clase. Creamos una instancia llamada `language_survey` ❷, con la pregunta "What language did you first learn to speak?" (¿Qué idioma aprendió a hablar primero?). Después, verificamos que la respuesta se ha guardado correctamente. Para ello empleamos el método `assert`, declarando que `English` se encuentra en la lista `language_survey.responses` ❸.

Por defecto, ejecutar el comando `pytest` sin argumentos ejecutará todas las pruebas que `pytest` descubra en el directorio actual. Para centrarse en las pruebas de un único archivo, pase por el nombre del archivo de la prueba que desea ejecutar. En esta ocasión ejecutaremos solamente una prueba que escribimos para `AnonymousSurvey`:

```
$ pytest test_survey.py
=====
 test session starts =====
--fragmento omitido--
test_survey.py . [100%]
=====
 1 passed in 0.01s =====
```

Es un buen comienzo, pero una encuesta solo resulta útil si genera más de una respuesta. Comprobemos que las tres respuestas

pueden guardarse correctamente. Para ello, añadiremos otro método a `TestAnonymousSurvey`:

```
from survey import AnonymousSurvey

def test_store_single_response():
    --fragmento omitido--


def test_store_three_responses():
    """Comprueba que se guardan correctamente tres respuestas individuales."""
    question = "What language did you first learn to speak?"
    language_survey = AnonymousSurvey(question)
    ❶ responses = ['English', 'Spanish', 'Mandarin']
    for response in responses:
        language_survey.store_response(response)

    ❷ for response in responses:
        assert response in language_survey.responses
```

Llamamos a la nueva función `test_store_three_responses()`. Creamos un objeto de encuesta igual que en `test_store_single_response()`. Definimos una lista que contenga tres respuestas diferentes ❶ y a continuación llamamos a `store_response()` para cada una de estas respuestas. Cuando se han almacenado las respuestas, escribimos otro bucle y comprobamos que todas las respuestas están en `language_survey.responses` ❷.

Al volver a ejecutar el archivo de prueba, las dos pruebas pasan (la de una respuesta y la de tres):

```
$ pytest test_survey.py
=====
test session starts =====
--fragmento omitido--
test_survey.py .. [100%]
```

```
===== 2 passed in 0.01s =====
```

Funciona a la perfección. No obstante, estas pruebas son un poco repetitivas, así que vamos a usar otra característica de `pytest` para hacerlas más eficientes.

Configuración de pruebas

En `test_survey.py` hemos creado una nueva instancia de `AnonymousSurvey` para cada función de prueba. Esto es adecuado para el ejemplo sencillo con el que estamos trabajando, pero, en un proyecto del mundo real con decenas o cientos de pruebas, sería un problema.

A la hora de pasar pruebas, una función de configuración de pruebas o *fixture* nos ayuda a configurar el entorno en el que vamos a pasar las pruebas. Con frecuencia, esto implica crear un recurso que pueda ser utilizado en más de una prueba. Podemos crear este entorno en `pytest` escribiendo una función con el decorador `@pytest.fixture`. Un decorador es una directiva colocada justo antes de una definición de función; Python aplica esta directiva a la función antes de ejecutarla, con el fin de alterar el comportamiento del código de función. No se preocupe si suena complicado: puede empezar a utilizar decoradores de paquetes de terceros antes de escribir los tuyos propios.

Vamos a utilizar una función de configuración de prueba o *fixture* para crear una única instancia de encuesta que pueda ser utilizada en ambas funciones de prueba en `test_survey.py`:

```
import pytest

from survey import AnonymousSurvey

❶ @pytest.fixture
❷ def language_survey():
    """Una encuesta disponible para todas las funciones de prueba."""
    pass
```

```

question = "What language did you first learn to speak?"
language_survey = AnonymousSurvey(question)
return language_survey

❸ def test_store_single_response(language_survey):
    """Prueba que se ha guardado correctamente una respuesta
    única."""
❹ language_survey.store_response('English')
    assert 'English' in language_survey.responses
❺ def test_store_three_responses(language_survey):
    """Prueba que se han guardado correctamente tres respuestas
    individuales."""
    responses = ['English', 'Spanish', 'Mandarin']

    for response in responses:
❻        language_survey.store_response(response)
    for response in responses:
        assert response in language_survey.responses

```

Ahora debemos importar `pytest`, dado que estamos usando un decorador definido en `pytest`. Aplicamos el decorador `@pytest.fixture` ❶ a la nueva función `language_survey()` ❷. Esta función crea un objeto `AnonymousSurvey` y devuelve la nueva encuesta.

Observe que las definiciones de ambas funciones de prueba han cambiado ❸ ❹; ahora, cada una de las funciones de prueba contiene un parámetro llamado `language_survey`. Cuando un parámetro en una función de prueba coincide con el nombre de una función con el decorador `@pytest.fixture`, la configuración de la prueba se ejecutará automáticamente y devolverá un valor que pasará a la función de prueba. En este ejemplo, la función `language_survey()` suministra tanto `test_store_single_response()` como `test_store_three_responses()` con una instancia `language_survey()`.

No existe código nuevo en ninguna de estas funciones de prueba, pero observe que se han eliminados dos líneas de cada función ❸ ❹: la línea que definía una pregunta y la línea que creó el objeto `AnonymousSurvey`.

Cuando volvemos a ejecutar el archivo de prueba, ambas pruebas pasan. Estas pruebas deberían resultarle de especial utilidad cuando intente expandir `AnonymousSurvey` con el fin de gestionar respuestas múltiples para cada persona. Una vez modificado el código para aceptar respuestas múltiples, puede pasar estas pruebas y asegurarse de que la capacidad de guardar una respuesta única o una serie de respuestas individuales no se ha visto afectada.

La estructura que hemos visto parece complicada: contiene el que probablemente sea el código más abstracto visto hasta ahora. No es necesario que utilice este método inmediatamente; es preferible escribir pruebas con mucho código repetitivo a no escribir ninguna prueba. Simplemente, sea consciente de que, una vez ha escrito pruebas suficientes como para que la repetición sea un obstáculo, existe una forma conocida de lidiar con esta repetición. Además, el empleo de *fixtures* en ejemplos sencillos como el que acabamos de ver no acorta el código ni facilita su seguimiento, pero, en proyectos con muchas pruebas o en situaciones donde sean necesarias muchas líneas de código para crear un recurso utilizado en múltiples pruebas, puede mejorar de forma drástica su código de pruebas.

Cuando quiera utilizar este método, escriba una función que genere el recurso utilizado por múltiples funciones de prueba. Añada el decorador `@pytest.fixture` a la nueva función y agregue el nombre de esta función como parámetro para cada una de las funciones de prueba que utilicen este recursos. Sus pruebas serán más cortas, fáciles de escribir y de mantener de ahí en adelante.

PRUÉBELO

- **11-3. Empleado:** Escriba una clase llamada `Empleado`. El método `__init__()` debería tomar un nombre, un apellido y un salario anual y guardar todos estos atributos. Escriba un método llamado `dar_aumento()` que

añada 5.000 euros al salario anual por defecto, pero que también acepte otros aumentos.

Escriba un caso de prueba para `Empleado` con dos funciones de prueba:

`test_dar_aumento_predeterminado()` y `test_dar_aumento_personalizado()`.

Escriba sus pruebas una vez sin utilizar el método `fixture` y asegúrese de que ambas pruebas pasan. A continuación, haga el mismo ejercicio utilizando dicho método para no tener que crear una nueva instancia de empleado en cada función de prueba. Pase de nuevo las pruebas y asegúrese de que ambas pasan.

Resumen

En este capítulo ha aprendido a escribir pruebas para funciones y clases con herramientas del módulo `pytest`. Ha aprendido a escribir funciones de prueba que verifiquen comportamientos concretos que deben darse en sus funciones y clases. Hemos aprendido a utilizar las funciones de configuración de pruebas o *fixtures* para crear de forma eficaz recursos que podrá utilizar en múltiples funciones de prueba en un archivo de pruebas.

Las pruebas son un tema importante desconocido para muchos principiantes. No es necesario escribir pruebas para todos los proyectos sencillos que cree mientras aprende. Sin embargo, cuando empiece a trabajar en proyectos que impliquen un esfuerzo de desarrollo significativo, debería probar los comportamientos críticos de sus funciones y clases. Tendrá la tranquilidad de que las novedades que vaya introduciendo en su proyecto no estropean las partes anteriores que funcionaban y eso le dará libertad para mejorar el código. Si por accidente estropea una funcionalidad existente, lo descubrirá de inmediato y podrá resolver el problema con más facilidad. Responder a una prueba fallida que ejecutamos nosotros es mucho más fácil que responder a un fallo del que nos ha informado un usuario descontento.

