PYTHON COURSO DE CURSO

A HANDS-ON, PROJECT-BASED INTRODUCCIÓN TO PROGRAMMING

ERIC MATTHES



no starch press

www.allitebooks.com

CURSO INTENSIVO DE PYTHON

www.allitebooks.com

www.allitebooks.com

PyTHON COURSODECRASO

Introducción a la programación basada en proyectos

por Eric Matthes



no starch press

San Francisco

www.allitebooks.com

CURSO ACELERADO DE PYTHON. Derechos de autor © 2016 por Eric Matthes.

Reservados todos los derechos. Queda prohibida la reproducción o transmisión total o parcial de esta obra, en cualquier forma o por cualquier medio, electrónico o mecánico, incluidos el fotocopiado, la grabación o cualquier sistema de almacenamiento o recuperación de información, sin la autorización previa por escrito del titular de los derechos de autor y del editor.

Primera edición

19 18 17 16 15 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-603-6
ISBN-13: 978-1-59327-603-4

Editor: William Pollock Editor de
producción: Riley Hoffman
Ilustración de portada: Josh
Ellingson Diseño de interiores:
Octopod Studios
Editores de desarrollo: William Pollock, Liz Chadwick y Leslie Shen Revisor
técnico: Kenneth Love
Correctora: Anne Marie Walker
Compositor: Riley Hoffman Corrector:
James Fraleigh

Para obtener información sobre distribución, traducciones o ventas al por mayor, póngase en contacto

directamente con No Starch Press, Inc: No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
teléfono: 415.863.9900; info@nostarch.com
www.nostarch.com

Biblioteca del Congreso Cataloging-in-Publication Data

Matthes, Eric, 1972-
Curso intensivo de Python : una introducción práctica a la programación basada en proyectos / por Eric Matthes. páginas cm
Incluye índice.
Resumen: "Introducción a la programación en Python basada en proyectos, con ejercicios. Cubre conceptos generales de programación, fundamentos de Python y resolución de problemas. Incluye tres proyectos: cómo crear un videojuego sencillo, cómo utilizar técnicas de visualización de datos para crear gráficos y diagramas, y cómo crear una aplicación web interactiva"-- Proporcionado por el editor.
ISBN 978-1-59327-603-4 -- ISBN 1-59327-603-6
1. Python (Lenguaje de programación informática) I.
Título. QA76.73.P98M38 2015
005.13'3--dc23
2015018135

No Starch Press y el logotipo de No Starch Press son marcas registradas de No Starch Press, Inc. Otros nombres de productos y empresas aquí mencionados pueden ser marcas comerciales de sus respectivos propietarios. En lugar de utilizar un símbolo de marca comercial cada vez que aparece un nombre de marca comercial, utilizamos los nombres únicamente a modo editorial y en beneficio del propietario de la marca comercial, sin intención de infringir la marca comercial.

La información contenida en este libro se distribuye "tal cual", sin garantía. Aunque se han tomado todas las precauciones posibles en la preparación de esta obra, ni el autor ni No Starch Press, Inc. tendrán responsabilidad alguna ante ninguna persona o entidad con respecto a cualquier pérdida o daño causado o supuestamente causado directa o indirectamente por la información contenida en ella.

www.allitebooks.com

Sobre el autor

Eric Matthes es un profesor de ciencias y matemáticas de secundaria que vive en Alaska, donde imparte un curso introductorio de Python. Escribe programas desde que tenía cinco años. En la actualidad, Eric se centra en escribir software que aborde las ineficiencias de la educación y aporte los beneficios del software de código abierto al campo de la educación. En su tiempo libre le gusta escalar montañas y pasar tiempo con su familia.

Acerca del revisor técnico

Kenneth Love ha sido programador y profesor de Python durante muchos años. Ha dado charlas y tutoriales en conferencias, ha impartido formación profesional, ha trabajado como freelance en Python y Django, y ahora enseña para una empresa de educación online. Kenneth es también el co-creador del paquete django-braces, que proporciona varios mixins útiles para las vistas basadas en clases de Django. Puedes seguirle en Twitter en @kennethlove.

www.allitebooks.com

www.allitebooks.com

Para mi padre, que siempre sacaba tiempo
para responder a mis preguntas sobre
programación, y para Ever, que acaba de
empezar a hacerme sus preguntas

www.allitebooks.com

www.allitebooks.com

CONTENIDOS BRIEFOS

Agradecimientos xxvii

Introducción xxix

PARTE I: FUNDAMENTOS 1

Capítulo 1: Primeros pasos 3

Capítulo 2: Variables y tipos de datos simples 19

Capítulo 3: Introducción a las listas 37

Capítulo 4: Trabajar con listas 53

Capítulo 5: Sentencias if 75

Capítulo 6: Diccionarios 95

Capítulo 7: Entrada del usuario y bucles while 117

Capítulo 8: Funciones 133

Capítulo 9: Clases 161

Capítulo 10: Ficheros y excepciones 189

Capítulo 11: Probar el código 215

PARTE II: PROYECTOS 231

Proyecto 1: Invasión alienígena

Capítulo 12: Un barco que dispara balas 235

Capítulo 13: ¡Extraterrestres! 265

Capítulo 14: Puntuación 291

www.allitebooks.com

Proyecto 2: Visualización de datos

Capítulo 15: Generación de datos 321

Capítulo 16: Descarga de datos 349

Capítulo 17: Trabajar con APIs 377

Proyecto 3: Aplicaciones web

Capítulo 18: Primeros pasos con Django 397

Capítulo 19: Cuentas de usuario 427

Capítulo 20: Estilización e implementación de una aplicación 455

Epílogo 483

Apéndice A: Instalación de Python 485

Apéndice B: Editores de texto 491

Apéndice C: Cómo obtener ayuda 499

Apéndice D: Uso de Git para el control de versiones 505

Índice 515

CONTENTESINDETAIL

AGRADECIMIENTOS xxvii

INTRODUCCIÓN xxix

¿Para quién es este libro? xxx
¿Qué puede esperar aprender? xxx
¿Por qué Python? xxxi

PARTE I: FUNDAMENTOS 1

1
CÓMO EMPEZAR 3

Configuración del entorno de programación 3
Python 2 y Python 3 4
Ejecución de fragmentos de código Python 4
Hola a todos 4
Python en diferentes sistemas operativos 5
Python en Linux 5
Python en OS X 8
Python en Windows 10
Resolución de problemas de instalación 15
Ejecutar programas Python desde un terminal 16
En Linux y OS X 16
En Windows 16
Ejercicio 1-1: python.org 17
Ejercicio 1-2: Hola Mundo Errores tipográficos 17
Ejercicio 1-3: Habilidades infinitas 17
Resumen 17

2
VARIABLES Y TIPOS DE DATOS SIMPLES 19

Qué ocurre realmente cuando ejecutas hello_world.py 19
Variables 20
Nombrar y utilizar variables 21
Cómo evitar errores de nombre al utilizar variables 21
Ejercicio 2-1: Mensaje simple 23
Ejercicio 2-2: Mensajes sencillos 23
Cuerdas 23
Cambio de mayúsculas y minúsculas en una cadena con métodos 24
Combinar o concatenar cadenas 25
Añadir espacios en blanco a cadenas con tabulaciones o nuevas líneas 26
Eliminación de espacios en blanco 26
Evitar errores de sintaxis con cadenas 28
Impresión en Python 2 29
Ejercicio 2-3: Mensaje personal 29
Ejercicio 2-4: Nombrar casos 29
Ejercicio 2-5: Cita célebre 29

Ejercicio 2-6: Cita célebre 2 ..... *29*
Ejercicio 2-7: Eliminación de nombres *29*
Números 30
Enteros 30
Flotadores 30
Evitar errores tipográficos con la función str() 31
Números enteros en Python 2 32
Ejercicio 2-8: Número Ocho *33*
Ejercicio 2-9: Número favorito *33*
Comentarios 33
¿Cómo se escriben los comentarios? 33
¿Qué tipo de comentarios debe escribir? 33
Ejercicio 2-10: Añadir comentarios *34*
El zen de Python 34
Ejercicio 2-11: Zen de Python *36*
Resumen 36

3
INTRODUCCIÓN A LAS LISTAS **37**

¿Qué es una lista? 37
Acceso a los elementos de una lista 38
Las posiciones de índice empiezan en 0, no en 1 39
Utilización de valores individuales de una lista 39
Ejercicio 3-1: Nombres *40*
Ejercicio 3-2: Saludos *40*
Ejercicio 3-3: Su propia lista *40*
Modificar, añadir y eliminar elementos 40
Modificación de elementos de una lista 40
Añadir elementos a una lista 41
Eliminar elementos de una lista 42
Ejercicio 3-4: Lista de invitados *46*
Ejercicio 3-5: Modificación de la lista de invitados *46*
Ejercicio 3-6: Más invitados *46*
Ejercicio 3-7: Reducción de la lista de invitados *47*
Organizar una lista 47
Ordenación permanente de una lista con el método sort() 47
Ordenación temporal de una lista con la función sorted() 48
Impresión de una lista en orden inverso 49
Determinar la longitud de una lista 49
Ejercicio 3-8: Ver el mundo *50*
Ejercicio 3-9: Invitados a cenar *50*
Ejercicio 3-10: Cada función *50*
Cómo evitar errores de índice al trabajar con listas 50
Ejercicio 3-11: Error intencionado *52*
Resumen 52

4
TRABAJAR CON LISTAS **53**

Recorrer una lista completa 53
Una mirada más atenta al bucle 54
Hacer más trabajo dentro de un bucle for 55
Hacer algo después de un bucle for 56

Evitar errores de sangría 57
Olvidar las sangrías 57
Olvido de sangrar líneas adicionales 58
Sangrías innecesarias 59
Sangría innecesaria después del bucle 59
Olvidar el colon 60
Ejercicio 4-1: Pizzas *60*
Ejercicio 4-2: Animales *60*
Hacer listas numéricas 61
Uso de la función range() 61
Uso de range() para hacer una lista de números 62
Estadística simple con una lista de números 63
Comprensión de listas 63
Ejercicio 4-3: Contar hasta veinte *64*
Ejercicio 4-4: Un millón *64*
Ejercicio 4-5: Suma de un millón *64*
Ejercicio 4-6: Números impares *64*
Ejercicio 4-7: De tres en tres *64*
Ejercicio 4-8: Cubos *64*
Ejercicio 4-9: Cubo Comprensión *64*
Trabajar con parte de una lista 65
Rebanar una lista 65
Recorrer un corte en bucle 66
Copiar una lista 67
Ejercicio 4-10: Rebanadas *69*
Ejercicio 4-11: Mis pizzas, tus pizzas *69*
Ejercicio 4-12: Más bucles *69*
Tuplas 69
Definición de una tupla 69
Recorrer en bucle todos los valores de una tupla 70
Escribir sobre una tupla 71
Ejercicio 4-13: Buffet *71*
Estilizar su código 72
Guía de estilo 72
Indentación 72
Longitud de línea 73
Líneas en blanco 73
Otras normas de estilo 73
Ejercicio 4-14: PEP 8 ..... *74*
Ejercicio 4-15: Revisión del código *74*
Resumen 74

5
DECLARACIONES SI **75**

Un ejemplo sencillo 76
Pruebas condicionales 76
Comprobación de la igualdad 76
Ignorar mayúsculas y minúsculas al comprobar la igualdad 77
Comprobación de la desigualdad 78
Comparaciones numéricas 78
Comprobación de condiciones múltiples 79
Comprobación de la presencia de un valor en una lista 80

Comprobar si un valor no está en una lista 81
Expresiones booleanas....81
Ejercicio 5-1: Pruebas condicionales 82
Ejercicio 5-2: Más pruebas condicionales 82
Declaraciones if....82
Sentencias if sencillas 82
Sentencias if-else 83
La cadena if-elif-else....84
Uso de múltiples bloques elif 86
Omitir el bloque else 86
Comprobación de condiciones múltiples....87
Ejercicio 5-3: Colores alienígenas nº 1 88
Ejercicio 5-4: Colores alienígenas #2 88
Ejercicio 5-5: Colores alienígenas nº 3 89
Ejercicio 5-6: Etapas de la vida 89
Ejercicio 5-7: Fruta favorita 89
Uso de sentencias if con listas....89
Comprobación de artículos especiales....90
Comprobar que una lista no está vacía 91
Utilización de listas múltiples 92
Ejercicio 5-8: Hola Admin....93
Ejercicio 5-9: Sin usuarios....93
Ejercicio 5-10: Comprobación de nombres de usuario 93
Ejercicio 5-11: Números ordinales....93
Estilo de las sentencias if....94
Ejercicio 5-12: Estilización de sentencias if 94
Ejercicio 5-13: Sus ideas....94
Resumen....94

6
DICCIONARIOS 95

Un diccionario sencillo 96
Trabajar con diccionarios....96
Acceso a los valores de un diccionario....97
Añadir nuevos pares clave-valor 97
Empezar con un diccionario vacío....98
Modificación de valores en un diccionario 99
Eliminación de pares clave-valor....100
Diccionario de objetos similares 100
Ejercicio 6-1: Persona 102
Ejercicio 6-2: Números favoritos....102
Ejercicio 6-3: Glosario 102
Recorrer un diccionario 102
Recorrer en bucle todos los pares clave-valor....103
Recorrer en bucle todas las claves de un diccionario 104
Recorrer en bucle y en orden las claves de un diccionario....106
Recorrer todos los valores de un diccionario....107
Ejercicio 6-4: Glosario 2 108
Ejercicio 6-5: Ríos....108
Ejercicio 6-6: Sondeo....108
Nido....109
Lista de diccionarios 109
Una lista en un diccionario....111

Un diccionario en un diccionario 113
Ejercicio 6-7: Personas *114*
Ejercicio 6-8: Mascotas *115*
Ejercicio 6-9: Lugares favoritos *115*
Ejercicio 6-10: Números favoritos *115*
Ejercicio 6-11: Ciudades *115*
Ejercicio 6-12: Extensiones *115*
Resumen 115

7
ENTRADA DEL USUARIO Y BUCLES WHILE **117**

Funcionamiento de la función input() 118
Escribir Prompts claros 118
Uso de int() para aceptar entradas numéricas 119
El operador módulo 120
Aceptar entradas en Python 2.7 121
Ejercicio 7-1: Coche de alquiler *121*
Ejercicio 7-2: Asientos en un restaurante *121*
Ejercicio 7-3: Múltiplos de diez *121*
Introducción a los bucles while 122
El bucle while en acción 122
Dejar que el usuario elija cuándo salir 122
Utilizar una bandera 124
Uso de break para salir de un bucle 125
Uso de continue en un bucle 126
Evitar bucles infinitos 126
Ejercicio 7-4: Ingredientes de pizza *127*
Ejercicio 7-5: Entradas de cine *127*
Ejercicio 7-6: Tres salidas *128*
Ejercicio 7-7: Infinito *128*
Uso de un bucle while con listas y diccionarios 128
Mover elementos de una lista a otra 128
Eliminación de todas las instancias de valores específicos de una lista 129
Rellenar un diccionario con datos introducidos por el usuario 130
Ejercicio 7-8: Deli *131*
Ejercicio 7-9: Sin pastrami *131*
Ejercicio 7-10: Vacaciones de ensueño *131*
Resumen 131

8
FUNCIONES **133**

Definición de una función 134
Pasar información a una función 134
Argumentos y parámetros 135
Ejercicio 8-1: Mensaje *135*
Ejercicio 8-2: Libro favorito *135*
Pasar argumentos 135
Argumentos posicionales 136
Argumentos por palabras clave 137
Valores por defecto 138

Llamadas a funciones equivalentes....139
Evitar errores de argumentación....140
Ejercicio 8-3: Camiseta....141
Ejercicio 8-4: Camisas grandes....141
Ejercicio 8-5: Ciudades....141
Valores de retorno....141
Devolución de un valor simple....142
Hacer que un argumento sea opcional....142
Devolver un diccionario....144
Uso de una función con un bucle while....145
Ejercicio 8-6: Nombres de ciudades....146
Ejercicio 8-7: Álbum....146
Ejercicio 8-8: Álbumes de usuario....146
Pasar una lista....147
Modificación de una lista en una función....147
Cómo evitar que una función modifique una lista....149
Ejercicio 8-9: Magos....150
Ejercicio 8-10: Grandes magos....150
Ejercicio 8-11: Magos inalterados....150
Pasar un número arbitrario de argumentos....151
Mezcla de argumentos posicionales y arbitrarios....152
Uso de argumentos arbitrarios de palabras clave....152
Ejercicio 8-12: Bocadillos....154
Ejercicio 8-13: Perfil de Usuario....154
Ejercicio 8-14: Automóviles....154
Almacenamiento de funciones en módulos....154
Importación de un módulo completo....154
Importación de funciones específicas....156
Uso de as para asignar un alias a una función....156
Uso de as para asignar un alias a un módulo....157
Importación de todas las funciones de un módulo....157
Funciones de estilismo....158
Ejercicio 8-15: Impresión de modelos....159
Ejercicio 8-16: Importaciones....159
Ejercicio 8-17: Funciones de estilo....159
Resumen....159

9
CLASES **161**

Creación y uso de una clase....162
Creación de la clase perro....162
Crear una instancia a partir de una clase....164
Ejercicio 9-1: Restaurante....166
Ejercicio 9-2: Tres restaurantes....166
Ejercicio 9-3: Usuarios....166
Trabajar con clases e instancias....167
La clase de coches....167
Establecer un valor por defecto para un atributo....168
Modificación de los valores de los atributos....168
Ejercicio 9-4: Número servido....171
Ejercicio 9-5: Intentos de inicio de sesión....171

Herencia....172
El método init () para una clase hija....172
Herencia en Python 2.7....173
Definición de atributos y métodos para la clase hija....174
Sobreescritura de métodos de la clase padre....175
Instancias como atributos....175
Modelado de objetos del mundo real....177
Ejercicio 9-6: Puesto de helados....178
Ejercicio 9-7: Admin....178
Ejercicio 9-8: Privilegios....178
Ejercicio 9-9: Actualización de Batería....178
Importar clases....179
Importar una sola clase....179
Almacenamiento de varias clases en un módulo....180
Importación de múltiples clases desde un módulo....181
Importación de un módulo completo....182
Importación de todas las clases de un módulo....182
Importación de un módulo a otro módulo....183
Encontrar su propio flujo de trabajo....184
Ejercicio 9-10: Restaurante importado....184
Ejercicio 9-11: Admin importado....184
Ejercicio 9-12: Módulos Múltiples....184
La biblioteca estándar de Python....184
Ejercicio 9-13: Reescritura de OrderedDict....186
Ejercicio 9-14: Dados....186
Ejercicio 9-15: Módulo Python de la Semana....186
Clases de estilismo....186
Resumen....187

10
EXPEDIENTES Y EXCEPCIONES 189

Lectura de un archivo....190
Lectura de un archivo completo....190
Rutas de archivos....191
Lectura línea por línea....193
Hacer una lista de líneas de un fichero....194
Trabajar con el contenido de un fichero....194
Archivos grandes: Un millón de dígitos....195
¿Está tu cumpleaños contenido en Pi?....196
Ejercicio 10-1: Aprender Python....197
Ejercicio 10-2: Aprender C....197
Escribir en un archivo....197
Escribir en un archivo vacío....197
Escribir varias líneas....198
Añadir a un fichero....199
Ejercicio 10-3: Invitado....199
Ejercicio 10-4: Libro de visitas....199
Ejercicio 10-5: Programación de encuestas....199
Excepciones....200
Tratamiento de la excepción ZeroDivisionError....200
Uso de bloques try-except....200
Uso de excepciones para evitar colisiones....201

El otro bloque....202
Manejo de la Excepción FileNotFoundError....203
Análisis de textos....204
Trabajar con varios archivos....205
Fracasar en silencio....206
Decidir qué errores notificar....207
Ejercicio 10-6: Suma....207
Ejercicio 10-7: Calculadora de sumas....208
Ejercicio 10-8: Perros y gatos....208
Ejercicio 10-9: Perros y gatos silenciosos....208
Ejercicio 10-10: Palabras comunes....208
Almacenamiento de datos....208
Uso de json.dump() y json.load()....209
Guardar y leer datos generados por el usuario....210
Refactorización....212
Ejercicio 10-11: Número favorito....214
Ejercicio 10-12: Recordar el número favorito....214
Ejercicio 10-13: Verificar Usuario....214
Resumen....214

11
COMPROBACIÓN DEL CÓDIGO 215

Probar una función....216
Pruebas unitarias y casos de prueba....217
Una prueba superada....217
Una prueba fallida....218
Respuesta a una prueba fallida....219
Añadir nuevas pruebas....221
Ejercicio 11-1: Ciudad, País....222
Ejercicio 11-2: Población....222
Probar una clase....222
Una variedad de métodos Assert....222
Una clase para probar....223
Prueba de la clase AnonymousSurvey....225
El método setUp()....227
Ejercicio 11-3: Empleado....228
Resumen....228

PARTE II: PROYECTOS 231

PROYECTO 1: INVASIÓN ALIENÍGENA

12
UN BARCO QUE DISPARA BALAS 235

Planificación del proyecto....236
Instalación de Pygame....236
Instalación de paquetes Python con pip....237
Instalación de Pygame en Linux....238