Los demás programadores respetarán más sus proyectos si incluyen algunas pruebas de inicio. Se sentirán más cómodos

experimentando con su código y mostrarán una mayor disposición a colaborar con usted. Si quiere contribuir a un proyecto en el que trabajan otros programadores, se esperará que demuestre que su código pasa las pruebas existentes y, por lo general, se le pedirá que escriba pruebas para los nuevos comportamientos que introduzca en el proyecto.

Experimente con las pruebas para familiarizarse con el proceso de probar el código. Escriba pruebas para los comportamientos más importantes de sus funciones y clases, pero no intente dar una cobertura total a sus primeros proyectos, a menos que tenga una razón concreta para hacerlo.

Parte II

PROYECTOS

¡Enhorabuena! Ya sabe lo suficiente sobre Python para empezar a crear proyectos interactivos y útiles. Crear sus propios proyectos le enseñará nuevas habilidades y le permitirá asentar la asimilación de los conceptos explicados en la primera parte del libro.

Esta segunda parte contiene tres tipos de proyectos. Puede trabajar en cualquiera de ellos o en todos los proyectos, en el orden que prefiera. Aquí tiene una breve descripción de cada proyecto para que decida cuál explorar primero.

Alien Invasion: Hacer un juego con Python

En el proyecto Alien Invasion (capítulos 12, 13 y 14), usaremos el paquete Pygame para desarrollar un juego en 2D. El objetivo del juego es derribar una flota de alienígenas a medida que descienden por la pantalla en niveles que van aumentando en velocidad y dificultad. Al finalizar el proyecto, habrá adquirido habilidades que le permitirán desarrollar sus propios juegos en 2D con Pygame.

Visualización de datos

El proyecto de visualización de datos empieza en el capítulo 15, donde aprenderá a generar datos y crear una serie de visualizaciones funcionales y bonitas de esos datos con Matplotlib y Plotly. En el capítulo 16, veremos cómo acceder a datos de fuentes en línea para alimentar un paquete de visualización que cree gráficos con datos meteorológicos y un mapa de la actividad sísmica mundial. Por último, el capítulo 17 explica cómo escribir un programa que descarga y visualiza datos automáticamente. Aprender a hacer

visualizaciones le permitirá explorar el campo de la minería de datos, una de las habilidades más demandadas hoy en día en el ámbito de la programación.

Aplicaciones web

En el proyecto de aplicaciones web (capítulos 18, 19 y 20), usaremos el paquete Django para crear una sencilla aplicación web que permita a los usuarios llevar un diario acerca de una serie de temas de estudio. Los usuarios crearán una cuenta con un nombre de usuario y una contraseña, introducirán un tema y harán entradas sobre aquello que estén aprendiendo. También veremos cómo desplegar la aplicación para que cualquier persona del mundo pueda acceder a ella.

Tras completar este proyecto, podrá empezar a crear sus propias aplicaciones web sencillas y estará listo para explorar recursos más exhaustivos sobre la creación de aplicaciones con Django.

12

UNA NAVE QUE DISPARA BALAS



¡Vamos a crear un juego llamado Alien Invasion! Usaremos Pygame, una colección de módulos divertidos y potentes de Python que gestionan gráficos, animación e incluso sonido, lo que nos facilita mucho la creación de juegos sofisticados. Con Pygame ocupándose de tareas como dibujar imágenes en la pantalla, podemos concentrarnos en la lógica de alto nivel de la dinámica del juego.

En este capítulo, instalaremos Pygame y crearemos un cohete espacial que se mueva hacia la izquierda y hacia la derecha disparando balas en respuesta a la entrada del jugador. En los dos capítulos siguientes, crearemos una flota alienígena para destruir y seguiremos refinando el juego poniendo límites al número de naves que se puede usar y añadiendo un marcador.

Mientras crea este juego, también aprenderá a manejar proyectos grandes que se distribuyan en varios archivos. Refactorizaremos mucho código y administraremos el contenido de los archivos para organizar el proyecto y hacer que el código sea efectivo.

Crear juegos es una forma ideal de divertirse mientras se aprende un lenguaje de programación. Jugar a un juego creado por uno mismo es muy satisfactorio, y escribir un juego tan sencillo nos ayudará a entender cómo desarrollan sus juegos los profesionales. Mientras trabaja en este capítulo, escriba y ejecute el código para identificar de qué manera cada bloque va aportando algo a la experiencia global del juego. Experimente con distintos valores y configuraciones para entender mejor cómo ir refinando las interacciones en sus juegos.

Nota: Alien Invasion ocupa varios archivos, así que cree una nueva carpeta `alien_invasion` en su sistema. Asegúrese de guardar ahí todos los archivos de proyecto para que las sentencias `import` funcionen correctamente.

Si se siente cómodo con el control de versiones, podría interesarle usarlo para este proyecto. Si no lo ha hecho nunca, consulte el apéndice D para una visión general del tema.

Planificación del proyecto

Cuando creamos un proyecto grande, es importante trazar un plan antes de empezar a escribir código. Este plan le ayudará a mantenerse centrado y aumenta las probabilidades de completar el proyecto. Vamos a escribir una descripción de la mecánica del juego. Aunque la siguiente descripción no cubre todos los detalles de Alien Invasion, da una idea clara de cómo empezar a montar el juego:

En Alien Invasion, el jugador controla una nave que aparece en el centro de la pantalla, en la parte inferior. El jugador puede mover la nave hacia la izquierda y hacia la derecha con las teclas de dirección y disparar balas con la **Barra espaciadora**. Cuando comienza el juego, una flota de extraterrestres llena el cielo y se mueve hacia abajo por la pantalla. El jugador dispara a los aliens y los destruye. Cuando el jugador consiga acabar con todos los alienígenas, aparece una nueva flota que se mueve más rápido que la anterior. Si un alien toca la nave del jugador o llega al fondo de la pantalla, el jugador pierde una vida. El juego termina cuando el jugador pierde tres vidas.

Para la primera fase de desarrollo, haremos una nave que se pueda desplazar hacia la derecha y hacia la izquierda cuando el jugador pulse las flechas de dirección del teclado y que dispare cuando el jugador pulse la **Barra espaciadora**. Tras configurar este comportamiento, podemos crear los aliens y refinar la mecánica del juego.

Instalar Pygame

Antes de empezar a escribir código, instale Pygame. Lo haremos del mismo modo en que instalamos pytest en el capítulo 11: con pip. Si se ha saltado el capítulo 11 o si necesita refrescar sus conocimientos sobre pip, vuelva a consultar este capítulo. Para instalar Pygame, escriba el siguiente comando:

```
$ python -m pip install --user pygame
```

Si usa un comando distinto de `python` para ejecutar programas o iniciar una sesión de terminal, como `python3`, asegúrese de utilizar dicho comando.

Iniciar el proyecto del juego

Empezaremos a construir el juego creando una ventana de Pygame vacía. Más adelante dibujaremos los elementos del juego, como la nave y los extraterrestres, en esta ventana. También haremos que nuestro juego responda a entrada de usuario, configuraremos el color de fondo y cargaremos una imagen de una nave.

Crear una ventana de Pygame y responder a entrada de usuario

Haremos una ventana vacía de Pygame creando una clase que represente el juego. En su editor de texto, cree un nuevo archivo y guárdelo como `alien_invasion.py`; luego escriba lo siguiente:

`alien_invasion.py`

```
import sys

import pygame

class AlienInvasion:
```

```

"""Clase general para gestionar los recursos y el comportamiento
del juego."""

def __init__(self):
    """Inicializa el juego y crea recursos."""
    ❶ pygame.init()

    ❷ self.screen = pygame.display.set_mode((1200, 800))
    pygame.display.set_caption("Alien Invasion")

def run_game(self):
    """Inicia el bucle principal para el juego."""
    ❸ while True:
        # Busca eventos de teclado y ratón.
        ❹ for event in pygame.event.get():
            ❺ if event.type == pygame.QUIT:
                sys.exit()

        # Hace visible la última pantalla dibujada.
    ❻ pygame.display.flip()

if __name__ == '__main__':
    # Hace una instancia del juego y lo ejecuta.
    ai = AlienInvasion()
    ai.run_game()

```

Primero, importamos los módulos `sys` y `pygame`. El módulo `pygame` contiene la funcionalidad que necesitamos para crear un juego. Usaremos las herramientas del módulo `sys` para salir del juego cuando el jugador quiera.