Instalación de Pygame en OS X....239
Instalación de Pygame en Windows....240
Inicio del proyecto de juego....240
Creación de una ventana Pygame y respuesta a las entradas del usuario....241
Configuración del color de fondo....242
Creación de una clase de configuración....243
Añadir la imagen del barco....244
Creación de la Clase Buque....245
Dibujar el barco en la pantalla....246
Refactorización: el módulo game_functions....247
Función check_events()....247
Función update_screen()....248
Ejercicio 12-1: Cielo azul....249
Ejercicio 12-2: Carácter del juego....249
Pilotar la nave....249
Respuesta a una pulsación de tecla....249
Permitir el movimiento continuo....250
Moverse a derecha e izquierda....252
Ajuste de la velocidad del barco....253
Limitar el alcance del barco....255
Refactorización de check_events()....255
Resumen rápido....256
alien_invasion.py....256
settings.py....256
game_functions.py....256
ship.py....257
Ejercicio 12-3: Cohete....257
Ejercicio 12-4: Claves....257
Balas de tiro....257
Añadir la configuración de la bala....257
Creación de la clase Bullet....258
Almacenamiento de balas en un grupo....259
Balas de disparo....260
Borrar balas antiguas....261
Limitar el número de balas....262
Creación de la función update_bullets()....263
Creación de la función fire_bullet()....264
Ejercicio 12-5: Tirador lateral....264
Resumen....264

13
ALIENS 265

Revisión de su proyecto....266
Creación del primer alienígena....266
Creación de la clase Alien....267
Creación de una instancia del Alien....268
Cómo hacer que el alien aparezca en pantalla....268
Construir la flota alienígena....269
Determinar cuántos alienígenas caben en una fila....269
Creación de filas de alienígenas....270
Creación de la flota....271

www.allitebooks.com

Refactorización de create_fleet() 273
Añadir filas 273
Ejercicio 13-1: Estrellas *276*
Ejercicio 13-2: Estrellas mejores *276*
Traslado de la flota 276
Desplazar a los extranjeros hacia la derecha 276
Creación de ajustes para la dirección de la flota 277
Comprobar si un alienígena ha tocado el borde 278
Abandonar la flota y cambiar de rumbo 278
Ejercicio 13-3: Gotas de lluvia *279*
Ejercicio 13-4: Lluvia constante *279*
Disparando a Aliens 280
Detección de colisiones de balas 280
Fabricación de balas más grandes para pruebas 281
Repoblación de la flota 282
Acelerar las balas 283
Refactorización de update_bullets() 283
Ejercicio 13-5: Captura *284*
Fin del juego 284
Detección de colisiones entre naves extraterrestres 284
Respuesta a las colisiones de naves alienígenas 285
Alienígenas que llegan al fondo de la pantalla 288
Se acabó el juego 288
Identificar cuándo deben ejecutarse las partes del juego 289
Ejercicio 13-6: Game Over *290*
Resumen 290

14
PUNTUACIÓN **291**

Añadir el botón Play 292
Creación de una clase Button 292
Dibujar el botón en la pantalla 294
Inicio del juego 295
Reiniciar el juego 296
Desactivar el botón de reproducción 297
Cómo ocultar el cursor del ratón 298
Ejercicio 14-1: Pulse P para reproducir *298*
Ejercicio 14-2: Práctica de puntería *298*
Subir de nivel 299
Modificación de los ajustes de velocidad 299
Restablecer la velocidad 300
Ejercicio 14-3: Práctica de objetivos desafiantes *301*
Puntuación 301
Visualización de la puntuación 301
Creación de un marcador 303
Actualizar el marcador a medida que se derriban alienígenas 304
Asegurarse de anotar todos los golpes 305
Aumento de los valores de los puntos 306
Redondear el marcador 307
Puntuaciones altas 308

Visualización del nivel....310
Visualización del número de buques....313
Ejercicio 14-4: Máxima puntuación de todos los tiempos....317
Ejercicio 14-5: Refactorización....317
Ejercicio 14-6: Expansión de la invasión alienígena....317
Resumen....317

PROYECTO 2: VISUALIZACIÓN DE DATOS

15
GENERACIÓN DE DATOS 321

Instalación de matplotlib....322
En Linux....322
En OS X....322
En Windows....323
Probando matplotlib....323
Galería matplotlib....323
Trazado de una gráfica lineal simple....324
Cambiar el tipo de etiqueta y el grosor del gráfico....324
Corrección de la trama....326
Trazado y estilo de puntos individuales con scatter()....326
Trazado de una serie de puntos con scatter()....328
Cálculo automático de datos....328
Eliminación de contornos de puntos de datos....329
Definición de colores personalizados....330
Utilización de un mapa de colores....330
Guardar automáticamente las parcelas....331
Ejercicio 15-1: Cubos....331
Ejercicio 15-2: Cubos de colores....331
Paseos al azar....331
Creación de la clase RandomWalk()....332
Elegir la dirección....332
Trazado del paseo aleatorio....333
Generación de múltiples paseos aleatorios....334
Estilizar el paseo....335
Colorear los puntos....335
Trazado de los puntos inicial y final....336
Limpieza de los ejes....337
Añadir puntos de trazado....337
Alterar el tamaño para llenar la pantalla....338
Ejercicio 15-3: Movimiento molecular....339
Ejercicio 15-4: Paseos aleatorios modificados....339
Ejercicio 15-5: Refactorización....339
Tirar dados con Pygal....339
Instalación de Pygal....340
Galería Pygal....340
Creación de la clase de troquel....340
Tirar el dado....341
Análisis de los resultados....341

Creación de un histograma 342
Tirar dos dados 343
Dados rodantes de diferentes tamaños 345
Ejercicio 15-6: Etiquetas automáticas *346*
Ejercicio 15-7: Dos D8 ..... *346*
Ejercicio 15-8: Tres dados *346*
Ejercicio 15-9: Multiplicación *346*
Ejercicio 15-10: Practicando con ambas bibliotecas *346*
Resumen 347

16
DESCARGA DE DATOS **349**

El formato de archivo CSV 350
Análisis de cabeceras de archivos CSV 350
Impresión de las cabeceras y sus posiciones 351
Extracción y lectura de datos 352
Trazado de datos en un gráfico de temperatura 353
El módulo datetime 354
Trazado de fechas 355
Trazar un marco temporal más largo 356
Trazado de una segunda serie de datos 357
Sombreado de un área en el gráfico 358
Comprobación de errores 359
Ejercicio 16-1: San Francisco *362*
Ejercicio 16-2: Comparación Sitka-Valle de la Muerte *362*
Ejercicio 16-3: Precipitaciones *362*
Ejercicio 16-4: Explorar *362*
Asignación de conjuntos de datos globales: Formato JSON 362
Descargar datos sobre la población mundial 362
Extracción de datos relevantes 363
Conversión de cadenas en valores numéricos 364
Obtención de códigos de país de dos cifras 365
Construir un mapa del mundo 367
Trazado de datos numéricos en un mapamundi 368
Trazado de un mapa de población completo 369
Agrupación de países por población 371
Estilizando Mapas del Mundo en Pygal 372
Aligerar el tema del color 374
Ejercicio 16-5: Todos los países *375*
Ejercicio 16-6: Producto Interior Bruto *375*
Ejercicio 16-7: Elija sus propios datos *375*
Ejercicio 16-8: Comprobación del módulo country_codes *375*
Resumen 375

17
TRABAJAR CON APIS **377**

Utilización de una API Web 378
Git y GitHub 378
Solicitud de datos mediante una llamada a la API 378
Instalación de solicitudes 379

Procesamiento de una respuesta API 379
Trabajar con el diccionario de respuestas 380
Resumen de los principales repositorios 382
Supervisión de los límites de la tasa API 383
Visualización de repositorios con Pygal 384
Perfeccionamiento de los gráficos Pygal 386
Añadir información sobre herramientas personalizada 387
Trazado de los datos 388
Añadir enlaces clicables a nuestro gráfico 390
La API de Hacker News 390
Ejercicio 17-1: Otras lenguas 393
Ejercicio 17-2: Debates activos 393
Ejercicio 17-3: Probando python_repos.py 393
Resumen 393

PROYECTO 3: APLICACIONES WEB

18
EMPEZANDO CON DJANGO 397

Creación de un proyecto 398
Escribir una especificación 398
Creación de un entorno virtual 398
Instalación de virtualenv 399
Activación del entorno virtual 399
Instalación de Django 400
Creación de un proyecto en Django 400
Creación de la base de datos 401
Visualización del proyecto 401
Ejercicio 18-1: Nuevos proyectos 402
Iniciar una aplicación 403
Definición de modelos 403
Activación de modelos 404
El sitio de administración de Django 406
Definición del modelo de entrada 408
Migración del modelo de entrada 409
Registro de entrada en el sitio de administración 409
El caparazón de Django 410
Ejercicio 18-2: Entradas cortas 412
Ejercicio 18-3: La API de Django 412
Ejercicio 18-4: Pizzería 412
Haciendo páginas: Página de inicio del diario de aprendizaje 412
Asignación de una URL 413
Escribir una vista 414
Escribir una plantilla 415
Ejercicio 18-5: Planificador de comidas 416
Ejercicio 18-6: Página de inicio de una pizzería 416
Creación de páginas adicionales 416
Herencia de plantillas 416
Página de temas 418
Páginas temáticas individuales 421

Ejercicio 18-7: Documentación de plantillas *424*
Ejercicio 18-8: Pizzería Páginas *424*
Resumen 425

19
CUENTAS DE USUARIO **427**

Permitir a los usuarios introducir datos 428
Añadir nuevos temas 428
Añadir nuevas entradas 432
Edición de entradas 435
Ejercicio 19-1: Blog *438*
Configuración de cuentas de usuario 439
La App de los usuarios 439
Página de inicio de sesión 440
Cierre de sesión 442
Página de registro 443
Ejercicio 19-2: Cuentas de blog *446*
Permitir que los usuarios sean propietarios de sus datos 446
Restringir el acceso con @login_required 447
Conexión de datos a determinados usuarios 448
Restringir el acceso a los temas a los usuarios apropiados 451
Proteger los temas de un usuario 451
Protección de la página edit_entry 452
Asociar nuevos temas al usuario actual 453
Ejercicio 19-3: Refactorización *454*
Ejercicio 19-4: Protección de new_entry *454*
Ejercicio 19-5: Blog protegido *454*
Resumen 454