Alien Invasion empieza como una clase llamada `AlienInvasion`. En el método `__init__()`, la función `pygame.init()` inicializa la configuración de fondo que necesita Pygame para funcionar correctamente ❶. A

continuación llamamos a `pygame.display.set_mode()` para crear una ventana ❷ en la que dibujaremos todos los elementos gráficos del juego. El argumento `(1200, 800)` es una tupla que define las dimensiones de la ventana del juego, que tendrá 1.200 píxeles de ancho por 800 de alto. (Puede ajustar estos valores dependiendo del tamaño de su monitor). Asignamos esta ventana al atributo `self.screen` para que esté disponible en todos los métodos de la clase.

El objeto que asignamos a `self.screen` se denomina "superficie". Una superficie en Pygame es una parte de la pantalla donde se puede mostrar un elemento del juego. Cada elemento, como un alien o una nave, ocupa su propia superficie. La superficie que devuelve `display.set_mode()` representa la ventana completa del juego. Cuando activamos el bucle de animación del juego, esta superficie volverá a dibujarse a cada paso por el bucle para actualizarse con cualquier cambio producido por la entrada del usuario.

El método `run_game()` controla el juego. Este método contiene un bucle `while` ❸, que se ejecuta continuamente. El bucle `while` contiene un bucle de eventos y código para administrar las actualizaciones de la pantalla. Un "evento" es una acción que realiza el usuario mientras juega, como pulsar una tecla o mover el ratón. Para que nuestro programa responda a eventos, escribimos un bucle que escuche los eventos y realice las acciones adecuadas dependiendo del tipo de eventos que haya. El bucle `for` ❹ anidado en el bucle `while` es un bucle de eventos.

Para acceder a los eventos detectados por Pygame, utilizaremos la función `pygame.event.get()`. Esta función devuelve una lista de los eventos que se han producido desde la última vez que se la llamó. Cualquier evento de teclado o ratón hará que se ejecute este bucle `for`. Dentro del bucle, escribiremos una serie de sentencias `if` para detectar eventos específicos y responder en consecuencia. Por ejemplo, cuando el jugador haga clic en el botón para cerrar la ventana del juego, se detecta un evento `pygame.QUIT` y llamamos a `sys.exit()` para salir ❺.

La llamada a `pygame.display.flip()` ❻ dice a Pygame que haga visible la última pantalla dibujada. En este caso, simplemente dibuja

una pantalla vacía en cada paso por el bucle `while`, borrando la pantalla antigua para que solo se vea la nueva. Cuando movemos elementos del juego, `pygame.display.flip()` actualiza constantemente la pantalla para mostrar las nuevas posiciones de esos elementos y ocultar las viejas, creando la ilusión de un movimiento suave.

Al final del archivo, creamos una instancia del juego y llamamos a `run_game()`. Colocamos `run_game()` en un bloque `if` que solo se ejecute si se llama al archivo directamente. Cuando ejecute este archivo `alien_invasion.py`, debería ver una ventana de Pygame vacía.

Controlar la tasa de *frames*

Lo ideal es que los juegos se ejecuten a la misma velocidad o con la misma tasa de *frames* o cuadros por segundo (fps) en cualquier sistema. Controlar la tasa de *frames* en un juego que pueda ejecutarse en diferentes sistemas es una cuestión compleja, pero Pygame ofrece una vía relativamente sencilla para conseguirlo. Vamos a crear un reloj y nos aseguraremos de que el segundero suene cada vez que pasa por el bucle principal. Siempre que el bucle procese más rápidamente que la tasa que definamos, Pygame calculará la cantidad de tiempo correcta para hacer una pausa, con el objetivo de que el juego se ejecute siguiendo una tasa consistente.

Definiremos el reloj en el método `__init__()`:

`alien_invasion.py`

```
def __init__(self):
    """Inicializa el juego y crear los recursos de juego."""
    pygame.init()
    self.clock = pygame.time.Clock()
    --fragmento omitido--
```

Una vez inicializado `pygame`, creamos una instancia de la clase `clock`, a partir del módulo `pygame.time`. A continuación haremos que el

reloj marque el tiempo al final del bucle `while` en `run_game()`:

```
def run_game(self):  
    """Inicia el bucle principal del juego."""  
    while True:  
        --fragmento omitido--  
        pygame.display.flip()  
        self.clock.tick(60)
```

El método `tick` toma un argumento: la tasa de *frames* del juego. En este caso utilizamos un valor de 60, de tal modo que Pygame procurará que el bucle se ejecute exactamente 60 veces por segundo.

Nota: El reloj de Pygame debe ayudar a que el juego se ejecute de forma consistente en la mayoría de sistemas. Si el juego no se ejecuta con consistencia en su sistema, prueba con diferentes valores de tasa de cuadros. Si no encuentra una tasa adecuada en su sistema, excluya el reloj y configure los ajustes del juego hasta conseguir una correcta ejecución.

Configurar el color de fondo

Pygame crea una pantalla negra por defecto, pero eso es aburrido. Vamos a configurar un color de fondo distinto. Lo haremos al final del método `__init__()`.

`alien_invasion.py`

```
def __init__(self):  
    --fragmento omitido--  
    pygame.display.set_caption("Alien Invasion")  
  
    # Configura el color de fondo.  
❶    self.bg_color = (230, 230, 230)
```

```

def run_game(self):
    --fragmento omitido--
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Redibuja la pantalla en cada paso por el bucle.
    ❷ self.screen.fill(self.bg_color)

    # Hace visible la última pantalla dibujada.
    pygame.display.flip()
    self.clock.tick(60)

```

Los colores se especifican en Pygame como RGB: una mezcla de rojo, verde y azul. Cada valor de color puede ir de 0 a 255. El valor de color `(255, 0, 0)` es rojo, `(0, 255, 0)` es verde y `(0, 0, 255)` es azul. Podemos mezclar distintos valores RGB para crear hasta 16 millones de colores. El valor de color `(230, 230, 230)` mezcla cantidades iguales de rojo, azul y verde, dando lugar a un color de fondo gris claro. Asignamos este color a `self.bg_color` ❶. Rellenamos la pantalla con el color de fondo usando el método `fill()` ❷, que actúa sobre una superficie y toma solo un argumento: un color.

Crear una clase Settings

Cada vez que introducimos una nueva funcionalidad en el juego, por lo general crearemos también una nueva configuración. En lugar de añadir configuraciones por todo el código, vamos a escribir un módulo `settings` que contenga una clase `Settings` para guardar todos estos valores en una misma ubicación. Este enfoque nos permite trabajar solo con un objeto `settings` cada vez que necesitemos acceder a una configuración individual. También hace que sea más

fácil modificar el aspecto y el comportamiento del juego a medida que el proyecto crezca: para modificar el juego solo tendremos que cambiar los valores pertinentes en `settings.py`, que es lo que vamos a crear ahora, en vez de buscar distintas configuraciones por todo el proyecto.

Cree un nuevo archivo llamado `settings.py` en su carpeta `alien_invasion` y añada esta clase `Settings` inicial:

settings.py

```
class Settings:
    """Una clase para guardar toda la configuración de Alien Invasion."""

    def __init__(self):
        """Inicializa la configuración del juego."""
        # Configuración de la pantalla
        self.screen_width = 1200
        self.screen_height = 800
        self.bg_color = (230, 230, 230)
```

Para crear una instancia de `Settings` en el proyecto y usarla para acceder a la configuración, tendremos que modificar `alien_invasion.py` así:

alien_invasion.py

```
--fragmento omitido--
import pygame

from settings import Settings

class AlienInvasion:

    """Clase general para gestionar los recursos y el comportamiento
    del juego."""
```

```

def __init__(self):
    """Inicializa el juego y crea recursos."""
    pygame.init()
    self.clock = pygame.time.Clock()
❶    self.settings = Settings()

❷    self.screen = pygame.display.set_mode(
        (self.settings.screen_width, self.settings.screen_height))
    pygame.display.set_caption("Alien Invasion")

def run_game(self):
    --fragmento omitido--
    # Redibuja la pantalla en cada paso por el bucle.
❸    self.screen.fill(self.settings.bg_color)

    # Hace visible la última pantalla dibujada.
    pygame.display.flip()
    self.clock.tick(60)
    --fragmento omitido--

```

Importamos `Settings` al archivo de programa principal. A continuación creamos una instancia de `Settings` y se la asignamos a `self.settings` ❶, después de llamar a `pygame.init()`. Cuando creamos una pantalla ❷, usamos los atributos `screen_width` y `screen_height` de `self.settings` y luego `self.settings` para acceder al color de fondo cuando rellenamos la pantalla ❸.