20
DISEÑO Y DESPLIEGUE DE UNA APLICACIÓN **455**

Registro de aprendizaje de estilismo 456
La aplicación django-bootstrap3 456
Uso de Bootstrap para estilizar el registro de aprendizaje 457
Modificación de base.html 458
Estilizar la página de inicio con un jumbotrón 461
Estilo de la página de inicio de sesión 461
Estilizar la página new_topic 463
Estilizando la Página de Temas 463
Estilizar las entradas de la página de temas 464
Ejercicio 20-1: Otras formas *466*
Ejercicio 20-2: Blog con estilo *466*
Despliegue de Learning Log 466
Crear una cuenta Heroku 466
Instalación de Heroku Toolbelt 466
Instalación de los paquetes necesarios 466
Creación de una lista de paquetes con un archivo requirements.txt 467
Especificación del tiempo de ejecución de Python 468
Modificación de settings.py para Heroku 468
Creación de un Procfile para iniciar procesos 469
Modificación de wsgi.py para Heroku 470

Crear un directorio para archivos estáticos....470
Uso local del servidor gunicorn....470
Uso de Git para rastrear los archivos del proyecto....471
Envío a Heroku....473
Configuración de la base de datos en Heroku....474
Perfeccionamiento de la implantación de Heroku....475
Asegurar el proyecto Live....476
Confirmación y envío de cambios....477
Creación de páginas de error personalizadas....478
Desarrollo en curso....480
La configuración SECRET_KEY....481
Eliminar un proyecto en Heroku....481
Ejercicio 20-3: Blog en directo....*482*
Ejercicio 20-4: Más 404s....*482*
Ejercicio 20-5: Diario de aprendizaje ampliado....*482*
Resumen....482

AFTERWORD **483**

A
INSTALANDO PYTHON **485**

Python en Linux....485
Encontrar la versión instalada....486
Instalación de Python 3 en Linux....486
Python en OS X....486
Encontrar la versión instalada....486
Uso de Homebrew para instalar Python 3....487
Python en Windows....488
Instalación de Python 3 en Windows....488
Encontrar el intérprete de Python....488
Cómo añadir Python a su variable de ruta....489
Palabras clave y funciones incorporadas de Python....489
Palabras clave de Python....489
Funciones incorporadas de Python....490

B
EDITORES DE TEXTO **491**

Geany....492
Instalación de Geany en Linux....492
Instalación de Geany en Windows....492
Ejecutar programas Python en Geany....493
Personalización de la configuración de Geany....493
Texto sublime....494
Instalación de Sublime Text en OS X....494
Instalación de Sublime Text en Linux....494
Instalación de Sublime Text en Windows....495
Ejecutar programas Python en Sublime Text....495
Configuración de Sublime Text....495
Personalización de la configuración de Sublime Text....496

IDLE 496
Instalación de IDLE en Linux 496
Instalación de IDLE en OS X 496
Instalación de IDLE en Windows 497
Personalización de la configuración de IDLE 497
Emacs y vim 497

C
AYUDA 499

Primeros pasos 499
Inténtelo de nuevo 500
Tómate un descanso 500
Consulte los recursos de este libro 500
Búsqueda en línea 501
Stack Overflow 501
Documentación oficial de Python 501
Documentación oficial de la Biblioteca 502
r/learnpython 502
Entradas del blog 502
IRC (Internet Relay Chat) 502
Crear una cuenta IRC 502
Canales a los que unirse 503
Cultura IRC 503

D
USO DE GIT PARA EL CONTROL DE VERSIONES 505

Instalación de Git 506
Instalación de Git en Linux 506
Instalación de Git en OS X 506
Instalación de Git en Windows 506
Configuración de Git 506
Elaboración de un proyecto 507
Ignorar archivos 507
Inicialización de un repositorio 507
Comprobación del estado 508
Añadir ficheros al repositorio 508
Comprometerse 509
Comprobación del registro 509
El Segundo Compromiso 510
Anulación de un cambio 511
Comprobación de confirmaciones anteriores 512
Borrar el repositorio 513

ÍNDICE 515

ACKEWLEDGmENTS

Este libro no habría sido posible sin el maravilloso y extremadamente profesional personal de No Starch Press. Bill Pollock me invitó a escribir un libro introductorio, y agradezco profundamente esa original oferta. Tyler Ortman me ayudó a dar forma a mi pensamiento en las primeras fases de la redacción. Los comentarios iniciales de Liz Chadwick y Leslie Shen sobre cada capítulo fueron inestimables, y Anne Marie Walker me ayudó a aclarar muchas partes del libro. Riley Hoffman respondió a todas mis preguntas sobre el proceso de elaboración de un libro completo y convirtió pacientemente mi trabajo en un bello producto acabado.

Me gustaría dar las gracias a Kenneth Love, el revisor técnico de *Python Crash Course*. Conocí a Kenneth en PyCon un año, y su entusiasmo por el lenguaje y la comunidad Python ha sido una fuente constante de inspiración profesional desde entonces. Kenneth fue más allá de la simple comprobación de hechos y revisó el libro con el objetivo de ayudar a los programadores principiantes a desarrollar una sólida comprensión del lenguaje Python y de la programación en general. Dicho esto, las inexactitudes que pueda haber son completamente mías.

Me gustaría dar las gracias a mi padre por iniciarme en la programación a una edad temprana y por no tener miedo de que rompiera su equipo. A mi mujer, Erin, por apoyarme y animarme a escribir este libro, y a mi hijo, Ever, cuya curiosidad me inspira cada día.

INTRODUCCIÓN



Todo programador tiene una historia sobre cómo aprendió a escribir su primer programa. Yo empecé a aprender de niño, cuando mi padre trabajaba para Digital Equipment Corporation, una de las empresas pioneras de la era informática moderna. Escribí mi primer programa en un que mi padre había montado en el sótano. El ordenador no consistía más que en una placa base conectada a un teclado sin carcasa, y tenía un tubo de rayos catódicos como monitor. Mi programa inicial era un simple juego de adivinar números, que se parecía a esto:

```
Estoy pensando en un número. Intenta adivinar en qué número estoy pensando: 25
¡Demasiado bajo! Adivina de nuevo: 50
¡Demasiado alto! Adivina otra vez: 42
Ya está. ¿Te gustaría volver a jugar? (sí/no) no
Gracias por jugar.
```

www.allitebooks.com

Siempre recordaré lo satisfecho que me sentí viendo a mi familia jugar a un juego que yo había creado y que funcionaba como yo p r e t e n d í a .

Esa experiencia temprana tuvo un impacto duradero. Construir algo con un propósito, algo que resuelva un p r o b l e m a , es una verdadera satisfacción. El software que escribo ahora satisface una necesidad más importante que los esfuerzos de mi infancia, pero la sensación de satisfacción que me produce crear un programa que funciona sigue siendo en gran medida la misma.

¿A quién va dirigido este libro?

El objetivo de este libro es ponerte al día con Python lo más rápidamente posible para que puedas crear programas que funcionen -juegos, visualizaciones de datos y aplicaciones web- mientras desarrollas una base de programación que te servirá para el resto de tu vida. *Python Crash Course* está escrito para personas de cualquier edad que nunca han programado en Python antes o nunca han programado en absoluto. Si quieres aprender los fundamentos de la programación rápidamente para poder centrarte en proyectos interesantes, y te gusta poner a prueba tu comprensión de nuevos conceptos resolviendo problemas significativos, este libro es para ti. *Python Crash Course* también es perfecto para profesores de secundaria y bachillerato que quieran ofrecer a sus alumnos una introducción a la programación basada en proyectos.

¿Qué puede esperar aprender?

El propósito de este libro es convertirte en un buen programador en general y en un buen programador de Python en particular. Aprenderás eficientemente y adoptarás buenos hábitos mientras te proporciono una base sólida en conceptos generales de programación. Después de trabajar en *Python Crash Course*, estarás listo para pasar a técnicas más avanzadas de Python, y tu próximo lenguaje de programación será aún más fácil de entender.

En la primera parte de este libro aprenderás los conceptos básicos de programación que necesitas para escribir programas en Python. Estos conceptos son los mismos que aprenderías al empezar en casi cualquier lenguaje de programación. Aprenderás sobre diferentes tipos de datos y las formas en que puedes almacenarlos en listas y diccionarios dentro de tus programas. Aprenderás a construir colecciones de datos y a trabajar a través de esas colecciones de manera eficiente. Aprenderá a utilizar los bucles `while` e `if` para comprobar ciertas condiciones, de modo que pueda ejecutar secciones específicas de código mientras esas condiciones sean verdaderas y ejecutar otras secciones cuando no lo sean, una técnica que ayuda enormemente a automatizar procesos.

Aprenderá a aceptar entradas de los usuarios para que sus programas sean interactivos y a mantenerlos en ejecución mientras el usuario esté

activo. Explorará cómo escribir funciones para que partes de su programa sean reutilizables, de modo que sólo tenga que escribir bloques de código que realicen ciertas funciones.

una vez, que luego podrás utilizar tantas veces como quieras. Luego extenderás este concepto a comportamientos más complicados con clases, haciendo que programas bastante simples respondan a una variedad de situaciones. Aprenderá a escribir programas que manejen errores comunes con elegancia. Después de trabajar con cada uno de estos conceptos básicos, escribirá algunos programas cortos que resuelven algunos problemas bien definidos. Finalmente, dará su primer paso hacia la programación intermedia aprendiendo a escribir pruebas para su código, de forma que pueda desarrollar sus programas sin preocuparse de introducir errores. Toda la información de la Parte I le preparará para emprender proyectos más grandes y complejos.

En la Parte II aplicarás lo aprendido en la Parte I a tres proyectos. Puedes realizar cualquiera de estos proyectos, o todos ellos, en el orden que más te convenga. En el primer proyecto (Capítulos 12-14) crearás un juego de disparos al estilo Space Invaders llamado Alien Invasion, que consiste en niveles de d i f i c u l t a d creciente. Después de completar este proyecto, estarás en el buen camino para desarrollar tus propios juegos 2D.

El segundo proyecto (capítulos 15-17) le introduce en la visualización de datos. El objetivo de los científicos de datos es dar sentido a la enorme cantidad de información de la que disponen a través de diversas técnicas de visualización. Trabajarás con conjuntos de datos que tú generas mediante código, conjuntos de datos descargados de fuentes en línea y conjuntos de datos que tus programas descargan automáticamente. Después de completar este proyecto, serás capaz de escribir programas que examinen grandes conjuntos de datos y hagan representaciones visuales de la información almacenada.

En el tercer proyecto (capítulos 18-20) construirás una pequeña aplicación web llamada Learning Log. Este proyecto te permite llevar un diario de ideas y conceptos que has aprendido sobre un tema específico. Podrás mantener registros separados para diferentes temas y permitir que otros creen una cuenta e inicien sus propios diarios. También aprenderás a desplegar tu proyecto para que cualquiera pueda acceder a él en línea desde cualquier lugar.

¿Por qué Python?