Cuando ejecute `alien_invasion.py` ahora no verá ningún cambio porque lo único que hemos hecho es mover a otro lugar la configuración que ya teníamos. Ya estamos listos para empezar a añadir elementos a la pantalla.

Añadir la imagen de la nave

Vamos a añadir la nave al juego. Para dibujar la nave del jugador en la pantalla, cargaremos una imagen y usaremos el método `blit()` de Pygame para dibujar la imagen.

Cuando escoja material gráfico para sus juegos, asegúrese de prestar atención a las licencias. La forma más segura y económica de empezar es usar gráficos con licencia gratuita para usar y modificar, como los de <https://opengameart.org/>.

Puede emplear prácticamente cualquier tipo de archivo de imagen en su juego, pero lo más fácil es usar un mapa de bits (`.bmp`) porque Pygame carga estos archivos por defecto. Aunque se puede configurar Pygame para que utilice otro tipo de archivos, algunos dependen de determinadas bibliotecas de imágenes que deberían estar instaladas en el ordenador. La mayoría de las imágenes que encontrará estarán en formato `.jpg` o `.png`, pero puede convertirlas en mapas de bits con herramientas como Photoshop, GIMP y Paint.

Preste especial atención al color de fondo de la imagen seleccionada. Busque un archivo con un fondo transparente o sólido que pueda reemplazar con cualquier color de fondo en un editor de imágenes. Sus juegos quedarán mejor si el color de fondo de la imagen coincide con el del juego. Otra opción es hacer que el color de fondo del juego coincida con el de la imagen.

Para Alien Invasion, puede usar el archivo `ship.bmp` (figura 12.1), disponible en los recursos del libro. El color de fondo del archivo coincide con la configuración que estamos usando en este proyecto. Cree una carpeta llamada `images` dentro de la carpeta del proyecto, `alien_invasion`. Guarde el archivo `ship.bmp` en `images`.

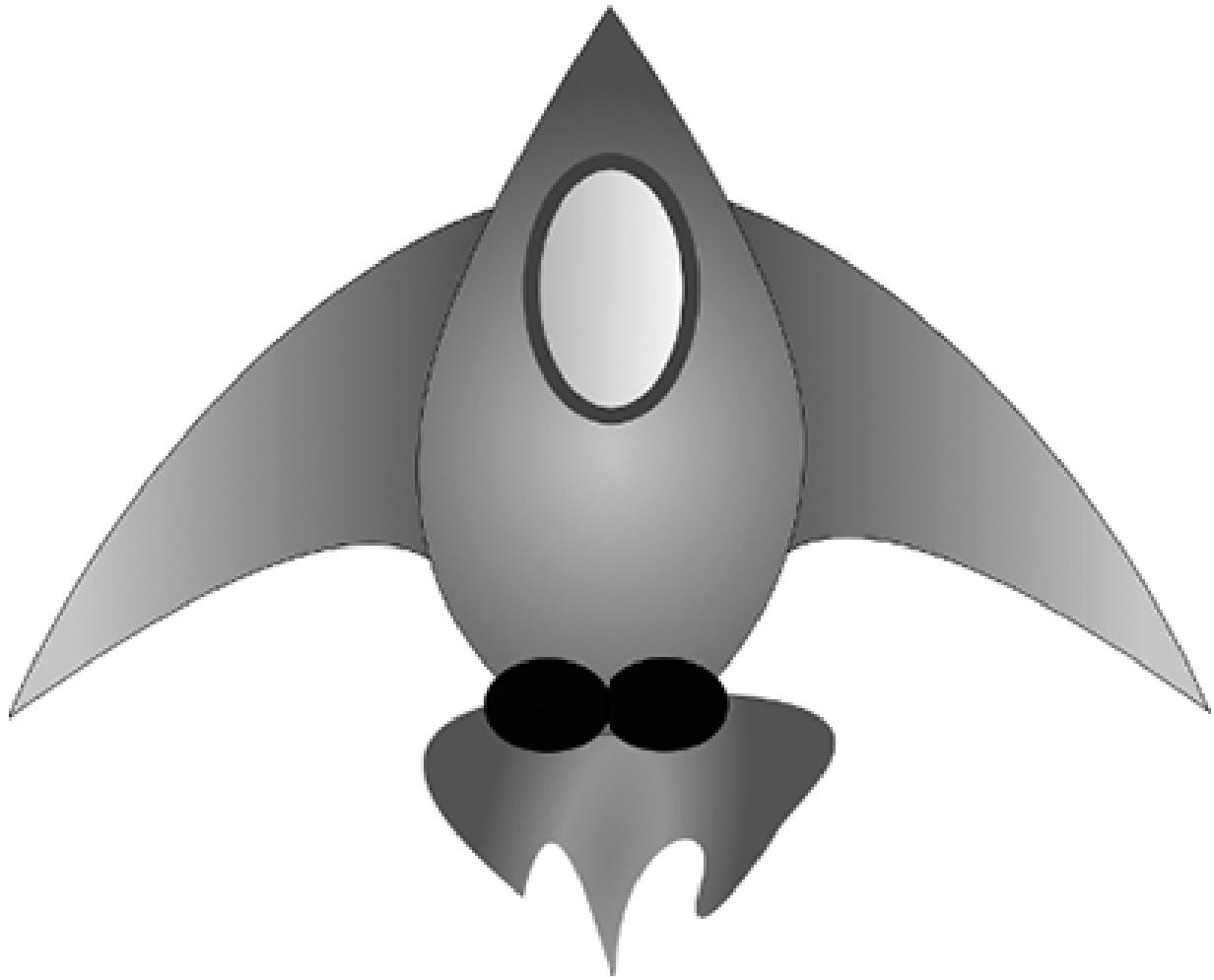


Figura 12.1. La nave de Alien Invasion.

Crear la clase Ship

Una vez elegida una imagen para la nave, tenemos que mostrarla en la pantalla. Para utilizar la nave, crearemos un nuevo módulo `ship` que contendrá la clase `Ship`. Esta clase administrará la mayor parte del comportamiento de la nave del jugador:

ship.py

```
import pygame
```

```
class Ship:
```

```

"""Una clase para gestionar la nave."""

def __init__(self, ai_game):
    """Inicializa la nave y configura su posición inicial."""
    ❶ self.screen = ai_game.screen
    ❷ self.screen_rect = ai_game.screen.get_rect()

    ❸ # Carga la imagen de la nave y obtiene su rect.
    self.image = pygame.image.load('images/ship.bmp')
    self.rect = self.image.get_rect()

    # Coloca inicialmente cada nave nueva en el centro de la parte
    # inferior de la pantalla.
    ❹ self.rect.midbottom = self.screen_rect.midbottom

    ❺ def blitme(self):
        """Dibuja la nave en su ubicación actual."""
        self.screen.blit(self.image, self.rect)

```

Pygame es eficiente porque nos deja tratar todos los elementos del juego como rectángulos ("rects"), aun cuando no tengan precisamente forma rectangular. Tratar un elemento así es eficiente porque los rectángulos son formas geométricas simples. Cuando Pygame necesita averiguar si dos elementos del juego han colisionado, por ejemplo, puede hacerlo más deprisa si trata cada objeto como un rectángulo. Este enfoque suele funcionar lo bastante bien como para que los jugadores no noten que no estamos trabajando con la forma exacta de cada elemento del juego. Trataremos la nave y la pantalla como rectángulos en esta clase.

Importamos el módulo `pygame` antes de definir la clase. El método `__init__()` de `Ship` toma dos parámetros: la referencia `self` y una referencia a la instancia actual de la clase `AlienInvasion`. Esto dará a `ship` acceso a todos los recursos del juego definidos en `AlienInvasion`. Asignamos la pantalla a un atributo de `Ship` ❶ para poder acceder a

ella fácilmente en todos los métodos de la clase. Accederemos al atributo `rect` de la pantalla usando el método `get_rect()` y asignándoselo a `self.screen_rect` ❷. Esto nos permite colocar la nave en la posición correcta en la pantalla.

Para cargar la imagen, llamamos a `pygame.image.load()` ❸ y le proporcionamos la ubicación de nuestra imagen de la nave. Esta función devuelve una superficie que representa la nave y que asignaremos a `self.image`. Cuando se carga la imagen, llamamos a `get_rect()` para acceder al atributo `rect` de la superficie de la nave y poder usarlo luego para colocar el cohete.

Cuando trabajamos con un objeto `rect`, podemos usar las coordenadas `x` e `y` de los bordes superior, inferior, derecho e izquierdo del rectángulo, además del centro, para colocar el objeto. Podemos configurar cualquiera de estos valores para establecer la posición actual del rectángulo. Para centrar un elemento del juego, trabajaremos con los atributos `center`, `centerx` o `centery` del rectángulo. Cuando trabajemos en un borde de la pantalla, usaremos los atributos `top`, `bottom`, `left` o `right`. También hay atributos que combinan estas propiedades, como `midbottom`, `midtop`, `midleft` y `midright`. Al ajustar la ubicación horizontal o vertical del rectángulo, podemos usar los atributos `x` e `y`, que son las coordenadas `x` e `y` de su esquina superior izquierda. Estos atributos nos evitan tener que hacer cálculos que antiguamente los desarrolladores de juegos debían realizar manualmente; los usará a menudo.

Nota: En Pygame, el origen (0, 0) se encuentra en la esquina superior izquierda de la pantalla y las coordenadas aumentan al bajar y moverse hacia la derecha. En una pantalla de 1.200 por 800, el origen está en la esquina superior izquierda y la esquina inferior derecha tiene las coordenadas (1200, 800). Estas coordenadas hacen referencia a la ventana del juego, no a la pantalla física.

Colocaremos la nave en el centro de la parte inferior de la pantalla. Para ello, haremos que el valor de `self.rect.midbottom` coincida con el atributo `midbottom` del rectángulo de la pantalla ❹.

Pygame usa estos atributos `rect` para colocar la nave de manera que esté centrada horizontalmente y alineada con la parte inferior de la pantalla.

Por último, definimos el método `blitme()` ❸ que dibuja la imagen en la pantalla en la posición especificada por `self.rect`.

Dibujar la nave en la pantalla

Vamos a actualizar `alien_invasion.py` para que cree una nave y llame a su método `blitme()`:

alien_invasion.py

```
--fragmento omitido--  
from settings import Settings  
from ship import Ship  
  
class AlienInvasion:  
    """Clase general para administrar los recursos y el  
comportamiento del juego."""  
  
    def __init__(self):  
        --fragmento omitido--  
        pygame.display.set_caption("Alien Invasion")  
  
❶        self.ship = Ship(self)  
  
    def run_game(self):  
        --fragmento omitido--  
        # Redibuja la pantalla en cada paso por el bucle.  
        self.screen.fill(self.settings.bg_color)  
❷        self.ship.blitme()  
  
        # Hace visible la última pantalla dibujada.
```

```
    pygame.display.flip()  
    self.clock.tick(60)  
  
--fragmento omitido--
```

Importamos `Ship` y hacemos una instancia después de crear la pantalla ❶. La llamada a `ship()` requiere un argumento, una instancia de `AlienInvasion`. El argumento `self` se refiere aquí a la instancia actual de `AlienInvasion`. Este es el parámetro que da a `Ship` acceso a los recursos del juego, como, por ejemplo, el objeto `screen`. Asignamos esta instancia de `Ship` a `self.ship`.

Después de llenar el fondo, dibujamos la nave en la pantalla llamando a `ship.blitme()` para que la nave aparezca encima del fondo ❷.

Al ejecutar `alien_invasion.py` ahora, debería aparecer una pantalla de juego vacía, con la nave en el centro de la parte inferior, como se muestra en la figura 12.2.

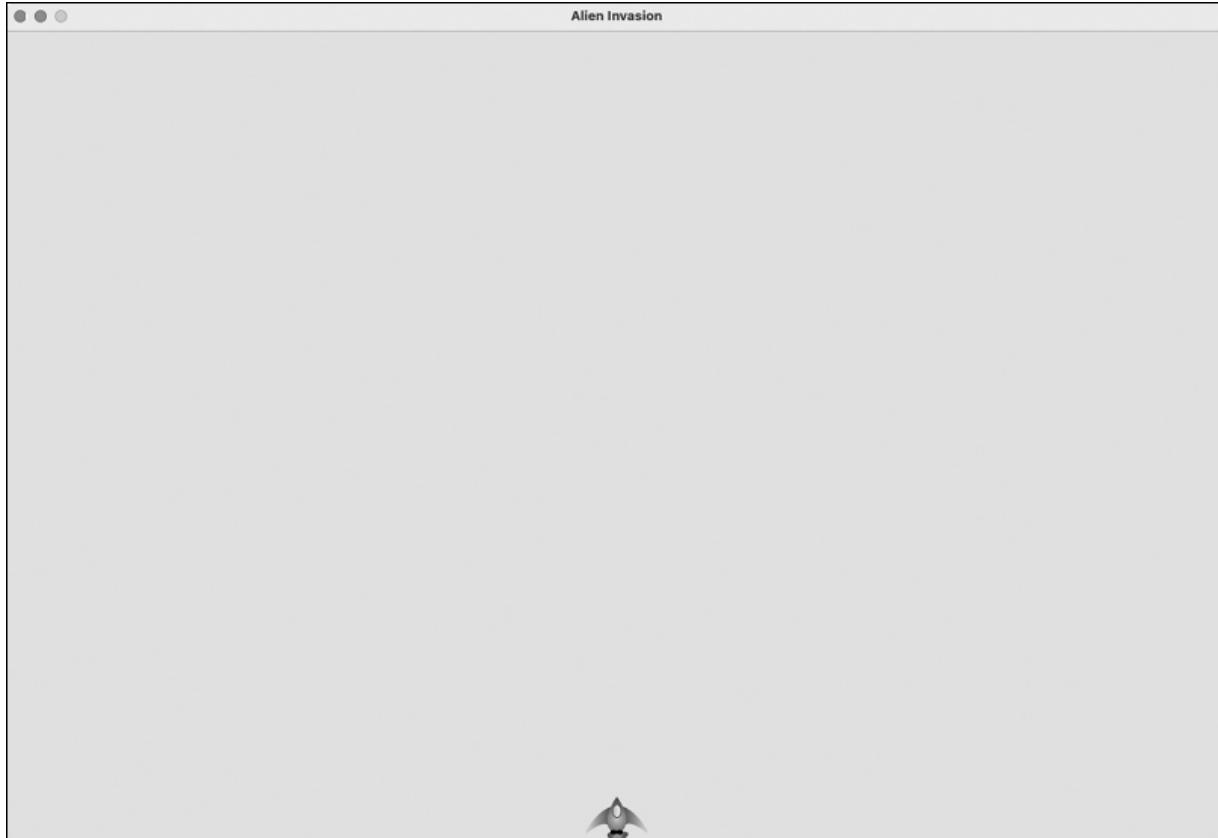


Figura 12.2. Alien Invasion con la nave en el centro de la parte inferior de la pantalla.

Refactorización: Los métodos `_check_events()` y `_update_screen()`

En proyectos grandes, a menudo tendrá que refactorizar el código que ha escrito antes de añadir más. La refactorización simplifica la estructura del código que tenemos escrito, de modo que sea más fácil ampliarlo. En esta sección, dividiremos el método `run_game()`, que se está haciendo bastante largo, en dos "métodos auxiliares". Un método auxiliar trabaja dentro de una clase, pero no se ha creado para ser utilizado por el código externo a dicha clase. En Python, un solo guion bajo inicial indica que estamos ante un método auxiliar.

El método `_check_events()`

Vamos a mover el código que gestiona los eventos a un método aparte llamado `_check_events()`. Esto simplificará `run_game()` y aislará el bucle de gestión de eventos. Este aislamiento nos permite gestionar los eventos independientemente de otros aspectos del juego, como la actualización de la pantalla.