Cada año me planteo si seguir usando Python o si pasarme a otro lenguaje, quizá uno más nuevo en el mundo de la programación. Pero sigo centrándome en Python por muchas razones. Python es un lenguaje increíblemente eficiente: tus programas harán más en menos líneas de código de lo que requerirían muchos otros lenguajes. La sintaxis de Python también te ayudará a escribir código "limpio". Tu código será fácil de leer, fácil de depurar y fácil de extender y desarrollar en comparación con otros lenguajes.

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

Una de las razones más importantes por las que sigo usando Python es la comunidad Python, que incluye un grupo de personas increíblemente diverso y acogedor. La comunidad es esencial para los programadores porque la programación no es una actividad solitaria. La mayoría de nosotros, incluso los programadores más experimentados, necesitamos pedir consejo a otros que ya han resuelto problemas similares. Tener una comunidad bien conectada y que te apoye es crítico para ayudarte a resolver problemas, y la comunidad Python apoya totalmente a gente como tú que está aprendiendo Python como su primer lenguaje de programación.

Python es un gran lenguaje para aprender, ¡así que empecemos!

PARTE I

B A S IC S

La Parte I de este libro te enseña los conceptos básicos que necesitarás para escribir programas en Python. Muchos de estos conceptos son comunes a todos los lenguajes de programación, por lo que te serán útiles a lo largo de tu vida como programador.

En el **Capítulo 1** instalarás Python en tu ordenador y ejecutarás tu primer programa, que imprime el mensaje *¡Hola mundo!* en la pantalla.

En el **capítulo 2** aprenderás a almacenar información en variables y a trabajar con texto y valores numéricos.

Los capítulos 3 y **4** introducen las listas. Las listas pueden almacenar tanta información como quieras en una variable, permitiéndote trabajar con esos datos de forma eficiente. Podrás trabajar con cientos, miles e incluso millones de valores en unas pocas líneas de código.

En el **Capítulo 5** usarás sentencias `if` para escribir código que responda de una manera si ciertas condiciones son verdaderas, y responda de una manera diferente si esas condiciones no son verdaderas.

El capítulo 6 te muestra cómo usar los diccionarios de Python, que te permiten hacer conexiones entre diferentes piezas de información. Al igual que las listas, los diccionarios pueden contener tanta información como necesites almacenar.

En el **Capítulo 7** aprenderás a aceptar entradas de los usuarios para hacer que tus programas sean interactivos. También aprenderá sobre los bucles `while`, que ejecutan bloques de código repetidamente mientras ciertas condiciones permanezcan verdaderas.

En el **Capítulo 8** escribirás funciones, que son bloques de código con nombre que realizan una tarea específica y pueden ejecutarse siempre

que las necesites.

El capítulo 9 introduce las clases, que te permiten modelar objetos del mundo real, como perros, gatos, personas, coches, cohetes y mucho más, de modo que tu código puede representar cualquier cosa real o abstracta.

El Capítulo 10 te muestra cómo trabajar con archivos y manejar errores para que tus programas no se bloqueen inesperadamente. Almacenarás datos antes de que tu programa se cierre, y los leerás de nuevo cuando el programa se ejecute de nuevo. Aprenderás sobre las excepciones de Python, que te permiten anticipar errores, y hacer que tus programas manejen esos errores con gracia.

En el **capítulo 11** aprenderás a escribir pruebas para tu código, con el fin de comprobar que tus programas funcionan como pretendes. Como resultado, podrá ampliar sus programas sin preocuparse de introducir nuevos errores. Probar tu código es una de las primeras habilidades que te ayudarán en la transición de programador principiante a programador intermedio.

1

GETTING STARTED



En este capítulo ejecutarás tu primer programa Python, *hello_world.py*. Primero, necesitarás para comprobar si Python está instalado en tu ordenador; si no lo está, lo instalarás. A continuación instala también un editor de texto para trabajar con tus programas Python. Los editores de texto reconocen el código Python y resaltan las secciones a medida que escribes, lo que facilita la comprensión de la estructura de tu código.

Configuración del entorno de programación

Python difiere ligeramente en los distintos sistemas operativos, por lo que tendrás que tener en cuenta algunas consideraciones. Aquí, veremos las dos versiones principales de Python actualmente en uso y esbozaremos los

pasos para configurar Python en tu sistema.

Python 2 y Python 3

En la actualidad, existen dos versiones de Python: Python 2 y la más reciente Python 3. Todos los lenguajes de programación evolucionan a medida que surgen nuevas ideas y tecnologías, y los desarrolladores de Python han hecho que el lenguaje sea cada vez más versátil y potente. La mayoría de los cambios son incrementales y apenas perceptibles, pero en algunos casos el código escrito para Python 2 puede no ejecutarse correctamente en sistemas con Python 3 instalado. A lo largo de este libro señalaré las áreas con diferencias significativas entre Python 2 y Python 3, de modo que sea cual sea la versión que utilices, podrás seguir las instrucciones.

Si ambas versiones están instaladas en tu sistema o si necesitas instalar Python, usa Python 3. Si Python 2 es la única versión en tu sistema y prefieres empezar a escribir código en lugar de instalar Python, puedes empezar con Python 2. Pero cuanto antes actualices a Python 3, mejor. Pero cuanto antes te actualices a Python 3 mejor, así trabajarás con la versión más reciente.

Ejecución de fragmentos de código Python

Python viene con un intérprete que se ejecuta en una ventana de terminal, lo que le permite probar partes de Python sin tener que guardar y ejecutar un programa completo.

A lo largo de este libro, verás fragmentos parecidos a este:

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

El texto en negrita es lo que escribirá y ejecutará pulsando ENTER. La mayoría de los ejemplos en el libro son pequeños programas autocontenidos que ejecutarás desde tu editor, porque así es como escribirás la mayor parte de tu código. Pero a veces se mostrarán conceptos básicos en una serie de fragmentos ejecutados a través de una sesión de terminal Python para demostrar conceptos aislados de forma más eficiente. Cada vez que veas los tres corchetes angulares en un listado de código ❶, estás viendo la salida de una sesión de terminal. Intentaremos codificar en el intérprete para tu sistema en un momento.

Hola a todos.

Una creencia muy arraigada en el mundo de la programación ha sido que imprimir un mensaje *¡Hola mundo!* en la pantalla como primer programa en un nuevo lenguaje te traerá suerte.

En Python, puedes escribir el programa *Hola Mundo* en una sola línea:

```
print("¡Hola mundo!")
```

Un programa tan sencillo tiene un propósito muy real. Si se ejecuta correctamente en tu sistema, cualquier programa Python que escribas debería

funcionar también. Veremos cómo escribir este programa en tu sistema particular en un momento.

Python en distintos sistemas operativos

Python es un lenguaje de programación multiplataforma, lo que significa que funciona en los principales sistemas operativos. Cualquier programa Python que escribas debería funcionar en cualquier ordenador moderno que tenga Python instalado. Sin embargo, los métodos para configurar Python en diferentes sistemas operativos varían ligeramente.

En esta sección aprenderás cómo configurar Python y ejecutar el programa *Hola Mundo* en tu propio sistema. Primero comprobarás si Python está instalado en tu sistema y lo instalarás si no lo está. Luego instalarás un editor de texto simple y guardarás un archivo Python vacío llamado *hola_mundo.py*. Finalmente, ejecutarás el programa *Hola Mundo* y solucionarás cualquier p r o b l e m a que no funcione. Te guiaré a través de este proceso para cada sistema operativo, para que tengas un entorno de programación Python amigable para principiantes.

Python en Linux

Los sistemas Linux están diseñados para programar, por lo que Python ya está instalado en la mayoría de los ordenadores Linux. La gente que escribe y mantiene Linux espera que hagas tu propia programación en algún momento y te animan a hacerlo. Por esta razón hay muy poco que tengas que instalar y muy pocas configuraciones que tengas que cambiar para empezar a programar.

Comprobación de la versión de Python

Abre una ventana de terminal ejecutando la aplicación Terminal en tu sistema (en Ubuntu, puedes pulsar CTRL-ALT-T). Para saber si Python está instalado, escribe **python** con una *p* minúscula. Deberías ver una salida que te dice qué versión de Python está instalada y un prompt >>> donde puedes empezar a introducir comandos de Python, como este:

```
$ python
Python 2.7.6 (por defecto, 22 mar 2014,
22:59:38) [GCC 4.8.2] en linux2
Escriba "ayuda", "copyright", "créditos" o "licencia" para obtener más
información.
>>>
```

Esta salida le dice que Python 2.7.6 es actualmente la versión por defecto de Python instalada en esta computadora. Cuando haya visto esta salida, presione CTRL-D o introduzca `exit()` para salir del prompt de Python y volver a un prompt de terminal.

Para comprobar Python 3, es posible que tenga que especificar esa versión; así que incluso si la salida muestra Python 2.7 como la versión por defecto, pruebe el comando `python3`:

```
$ python3
Python 3.5.0 (por defecto, 17 sep 2015,
13:05:18) [GCC 4.8.4] en linux.
```

```
Escriba "ayuda", "copyright", "créditos" o "licencia" para obtener más
información.
>>>
```

Esta salida significa que también tienes Python 3 instalado, por lo que podrás usar cualquiera de las dos versiones. Siempre que veas el comando `python` en este
introduzca `python3` en su lugar. La mayoría de las distribuciones de Linux ya tienen Python instalado, pero si por alguna razón la tuya no lo tiene o si tu sistema viene con Python 2 y quieres instalar Python 3, consulta el Apéndice A.

Instalación de un editor de texto

Geany es un editor de texto simple: es fácil de instalar, le permitirá ejecutar casi todos sus programas directamente desde el editor en lugar de a través de un terminal, utiliza resaltado de sintaxis para colorear su código, y ejecuta su código en una ventana de terminal para que se acostumbre a usar terminales. El Apéndice B proporciona información sobre otros editores de texto, pero recomiendo usar Geany a menos que tenga una buena razón para usar un editor diferente.

Puede instalar Geany en una sola línea en la mayoría de los sistemas Linux:

```
$ sudo apt-get install geany
```

Si esto no funciona, consulte las instrucciones en *http://geany.org/Download/ ThirdPartyPackages/*.

Ejecución del programa "Hola Mundo

Para iniciar tu primer programa, abre Geany. Pulsa la tecla Super (a menudo llamada tecla Windows) y busca Geany en tu sistema. Crea un acceso directo arrastrando el icono a tu barra de tareas o escritorio. Luego crea una carpeta en algún lugar de tu sistema para tus proyectos y llámala *python_work*. (Es mejor utilizar letras minúsculas y guiones bajos para los espacios en los nombres de archivos y carpetas).
porque estas son convenciones de nomenclatura de Python). Vuelve a Geany y guarda un archivo Python vacío (□□□□□□□▸□□□□□□□ **como**) llamado *hola_mundo.py* en tu carpeta de *trabajo python_*. La extensión *.py* le dice a Geany que tu archivo contendrá un archivo Python
programa. También le indica a Geany cómo ejecutar su programa y resaltar el texto de forma útil.