Esta es la clase `AlienInvasion` con el nuevo método `_check_events()`, que solo afecta al código de `run_game()`:

alien_invasion.py

```
def run_game(self):
    """Inicia el bucle principal del juego."""
    while True:
       ❶        self._check_events()

        # Redibuja la pantalla en cada paso por el bucle.
        --fragmento omitido--
```

```
❷ def _check_events(self):
    """Responde a pulsaciones de teclas y eventos de ratón."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Escribimos un nuevo método `_check_events()` ❷ y movemos ahí las líneas que comprueban si el jugador ha hecho clic para cerrar la ventana.

Para llamar a un método desde dentro de una clase, usamos la notación de punto con la variable `self` y el nombre del método ❸. Llamamos al método desde dentro del bucle `while` de `run_game()`.

El método `_update_screen()`

Para simplificar más todavía `run_game()`, moveremos el código para actualizar la pantalla a un método aparte llamado `_update_screen()`:

alien_invasion.py

```
def run_game(self):
    """Inicia el bucle principal del juego."""
    while True:
        self._check_events()
        self._update_screen()
        self.clock.tick(60)

def _check_events(self):
    --fragmento omitido--

def _update_screen(self):
    """Actualiza las imágenes en la pantalla y cambia a la pantalla nueva."""
    self.screen.fill(self.settings.bg_color)
```

```
    self.ship.blitme()  
  
    pygame.display.flip()
```

Hemos movido el código que dibuja el fondo y la nave y cambia la pantalla a `_update_screen()`. Ahora el cuerpo del bucle principal de `run_game()` es mucho más simple. Es fácil ver que, a cada paso por el bucle, estamos buscando eventos nuevos, actualizando la pantalla y haciendo funcionar el reloj. Si ya ha creado unos cuantos juegos, seguramente empezará desmenuzando el código en métodos como estos. Por el contrario, si nunca se ha enfrentado a un proyecto de estas características, es posible que no sepa exactamente cómo estructurar el código en sus fases iniciales. Este enfoque consistente en escribir código que funcione para después reestructurarlo a medida que gana en complejidad nos da una idea de un proceso de desarrollo realista: empezamos escribiendo código lo más sencillo posible, y lo refactorizamos a medida que se complique el proyecto.

Ahora que tenemos el código reestructurado para que sea más fácil ampliarlo, podemos trabajar en los aspectos dinámicos del juego.

PRUEBEO

- **12-1. Cielo azul:** Cree una ventana de Pygame con un fondo azul.
- **12-2. Personaje de juego:** Busque una imagen en mapa de bits de un personaje de juego que le guste o convierta la imagen a mapa de bits. Escriba una clase que dibuje el personaje en el centro de la pantalla y haga coincidir el color de fondo de la imagen con el color de fondo de la pantalla o viceversa.

Pilotar la nave

A continuación, daremos al jugador la capacidad de mover la nave hacia la izquierda y hacia la derecha. Escribiremos código que

responda cuando el jugador pulse las teclas de dirección. Nos concentraremos primero en el movimiento hacia la derecha y luego aplicaremos los mismos principios al movimiento hacia la izquierda. Al añadir este código, aprenderá a controlar el movimiento de imágenes en la pantalla y a responder a entrada de usuario.

Responder a pulsaciones de teclas

Siempre que el jugador pulse una tecla, la pulsación se registrará en Pygame como un evento. El método `pygame.event.get()` recoge todos los eventos. Tenemos que especificar en nuestro método `_check_events()` qué tipo de eventos queremos que compruebe el juego. Cada pulsación de una tecla se registra como un evento `KEYDOWN`.

Cuando Pygame detecta un evento `KEYDOWN`, necesitamos comprobar si la tecla que se ha pulsado es una que desencadena una acción determinada. Por ejemplo, si el jugador pulsa la tecla **Flecha derecha**, queremos aumentar el valor `rect.x` de la nave para moverla a la derecha:

alien_invasion.py

```
def _check_events(self):
    """Responde a pulsaciones de teclado y eventos de ratón."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
       ❶        elif event.type == pygame.KEYDOWN:
           ❷            if event.key == pygame.K_RIGHT:
                # Mueve la nave a la derecha.
           ❸            self.ship.rect.x += 1
```

Dentro de `_check_events()` añadimos un bloque `elif` al bucle de eventos para responder cuando Pygame detecte un evento `KEYDOWN`

- ❶. Comprobamos si la tecla pulsada, `event.key`, es **Flecha derecha**
- ❷. Esta tecla está representada como `pygame.K_RIGHT`. Si se ha pulsado, movemos la nave hacia la derecha aumentando el valor de `self.ship.rect.x` en 1 ❸.

Ahora, al ejecutar `alien_invasion.py`, la nave debería moverse hacia la derecha un píxel cada vez que se pulse la tecla **Flecha derecha**. Es un comienzo, pero no es una forma eficiente de controlar la nave. Vamos a mejorar este control permitiendo el movimiento continuo.

Permitir un movimiento continuo

Cuando el jugador mantenga pulsada la **Flecha derecha**, queremos que la nave siga moviéndose hacia la derecha hasta que el jugador suelte la tecla. Haremos que el juego detecte un evento `pygame.KEYUP` para saber cuándo se suelta la **Flecha derecha**; luego usaremos los eventos `KEYDOWN` y `KEYUP` junto con una bandera llamada `moving_right` para implementar el movimiento continuo. Cuando la bandera `moving_right` sea `False`, la nave no se moverá. Cuando el jugador pulse la **Flecha derecha**, la bandera se pondrá en `True` y, cuando el jugador suelte la tecla, la bandera volverá a `False`.

La clase `Ship` controla todos los atributos de la nave, así que le daremos uno llamado `moving_right` y un método `update()` para comprobar el estado de la bandera `moving_right`. El método `update()` cambiará la posición de la nave si la bandera es `True`. Llamaremos a este método una vez en cada paso por el bucle `while` para actualizar la posición de la nave. Estos son los cambios que haremos en `Ship`:

`ship.py`

```
class Ship:  
    """Una clase para gestionar la nave."""  
  
    def __init__(self, ai_game):
```

```
--fragmento omitido--  
  
# Inicia cada nave nueva en el centro de la parte inferior de  
la pantalla.  
  
self.rect.midbottom = self.screen_rect.midbottom  
  
# Bandera de movimiento; empieza con una bandera que no se  
mueve.  
  
❶ self.moving_right = False  
  
❷ def update(self):  
  
    """Actualiza la posición de la nave en función de la bandera de  
    movimiento."""  
  
    if self.moving_right:  
  
        self.rect.x += 1  
  
def blitme(self):  
  
--fragmento omitido--
```

Añadimos un atributo `self.moving_right` en el método `__init__()` y lo configuramos inicialmente como `False` ❶. Luego añadimos `update()`, que mueve la nave hacia la derecha si la bandera es `True` ❷. Llamamos a este método a través de una instancia de `Ship`, por eso no se considera un método auxiliar.

Ahora tenemos que modificar `_check_events()` para que `moving_right` se establezca como `True` cuando se pulse la **Flecha derecha** y como `False` cuando se suelte la tecla:

alien_invasion.py

```
def _check_events(self):  
  
    """Responde a pulsaciones de teclado y eventos de ratón."""  
  
    for event in pygame.event.get():  
  
--fragmento omitido--
```

```
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_RIGHT:
            ❶          self.ship.moving_right = True
    ❷    elif event.type == pygame.KEYUP:
        if event.key == pygame.K_RIGHT:
            self.ship.moving_right = False
```

Aquí modificamos cómo el juego responde cuando el jugador pulsa la tecla **Flecha derecha**: en vez de cambiar directamente la posición de la nave, solo establecemos `moving_right` en `True` ❶. A continuación, añadimos un nuevo bloque `elif`, que responde a eventos `KEYUP` ❷. Cuando el jugador suelta la tecla (`K_RIGHT`), configuramos `moving_right` en `False`. A continuación, modificamos el bucle `while` de `run_game()` para que llame al método `update()` de la nave en cada paso por el bucle:

alien_invasion.py