Una vez guardado el archivo, introduzca la siguiente línea:

```
print("¡Hola mundo Python!")
```

Si múltiples versiones de Python están instaladas en tu sistema, necesitas asegurarte de que Geany está configurado para usar la versión correcta. Vaya a **Build▸Set Build Commands**. Deberías ver las palabras *Compilar* y *Ejecutar* con un com-
al lado de cada uno. Geany asume que el comando correcto para cada uno es `python`, pero si su sistema utiliza el comando `python3`, tendrá que cambiarlo.

Si el comando `python3` funcionó en una sesión de terminal, cambia los comandos Compilar y Ejecutar para que Geany use el intérprete

Python 3. Su comando Compile debería verse así:

```
python3 -m py_compile "%f"
```

Tienes que escribir este comando exactamente como se muestra. Asegúrate de que los espacios y las mayúsculas coinciden con lo que se muestra aquí.

Su comando Ejecutar debería tener este aspecto:

```
python3 "%f"
```

De nuevo, asegúrese de que el espaciado y las mayúsculas coinciden con lo que se muestra aquí. La Figura 1-1 muestra cómo deberían verse estos comandos en el menú de configuración de Geany.

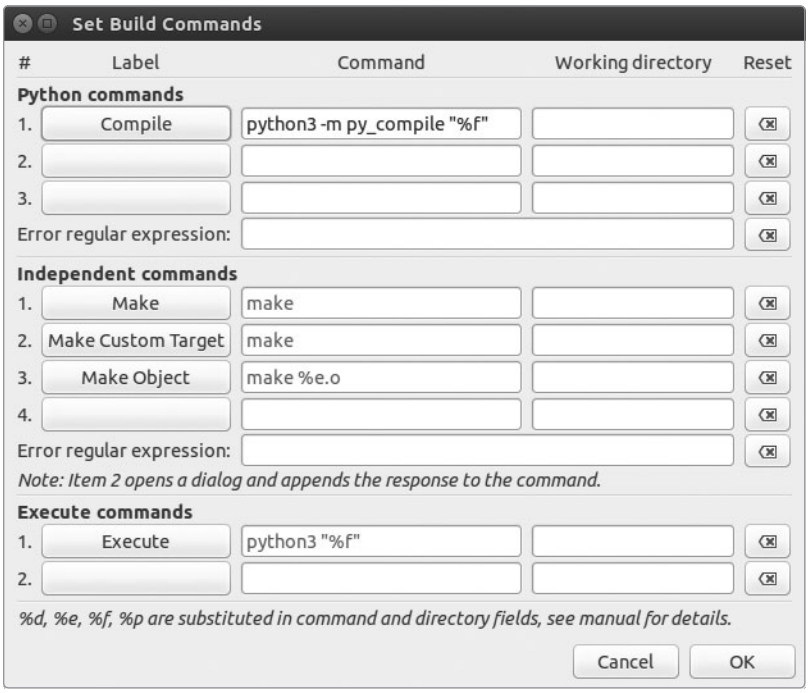


Figura 1-1: Aquí, Geany está configurado para usar Python 3 en Linux.

Ahora ejecute *hello_world.py* seleccionando **Build▸Execute** en el menú, haciendo clic en el icono Ejecutar (que muestra un conjunto de engranajes), o pulsando F5.

Debería aparecer una ventana de terminal con la siguiente salida:

```
¡Hola mundo Python!

-----------------
(el programa sale con el código:
0) Pulse Retorno para continuar.
```

Si no lo ves, comprueba cada carácter de la línea que has introducido. ¿Ha escrito accidentalmente en `mayúsculas`? ¿Olvidaste una o ambas comillas o paréntesis? Los lenguajes de programación esperan una sintaxis muy específica, y si no la proporcionas, obtendrás errores. Si no puede

ejecutar el programa, consulte "Resolución de problemas de instalación" en la página 15.

Ejecutar Python en una sesión de terminal

Puedes probar a ejecutar fragmentos de código Python abriendo un terminal y escribiendo `python` o `python3`, como hiciste al comprobar tu versión. Hazlo de nuevo, pero esta vez introduce la siguiente línea en la sesión de terminal:

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

Deberías ver tu mensaje impreso directamente en la ventana de terminal actual. Recuerde que puede cerrar el intérprete de Python pulsando CTRL-D o escribiendo el comando `exit()`.

Python en OS X

Python ya está instalado en la mayoría de los sistemas OS X. Una vez que sepas que Python está instalado, tendrás que instalar un editor de texto y asegurarte de que está configurado correctamente.

Comprobar si Python está instalado

Abra una ventana de terminal yendo a **Aplicaciones▸Utilidades▸Terminal**. También puede pulsar la barra espaciadora COMANDO, escribir **terminal** y pulsar INTRO.
Para saber si Python está instalado, introduzca **python con una** *p* minúscula. Debería ver una salida que le indica qué versión de Python está instalada en su sistema y un `>>>` prompt donde puede empezar a introducir comandos de Python, como esto:

```
$ python
Python 2.7.5 (por defecto, 9 mar 2014, 22:15:05)
[GCC 4.2.1 compatible Apple LLVM 5.0 (clang-500.0.68)] en darwin
Escriba "ayuda", "copyright", "créditos" o "licencia" para obtener más
información.
>>>
```

Esta salida le dice que Python 2.7.5 es actualmente la versión por defecto instalada en esta computadora. Cuando haya visto esta salida, pulse CTRL-D o introduzca `exit()` para salir del prompt de Python y volver a un prompt de terminal.

Para comprobar si tienes Python 3, prueba el comando `python3`. Puede que aparezca un mensaje de error, pero si la salida muestra que tienes Python 3 instalado, podrás usar Python 3 sin tener que instalarlo. Si `python3` funciona en tu sistema, siempre que veas el comando `python` en este libro, asegúrate de usar `python3` en su lugar. Si por alguna razón tu sistema no viene con Python o si sólo tienes Python 2 y quieres instalar Python 3 ahora, consulta el Apéndice A.

Ejecutar Python en una sesión de terminal

Puedes probar a ejecutar fragmentos de código Python abriendo un terminal y escribiendo `python` o `python3`, como hiciste al comprobar tu versión. Hazlo de nuevo, pero esta vez introduce la siguiente línea en la sesión de terminal:

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

Deberías ver tu mensaje impreso directamente en la ventana de terminal actual. Recuerde que puede cerrar el intérprete de Python pulsando CTRL-D o escribiendo el comando `exit()`.

Instalar un editor de texto

Sublime Text es un editor de texto simple: es fácil de instalar en OS X, te permitirá ejecutar casi todos tus programas directamente desde el editor en lugar de a través de un terminal, utiliza resaltado de sintaxis para colorear tu código, y ejecuta tu código en una sesión de terminal incrustada en la ventana de Sublime Text para que sea fácil ver la salida. El Apéndice B proporciona información sobre otros editores de texto, pero recomiendo usar Sublime Text a menos que tengas una buena razón para usar un editor diferente.

Puede descargar un instalador para Sublime Text desde *http://sublimetext.com/3*. Haga clic en el enlace de descarga y busque un instalador para OS X. Sublime Text tiene una política de licencias muy liberal: puedes usar el editor gratis todo el tiempo que quieras, pero el autor te pide que compres una licencia si te gusta y quieres usarlo continuamente. Una vez descargado el instalador, ábrelo y arrastra el icono de Sublime Text a la carpeta *Aplicaciones*.

Configuración de Sublime Text para Python 3

Si usas un comando distinto de `python` para iniciar una sesión de terminal Python, necesitarás configurar Sublime Text para que sepa dónde encontrar la versión correcta de Python en tu sistema. Ejecuta el siguiente comando para averiguar la ruta completa a tu intérprete de Python:

```
$ type -a python3
python3 es /usr/local/bin/python3
```

Ahora abre Sublime Text, y ve a **Tools▸Build System▸New Build System**, que abrirá un nuevo archivo de configuración para ti. Borre lo que ver e introducir lo siguiente:

Python3.sublime-build

```
{
    "cmd": ["/usr/local/bin/python3", "-u", "$archivo"],
}
```

Este código le dice a Sublime Text que use el comando `python3` de su sistema cuando ejecute el archivo actualmente abierto. Asegúrate de usar la ruta que encontraste al emitir el comando `type -a python3` en el paso anterior. Guarda el archivo como *Python3.sublime-build* en el directorio por defecto que Sublime Text abre cuando eliges Guardar.

Ejecución del programa "Hola Mundo

Para iniciar tu primer programa, ejecuta Sublime Text abriendo la carpeta *Aplicaciones* y haciendo doble clic en el icono de Sublime Text. También puedes pulsar la barra espaciadora COMANDO e introducir **Sublime Text** en la barra de búsqueda que aparece.

Crea una carpeta llamada *python_work* en algún lugar de tu sistema para tus proyectos. (Es mejor usar letras minúsculas y guiones bajos en lugar de espacios en los nombres de archivos y carpetas, porque son convenciones de nomenclatura de Python). Guarda un archivo Python vacío (**File▸Save As**) llamado *hello_world.py* en su *python_work* carpeta. La extensión *.py* le indica a Sublime Text que su archivo contendrá un archivo Python y le indica cómo ejecutar su programa y resaltar el texto de forma útil.

Una vez guardado el archivo, introduzca la siguiente línea:

```
print("¡Hola mundo Python!")
```

Si el comando `python` funciona en su sistema, puede ejecutar su programa seleccionando **Herramientas▸Construir** en el menú o pulsando CTRL-B. Si configuró Sublime Text para usar un comando distinto de `python`, seleccione **Herramientas▸Sistema de construcción** y luego seleccione **Python 3**. Esto establece Python 3 como el versión por defecto de Python, y podrá seleccionar **Tools▸Build** o simplemente pulse COMANDO-B para ejecutar sus programas a partir de ahora.

Debería aparecer una pantalla de terminal en la parte inferior de la ventana de Sublime Text, mostrando la siguiente salida:

```
¡Hola mundo Python!
[Terminado en 0.1s]
```

Si no lo ves, comprueba cada carácter de la línea que has introducido. ¿Ha escrito accidentalmente en `mayúsculas`? ¿Olvidaste una o ambas comillas o paréntesis? Los lenguajes de programación esperan una sintaxis muy específica, y si no la proporcionas, obtendrás errores. Si no puede ejecutar el programa, consulte "Solución de problemas de instalación" en la página 15.

Python en Windows

Windows no siempre viene con Python, por lo que probablemente tendrás que descargarlo e instalarlo, y luego descargar e instalar un editor de texto.