```
def run_game(self):
    """Inicia el bucle principal del juego."""
    while True:
        self._check_events()
        self.ship.update()
        self._update_screen()
        self.clock.tick(60)
```

La posición de la nave se actualizará después de comprobar si hay eventos de teclado y antes de actualizar la pantalla. Esto permite actualizar la posición de la nave en respuesta a la entrada del jugador y se asegura de que se utilizará la posición actualizada cuando se dibuje la nave en la pantalla.

Al ejecutar `alien_invasion.py` y mantener pulsada la **Flecha derecha**, la nave debería moverse continuamente hacia la derecha hasta que se suelte la tecla.

Movimiento hacia la izquierda y hacia la derecha

Ahora que la nave se puede mover continuamente hacia la derecha, añadir movimiento hacia la izquierda es bastante fácil. De nuevo, modificaremos la clase `Ship` y el método `_check_events()`. Estos son los cambios relevantes en `__init__()` y `update()` en `Ship`:

`ship.py`

```
def __init__(self, ai_game):
    --fragmento omitido--
    # Banderas de movimiento; comienza con una nave que no está en
    movimiento.
    self.moving_right = False
    self.moving_left = False

def update(self):
    """Actualiza la posición de la nave en función de las banderas de
    movimiento."""
    if self.moving_right:
        self.rect.x += 1
    if self.moving_left:
        self.rect.x -= 1
```

En `__init__()`, añadimos una bandera `self.moving_left`. En `update()`, usamos dos bloques `if` separados en vez de un `elif` para permitir que el valor `rect.x` de la nave se incremente y disminuya cuando se mantengan pulsadas las dos teclas. El resultado es que la nave no se mueve. Si usásemos `elif` para el movimiento hacia la izquierda, la tecla **Flecha derecha** siempre tendría prioridad. Al usar dos bloques `if`, los movimientos son más precisos cuando el jugador mantenga momentáneamente las dos teclas pulsadas al cambiar de dirección.

Tenemos que añadir dos elementos a `_check_events()`:

`alien_invasion.py`

```
def _check_events(self):
    """Responde a pulsaciones de teclado y eventos de ratón."""
    for event in pygame.event.get():
        --fragmento omitido--
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = True

        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                self.ship.moving_right = False
            elif event.key == pygame.K_LEFT:
                self.ship.moving_left = False
```

Si se produce un evento `KEYDOWN` para la tecla `K_LEFT`, establecemos `moving_left` en `True`. Si se produce un evento `KEYUP` para la tecla `K_LEFT`, establecemos `moving_left` en `False`. Podemos usar bloques `elif` aquí porque cada evento está conectado a una sola tecla. Si el jugador pulsa las dos a la vez, se detectarán dos eventos diferentes.

Al ejecutar `alien_invasion.py` de nuevo, deberíamos poder mover la nave continuamente hacia la derecha y hacia la izquierda. Si pulsamos las dos teclas a la vez, la nave debería detenerse. A continuación, refinaremos más el movimiento de la nave. Vamos a ajustar la velocidad y el límite hasta el que puede moverse para que no pueda desaparecer por los lados de la pantalla.

Ajustar la velocidad de la nave

Ahora mismo, la nave se mueve un píxel por ciclo por el bucle `while`, pero podemos refining más el control de la velocidad de la nave añadiendo un atributo `ship_speed` a la clase `Settings`. Utilizaremos este

atributo para determinar hasta dónde puede moverse la nave en cada paso por el bucle. Así queda el nuevo atributo en `settings.py`:

`settings.py`

```
class Settings:
    """Una clase para guardar todas las configuraciones de Alien Invasion."""

    def __init__(self):
        --fragmento omitido--

    # Configuración de la nave.
    self.ship_speed = 1.5
```

Establecemos el valor inicial de `ship_speed` en `1.5`. Ahora, cuando la nave se mueva, su posición se ajusta en 1,5 píxeles en vez de en 1 a cada paso por el bucle.

Usamos flotantes para configurar la velocidad porque nos ofrecerán un control más preciso de la velocidad de la nave cuando aumentemos el tiempo del juego más adelante. Sin embargo, los atributos `rect` como `x` solo almacenan valores enteros, así que tendremos que hacer algunas modificaciones en `Ship`:

`ship.py`

```
class Ship:
    """Una clase para gestionar la nave."""

    def __init__(self, ai_game):
        """Inicializa la nave y establece su posición inicial."""
        self.screen = ai_game.screen


- self.settings = ai_game.settings


--fragmento omitido--
```

```

# Inicia cada nave nueva en el centro de la parte inferior de
# la pantalla.

self.rect.midbottom = self.screen_rect.midbottom

# Guarda un valor decimal para la posición horizontal exacta de
# la nave.

❷ self.x = float(self.rect.x)

# Banderas de movimiento; comienza con una nave que no se está
# moviendo.

self.moving_right = False
self.moving_left = False

def update(self):
    """Actualiza la posición de la nave en función de las banderas
    de movimiento."""
    # Actualiza el valor x de la nave, no el rect.

    if self.moving_right:
❸        self.x += self.settings.ship_speed
    if self.moving_left:
        self.x -= self.settings.ship_speed

    # Actualiza el objeto rect de self.x.

❹        self.rect.x = self.x

def blitme(self):
    --fragmento omitido--

```

Creamos un atributo `settings` para `Ship`, para poder usarlo en `update()`. **❶**. Como estamos ajustando la posición de la nave en fracciones de píxel, necesitamos asignar a la posición una variable que pueda almacenar flotantes. Podemos usar un flotante para

configurar un atributo de `rect`, pero el `rect` solo conservará la porción entera de ese valor. Para hacer un seguimiento preciso de la posición de la nave, definimos un nuevo atributo `self.x` que pueda albergar valores decimales ❷. Usamos la función `float()` para convertir el valor de `self.rect.x` en un flotante y asignamos ese valor a `self.x`.

Ahora, cuando cambiamos la posición de la nave en `update()`, el valor de `self.x` se ajusta en la cantidad almacenada en `settings.ship_speed` ❸. Una vez actualizado `self.x`, usamos el nuevo valor para actualizar `self.rect.x`, que controla la posición de la nave ❹. Solo la porción entera de `self.x` se guardará en `self.rect.x`, pero nos sirve para mostrar la nave.

Ahora podemos cambiar el valor de `ship_speed`, y añadir cualquier valor mayor que 1 hará que la nave se mueva más rápido. Esto contribuye a que la nave responda lo bastante rápido para disparar a los marcianos y nos permitirá cambiar el tempo del juego a medida que el jugador progrese.

Limitar el alcance de la nave

En este punto, si mantenemos las teclas de dirección pulsadas el tiempo suficiente, la nave desaparecerá por cualquiera de los bordes de la pantalla. Vamos a corregir esto para que la nave se detenga al llegar al borde de la pantalla. Lo haremos modificando el método `update()` de `Ship`:

`ship.py`

```
def update(self):  
    """Actualiza la posición de la nave en función de las banderas de  
    movimiento."""  
    # Actualiza el valor x de la nave, no el rect.  
    ❶    if      self.moving_right      and      self.rect.right      <  
        self.screen_rect.right:  
        self.x += self.settings.ship_speed
```

```
❷     if self.moving_left and self.rect.left >0:  
         self.x -= self.settings.ship_speed  
  
        # Actualiza el objeto rect de self.x.  
        self.rect.x = self.x
```

Este código comprueba la posición de la nave antes de cambiar el valor de `self.x`. El código `self.rect.right` devuelve la coordenada `x` del borde derecho del `rect` de la nave. Si este valor es menor que el devuelto por `self.screen_rect.right`, la nave no ha llegado al borde derecho de la pantalla ❶. Lo mismo pasa con el izquierdo: si el valor del lado izquierdo del `rect` es mayor que cero, la nave no ha llegado al borde izquierdo de la pantalla ❷. Esto garantiza que la nave permanece dentro de estos límites antes de ajustar el valor de `self.x`.

Cuando volvamos a ejecutar `alien_invasion.py`, la nave debería dejar de moverse al llegar a cualquiera de los bordes de la pantalla. Está muy bien; lo único que hemos hecho es añadir una prueba condicional en una sentencia `if`, pero da la sensación de que la nave choca con una pared o un campo de fuerza en los bordes de la pantalla.

Refactorización de `_check_events()`

El método `_check_events()` se hará más largo a medida que desarrollemos el juego, así que vamos a dividirlo en dos métodos diferentes: uno que gestione los eventos `KEYDOWN` y otro que maneje los eventos `KEYUP`:

`alien_invasion.py`

```
def _check_events(self):  
    """Responde a pulsaciones de teclado y eventos de ratón."""  
    for event in pygame.event.get():
```

```

if event.type == pygame.QUIT:
    sys.exit()

elif event.type == pygame.KEYDOWN:
    self._check_keydown_events(event)

elif event.type == pygame.KEYUP:
    self._checkkeyup_events(event)

def _check_keydown_events(self, event):
    """Responde a pulsaciones de teclas."""
    if event.key == pygame.K_RIGHT:
        self.ship.moving_right = True
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True

def _checkkeyup_events(self, event):
    """Responde a liberaciones de teclas."""
    if event.key == pygame.K_RIGHT:
        self.ship.moving_right = False
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = False

```

Creamos dos nuevos métodos auxiliares: `_check_keydown_events()` y `_checkkeyup_events()`. Cada uno necesita un parámetro `self` y un parámetro `event`. El cuerpo de estos dos métodos se copia de `_check_events()` y hemos sustituido el código viejo por llamadas a los nuevos métodos. El método `_check_events()` es más sencillo ahora y tiene una estructura más limpia que hará más fácil desarrollar más respuestas a la entrada del jugador.

Pulsar Q para salir

Ahora que estamos respondiendo eficazmente a las pulsaciones de teclado, podemos añadir otra forma de salir del juego. Resulta tedioso tener que hacer clic en la X de la esquina de la ventana del juego cada vez que queremos probar una característica nueva, así

que vamos a añadir un atajo de teclado para salir del juego pulsando **Q**:

alien_invasion.py

```
def _check_keydown_events(self, event):
    --fragmento omitido--
    elif event.key == pygame.K_LEFT:
        self.ship.moving_left = True
    elif event.key == pygame.K_q:
        sys.exit()
```

En `_check_keydown_events()`, añadimos un nuevo bloque que termine el juego cuando el jugador pulse **Q**. Ahora, cuando hagamos pruebas, podemos pulsar esa tecla para cerrar el juego en vez de usar el cursor para cerrar la ventana.

Ejecutar el juego en modo pantalla completa

Pygame tiene un modo pantalla completa que quizás le guste más que ejecutar el juego en una ventana normal. Algunos juegos quedan mejor en pantalla completa y, en algunos sistemas, el juego puede ofrecer un mejor rendimiento en dicho modo.

Para ejecutar el juego en pantalla completa, haga estos cambios en `__init__()`:

alien_invasion.py

```
def __init__(self):
    """Inicializa el juego y crea recursos."""
    pygame.init()
    self.settings = Settings()

❶    self.screen = pygame.display.set_mode((0, 0), pygame.FULLSCREEN)
❷    self.settings.screen_width = self.screen.get_rect().width
```

```
self.settings.screen_height = self.screen.get_rect().height  
pygame.display.set_caption("Alien Invasion")
```

Al crear la superficie de la pantalla, pasamos un tamaño de `(0, 0)` y el parámetro `pygame.FULLSCREEN` ❶. Esto le indica a Pygame que use un tamaño de ventana que rellene toda la pantalla. Como no sabemos el alto y el ancho de la pantalla de antemano, actualizamos estas configuraciones después de crear la pantalla ❷. Utilizamos los atributos `width` y `height` del rectángulo de la pantalla para actualizar el objeto `settings`.

Si le gusta el aspecto y el comportamiento del juego en el modo pantalla completa, mantenga esta configuración. Si le gustaba más el juego en su propia ventana, puede volver a la configuración original en la que especificábamos un tamaño de pantalla concreto para el juego.

Nota: Asegúrese de que puede salir pulsando **Q** antes de ejecutar el juego en pantalla completa; Pygame no ofrece un modo predeterminado de salir del juego cuando se ejecuta en pantalla completa.

Un resumen rápido

En el siguiente apartado, añadiremos la capacidad de disparar, lo que implica la creación de un nuevo archivo llamado `bullet.py` y hacer algunas modificaciones en los archivos que estamos usando. Ahora mismo, tenemos tres archivos que contienen una serie de clases y métodos. Para tener clara la organización del proyecto, vamos a repasar estos archivos antes de añadir más funcionalidad.

`alien_invasion.py`

El archivo principal, `alien_invasion.py`, contiene la clase `AlienInvasion`. Esta clase crea varios atributos importantes que se usan a lo largo del juego: las configuraciones se asignan a `settings`,

la superficie principal a `screen` y también se crea en este archivo una instancia de `ship`. El bucle principal del juego, un bucle `while`, también está guardado en este módulo. El bucle `while` llama a `_check_events()`, `ship.update()` y `_update_screen()`. También controla el reloj con cada pase por el bucle.

El método `_check_events()` detecta eventos relevantes, como pulsaciones y liberaciones de teclas, y los procesa a través de los métodos `_check_keydown_events()` y `_check_keyup_events()`. De momento, estos métodos gestionan el comportamiento de la nave. La clase `AlienInvasion` también contiene `_update_screen()`, que redibuja la pantalla en cada paso por el bucle principal.

El archivo `alien_invasion.py` es el único que necesitamos para jugar a Alien Invasion. Los otros archivos, `settings.py` y `ship.py`, contienen código que se importa a este archivo.

settings.py

El archivo `settings.py` contiene la clase `Settings`. Esta clase solo tiene un método `_init_()`, que inicializa atributos que controlan el aspecto del juego y la velocidad de la nave.

ship.py

El archivo `ship.py` contiene la clase `Ship`, que tiene un método `_init_()`, un método `update()` para gestionar la posición de la nave y un método `blitme()` para dibujar la nave en la pantalla. La imagen de la nave está guardada en `ship.bmp`, que está en la carpeta `images`.

PRUÉBELO

- **12-3. Documentación de Pygame:** Ya tenemos el juego lo bastante avanzado, por lo que conviene echarle un vistazo a la documentación de Pygame. La página de inicio de Pygame es <https://pygame.org/> y la de la documentación, <https://pygame.org/docs/>. De momento puede leerla

por encima. No la necesita para completar este proyecto, pero le será útil cuando quiera modificar Alien Invasion o crear sus propios juegos.

- **12-4. Cohete:** Haga un juego que empiece con un cohete en el centro de la pantalla. Permita al jugador moverlo hacia arriba, hacia abajo, hacia la derecha y hacia la izquierda usando las cuatro teclas de dirección. Asegúrese de que el cohete nunca sobrepasa los bordes de la pantalla.
- **12-5. Teclas:** Haga un archivo de Pygame que cree una pantalla vacía. En el bucle de eventos, imprima el atributo `event.key` siempre que se detecte un evento `pygame.KEYDOWN`. Ejecute el programa y pulse varias teclas para ver cómo responde Pygame.

Disparar balas

Vamos a añadir la capacidad de disparar. Escribiremos código que dispare una bala, representada por un pequeño rectángulo, cuando el jugador pulse la **Barra espaciadora**. Las balas subirán rectas y desaparecerán por la parte superior de la pantalla.

Añadir la configuración de las balas

Al final del método `__init__()`, actualizaremos `settings.py` para incluir los valores que necesitaremos para una nueva clase `Bullet`:

`settings.py`

```
def __init__(self):  
    --fragmento omitido--  
    # Configuración de las balas.  
    self.bullet_speed = 2.0  
    self.bullet_width = 3  
    self.bullet_height = 15  
    self.bullet_color = (60, 60, 60)
```

Esta configuración crea balas de color gris oscuro con una anchura de 3 píxeles y una altura de 15. Las balas se moverán algo más rápido que la nave.

Crear la clase Bullet

Ahora creamos un archivo `bullet.py` para almacenar nuestra clase `Bullet`. Esta es la primera parte de `bullet.py`:

bullet.py

```
import pygame

from pygame.sprite import Sprite


class Bullet(Sprite):
    """Una clase para gestionar las balas disparadas desde la nave."""

    def __init__(self, ai_game):
        """Crea un objeto para la bala en la posición actual de la nave."""
        super().__init__()

        self.screen = ai_game.screen
        self.settings = ai_game.settings
        self.color = self.settings.bullet_color

        # Crea un rectángulo para la bala en (0, 0) y luego establece
        # la posición correcta.
        ❶ self.rect = pygame.Rect(0, 0, self.settings.bullet_width,
                               self.settings.bullet_height)
        ❷ self.rect.midtop = ai_game.ship.rect.midtop

        # Guarda la posición de la bala como flotante.
```