Instalación de Python

Primero, comprueba si Python está instalado en tu sistema. Abra una ventana de comandos introduciendo **command** en el menú Inicio o manteniendo pulsada la tecla MAYÚS mientras hace clic con el botón derecho del ratón en el escritorio y selecciona **Abrir ventana de comandos aquí**. En la ventana de terminal, introduce **python** en minúsculas. Si aparece un prompt de Python (>>>), Python está instalado en tu sistema. Sin embargo, es probable que aparezca un mensaje de error indicando que `python` no es un comando reconocido.

En ese caso, descarga un instalador de Python para Windows. Vaya a *http:// python.org/downloads/*. Deberías ver dos botones, uno para descargar Python 3 y otro para descargar Python 2. Haz clic en el botón Python 3, que debería iniciar automáticamente la descarga del instalador correcto para tu sistema. Una vez descargado el archivo, ejecuta el instalador. Asegúrese de marcar la opción Add Python to PATH, que le facilitará la configuración correcta de su sistema. La Figura 1-2 muestra esta opción marcada.

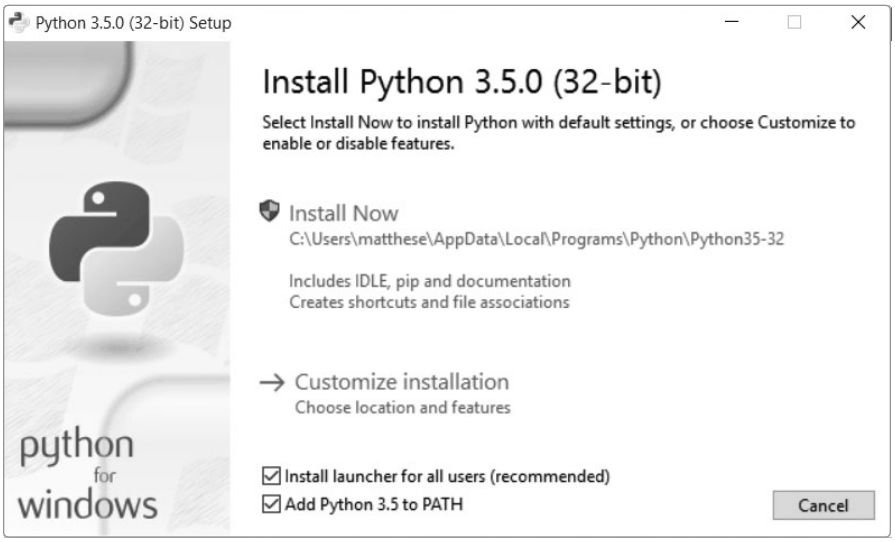


Figura 1-2: Asegúrese de marcar la casilla Añadir Python a PATH.

Iniciar una sesión de terminal con Python

La configuración de tu editor de texto será sencilla si primero configuras tu sistema para ejecutar Python en una sesión de terminal. Abre una ventana de comandos e introduce **python** en minúsculas. Si aparece un prompt de Python (>>>), Windows ha encontrado la versión de Python que acabas de instalar:

```
C:³> python
Python 3.5.0 (v3.5.0:374f501f4567, 13 sep 2015, 22:15:05) [MSC v.1900 32 bit
(Intel)] en win32
Escriba "ayuda", "copyright", "créditos" o "licencia" para obtener más
información.
```

Si esto funcionó, puedes pasar a la siguiente sección, "Ejecutar Python en una sesión de terminal".

Sin embargo, es posible que vea una salida más parecida a ésta:

```
C:³> python
'python' no se reconoce como comando interno o externo, programa operable o
archivo por lotes.
```

En este caso necesitas decirle a Windows cómo encontrar la versión de Python que acabas de instalar. El comando `python` de tu sistema suele estar guardado en tu unidad *C,* así que abre el Explorador de Windows y abre tu unidad *C*. Busca una carpeta que empiece por el nombre *Python*, abre esa carpeta y encuentra el archivo *python* (en minúsculas). Por ejemplo, tengo una carpeta *Python35* con un archivo llamado *python* dentro de ella, así que la ruta al comando `python` en mi sistema es *C:\Python35\ python*. Si no, introduce **python** en el cuadro de búsqueda del Explorador de Windows para que te muestre exactamente dónde está almacenado el comando `python` en tu sistema.

Cuando crea que conoce la ruta, pruébela introduciendo esa ruta en un archivo

ventana de terminal. Abra una ventana de comandos e introduzca la ruta completa que acaba de encontrar:

```
C:\> C:\Python35\python
Python 3.5.0 (v3.5.0:374f501f4567, 13 sep 2015, 22:15:05) [MSC v.1900 32 bit
(Intel)] en win32
Escriba "ayuda", "copyright", "créditos" o "licencia" para obtener más
información.
>>>
```

Si esto ha funcionado, ya sabes cómo acceder a Python en tu sistema.

Ejecutar Python en una sesión de terminal

Introduce la siguiente línea en tu sesión de Python, y asegúrate de ver la salida *¡Hola mundo Python!*

```
>>> print("¡Hola mundo Python!")
¡Hola mundo Python!
>>>
```

Cada vez que quieras ejecutar un fragmento de código Python, abre una ventana de comandos e inicia una sesión de terminal Python. Para cerrar la sesión de terminal, pulsa CTRL-Z y luego ENTER, o introduce el comando `exit()`.

Instalación de un editor de texto

Geany es un editor de texto simple: es fácil de instalar, le permitirá ejecutar casi todos sus programas directamente desde el editor en lugar de a través de una terminal, utiliza resaltado de sintaxis para colorear su código, y ejecuta su

código en una ventana de terminal para que se acostumbre a usar terminales. El Apéndice B proporciona información sobre otros editores de texto, pero recomiendo usar Geany a menos que tenga una buena razón para usar un editor diferente.

Puede descargar un instalador de Windows para Geany desde *http://geany.org/*.

Haz clic en **Releases** en el menú Download, y busca el instalador *geany-1.25_setup.exe* o algo similar. Ejecute el instalador y acepte todos los valores predeterminados.

Para iniciar tu primer programa, abre Geany: pulsa la tecla Windows y busca Geany en tu sistema. Deberías hacer un acceso directo arrastrando el icono a tu barra de tareas o escritorio. Crea una carpeta llamada *python_work en* algún lugar de tu sistema para tus proyectos. (Es mejor usar letras minúsculas y guiones bajos para los espacios en los nombres de archivos y carpetas, porque son convenciones de nomenclatura de Python). Vuelve a Geany y guarda un archivo Python vacío (**File▸Save As**) llamado *hello_world.py* en tu carpeta *python_work*. La extensión *.py* le dice a Geany que su archivo contendrá un programa Python. También le dice a Geany cómo ejecutar su programa y resaltar el texto de forma útil.

Después de guardar el archivo, escriba la siguiente línea:

```
print("¡Hola mundo Python!")
```

Si el comando `python` funcionó en su sistema, no tendrá que configurar Geany; sáltese la siguiente sección y pase a "Ejecutar el programa Hola Mundo" en la página 14. Si necesitara introducir una ruta como *C:\Python35\python* para iniciar un intérprete de Python, siga las instrucciones de la siguiente sección para configurar Geany para su sistema.

Configuración de Geany

Para configurar Geany, vaya a **Build▸Set Build Commands**. Deberías ver las palabras *Compilar* y *Ejecutar* con un comando al lado de cada una. El comando Compilar y Ejecutar comandos comienzan con `python` en minúsculas, pero Geany no sabe donde tu sistema almacenó el comando `python`. Necesitas añadir la ruta que usaste en la sesión de terminal.

En los comandos Compilar y Ejecutar, añada la unidad en la que se encuentra su comando `python` y la carpeta en la que se almacena el comando `python`. Su comando Compilar debe ser algo como esto:

```
C:\Python35\python -m py_compile "%f"
```

Su ruta puede ser un poco diferente, pero asegúrese de que los espacios y las mayúsculas coinciden con lo que se muestra aquí.

Su comando Ejecutar debería tener este aspecto:

```
C:\Python35\python "%f"
```

De nuevo, asegúrese de que el espaciado y las mayúsculas en su comando Ejecutar coinciden con lo que se muestra aquí. La Figura 1-3 muestra cómo deberían verse estos comandos en el menú de configuración

de Geany.

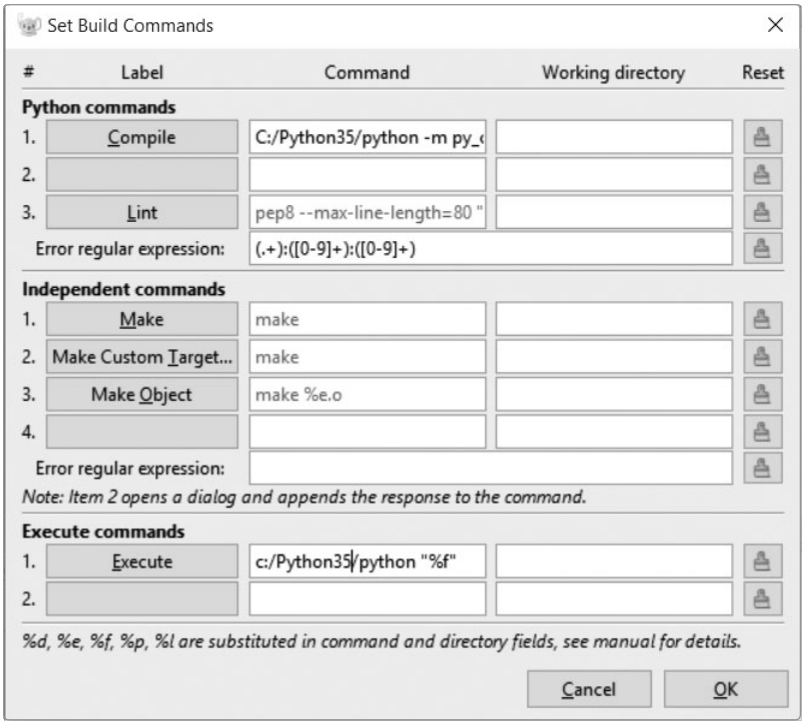


Figura 1-3: Aquí, Geany está configurado para usar Python 3 en Windows.

Una vez que hayas configurado correctamente estos comandos, haz clic en **Aceptar**.

Ejecución del programa "Hola Mundo

Ahora debería poder ejecutar su programa con éxito. Ejecute *hello_world.*py seleccionando **Build▸Execute** en el menú, haciendo clic en el icono Ejecutar (que muestra un conjunto de engranajes), o pulsando F5. Una ventana de terminal debería
con el siguiente resultado:

```
¡Hola mundo Python!

------------------.
(el programa sale con el código:
0) Pulse Retorno para continuar.
```

Si no lo ves, comprueba cada carácter de la línea que has introducido. ¿Ha escrito accidentalmente en `mayúsculas`? ¿Olvidaste una o ambas comillas o paréntesis? Los lenguajes de programación esperan una sintaxis muy específica, y si no la proporcionas, obtendrás errores. Si no puede ejecutar el programa, consulte la siguiente sección para obtener ayuda.

Resolución de problemas de instalación

Con suerte, la configuración de tu entorno de programación ha sido un éxito, pero si no has podido ejecutar *hello_world.py*, aquí tienes algunos remedios que puedes probar:

- Cuando un programa contiene un error significativo, Python muestra un *rastreo*. Python busca en el archivo e intenta informar del problema. El rastreo puede darle una pista sobre qué problema está impidiendo que el programa se ejecute.
- Aléjese del ordenador, tómese un breve descanso y vuelva a intentarlo. Recuerde que la sintaxis es muy importante en programación, por lo que incluso la falta de dos puntos, unas comillas mal colocadas o unas paréntesis mal colocadas pueden impedir que un programa se ejecute correctamente. Vuelva a leer las partes más importantes de este capítulo, repase lo que ha hecho y vea si puede encontrar el error.
- Empieza de nuevo. Probablemente no necesites desinstalar nada, pero podría tener sentido borrar tu archivo *hello_world.py* y crearlo de nuevo desde cero.
- Pide a otra persona que siga los pasos de este capítulo, en tu ordenador o en otro distinto, y observa atentamente lo que hace. Puede que te hayas saltado un pequeño paso que otra persona haya detectado.
- Encuentra a alguien que sepa Python y pídele que te ayude a configurarlo. Si preguntas, es posible que conozcas a alguien que utilice Python.
- Las instrucciones de configuración de este capítulo también están disponibles en línea, a través de https://www.nostarch.com/pythoncrashcourse/. Es posible que la versión en línea de estas instrucciones le resulte más útil.
- Pide ayuda en línea. El Apéndice C ofrece una serie de recursos y áreas en línea, como foros y sitios de chat en directo, donde puedes pedir soluciones a personas que ya han resuelto el problema al que te enfrentas.

No te preocupes por molestar a los programadores experimentados. Todos los programadores se han atascado alguna vez, y la mayoría estarán encantados de ayudarte a configurar tu sistema correctamente. Siempre y cuando puedas decir claramente lo que estás tratando de hacer, lo que ya has intentado, y los resultados que estás obteniendo, hay una buena probabilidad de que alguien sea capaz de ayudarte. Como se mencionó en la Introducción, la comunidad Python es muy amigable con los principiantes.

Python debería funcionar bien en cualquier ordenador moderno, así que busca la forma de pedir ayuda si tienes problemas hasta ahora. Los primeros problemas pueden ser frustrantes, pero vale la pena resolverlos. Una vez que tengas *hello_world.py* funcionando, puedes empezar a aprender Python, y tu trabajo de programación será más interesante y satisfactorio.

Ejecutar programas Python desde un terminal

La mayoría de los programas que escriba en su editor de texto los ejecutará directamente desde el editor, pero a veces es útil ejecutar programas desde un terminal. Por ejemplo, puede que quieras ejecutar un programa existente sin abrirlo para editarlo.

Puedes hacer esto en cualquier sistema con Python instalado si sabes cómo acceder al directorio donde has guardado tu archivo de programa. Para intentarlo, asegúrate de que has guardado el archivo *hello_world.py* en la carpeta *python_work* de tu escritorio.

En Linux y OS X

Ejecutar un programa Python desde una sesión de terminal es lo mismo en Linux que en OS X. El comando de terminal `cd`, para *cambiar directorio*, se utiliza para navegar a través de su sistema de archivos en una sesión de terminal. El comando `ls`, para *listar*, muestra todos los archivos no ocultos que existen en el directorio actual.

Abra una nueva ventana de terminal y ejecute los siguientes comandos *hola_mundo.py*:

```
❶ ~$ cd Escritorio/python_work/
❷ ~/Desktop/python_work$ ls
hola_mundo.py
❸ ~/Desktop/python_work$ python hola_mundo.py
¡Hola mundo Python!
```

En ❶ usamos el comando `cd` para navegar a la carpeta *python_work*, que está en la carpeta *Desktop*. A continuación, utilizamos el comando `ls` para asegurarnos de que hello_world.*py* se encuentra en esta carpeta ❷. A continuación, ejecutamos el archivo utilizando el comando `python hello_world.py` ❸.

Es así de sencillo. Basta con utilizar el comando `python` (o `python3`) para ejecutar programas Python.

En Windows

El comando de terminal `cd`, para *cambiar de directorio*, se utiliza para navegar por el sistema de archivos en una ventana de comandos. El comando `dir`, de *directorio, le* muestra todos los archivos que existen en el directorio actual.

Abra una nueva ventana de terminal y ejecute los siguientes comandos *hola_mundo.py*:

```
❶ C:\> cd Escritorio\python_work
❷ C:\Desktop\python_work> dir
hola_mundo.py
❸ C:\Desktop\python_work> python hola_mundo.py
¡Hola mundo Python!
```

En ❶ usamos el comando `cd` para navegar a la carpeta *python_work*, que está en la carpeta *Desktop*. A continuación, utilizamos el comando `dir` para asegurarnos de que hello_world.*py* se encuentra en esta carpeta ❷. Luego, ejecutamos el archivo usando el comando `python hello_world.py` ❸.

Si no ha configurado su sistema para utilizar el comando simple `python`, es posible que tenga que utilizar la versión más larga de este comando:

```
C:\$ cd Desktop\python_work
C:\Desktop\python_work$ dir
hola_mundo.py
C:\Desktop\python_work$ C:\Python35\python hola_mundo.py
¡Hola mundo Python!
```

La mayoría de tus programas funcionarán bien directamente desde tu editor, pero a medida que tu trabajo se vuelva más complejo, puede que escribas programas que necesites ejecutar desde un terminal.

TRY IT yOURSELF

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

1-1. python.org: Explora la página principal de Python (http://python.org/) para encontrar temas que te interesen. A medida que te familiarices con Python, diferentes partes del sitio te serán más útiles.

1-2. Errores tipográficos en Hello World: Abre el archivo hola_mundo.*py* que acabas de crear. Escribe una errata en alguna parte de la línea y vuelve a ejecutar el programa. ¿Puedes hacer una errata que genere un error? ¿Puedes entender el mensaje de error? ¿Puedes hacer una errata que no genere un error? ¿Por qué crees que no generó un error?

1-3. Habilidades infinitas: Si tuvieras infinitas habilidades de programación, ¿qué construirías? Estás a punto de aprender a programar. Si tienes un objetivo final en mente, tendrás un uso inmediato para tus nuevas habilidades; ahora es un buen momento para redactar descripciones de lo que te gustaría crear. Es un buen hábito llevar un cuaderno de "ideas" al que puedas recurrir siempre que quieras empezar un nuevo proyecto. Dedica unos minutos a describir tres programas que te gustaría crear.

Resumen

En este capítulo aprendiste un poco sobre Python en general, e instalaste Python en tu sistema si aún no estaba allí. También instalaste un editor de texto para facilitar la escritura de código Python. Aprendiste a ejecutar fragmentos de código Python en una sesión de terminal, y ejecutaste tu primer programa real, *hello_world.py*. Probablemente también aprendiste algo sobre resolución de problemas.

En el próximo capítulo aprenderás sobre los diferentes tipos de datos con los que puedes trabajar en tus programas Python, y también aprenderás a usar variables.

www.allitebooks.com

2

VARIABLES Y SIMPLE DATA TYPES



En este capítulo aprenderás sobre los diferentes tipos de datos con los que puedes trabajar en tus programas Python. También aprenderás cómo para almacenar sus datos en variables y cómo utilizar esas variables en sus programas.

Qué ocurre realmente cuando ejecutas hello_world.py

Echemos un vistazo más de cerca a lo que hace Python cuando ejecutas *hola_mundo.py*. Resulta que Python hace una buena cantidad de trabajo, incluso cuando ejecuta un programa simple:

hola_mundo.py

```
print("¡Hola mundo Python!")
```

Cuando ejecutes este código, deberías ver esta salida:

```
¡Hola mundo Python!
```

Cuando ejecutas el archivo *hola_mundo.py*, la terminación *.py* indica que el archivo es un programa Python. Su editor entonces ejecuta el archivo a través del *intérprete de Python,* que lee el programa y determina lo que cada palabra del programa. Por ejemplo, cuando el intérprete ve la palabra `print`, imprime en la pantalla lo que haya dentro de los paréntesis.

A medida que escribes tus programas, tu editor resalta diferentes partes de tu programa de diferentes maneras. Por ejemplo, reconoce que `print` es el nombre de una función y muestra esa palabra en azul. Reconoce que "¡Hola mundo Python!" no es código Python y muestra esa frase en naranja. Esta característica se llama *resaltado de sintaxis* y es muy útil cuando empiezas a escribir tus propios programas.

Variables

Intentemos usar una variable en *hello_world.py*. Añade una nueva línea al principio del archivo, y modifica la segunda línea:

```
message = "¡Hola mundo Python!"
print(message)
```

Ejecute este programa para ver qué ocurre. Debería ver la misma salida que vio anteriormente:

```
¡Hola mundo Python!
```

Hemos añadido una *variable* llamada `mensaje`. Cada variable contiene un *valor*, que es la información asociada a esa variable. En este caso el valor es el texto "¡Hola mundo Python!".

Añadir una variable supone un poco más de trabajo para el intérprete de Python. Cuando procesa la primera línea, asocia el texto "¡Hola mundo Python!" con la variable `mensaje`. Cuando llega a la segunda línea, imprime en pantalla el valor asociado al `mensaje`.

Vamos a ampliar este programa modificando hello_world.*py* para imprimir un segundo mensaje. Añade una línea en blanco a *hola_mundo.py*, y luego añade dos nuevas líneas de código:

```
message = "¡Hola mundo Python!"
print(message)

message = "¡Hola mundo Python Crash Course!"
print(message)
```

Ahora, cuando ejecutes *hello_world.py*, deberías ver dos líneas de salida:

```
¡Hola mundo Python!
¡Hola mundo Python Crash Course!
```

Puedes cambiar el valor de una variable en tu programa en cualquier momento, y Python siempre mantendrá un registro de su valor actual.

Nombrar y utilizar variables

Cuando usas variables en Python, necesitas adherirte a algunas reglas y guías. Romper algunas de estas reglas causará errores; otras líneas guía sólo te ayudarán a escribir código más fácil de leer y entender. Asegúrate de tener en cuenta las siguientes reglas sobre variables:

- Los nombres de las variables sólo pueden contener letras, números y guiones bajos. Pueden empezar por una letra o un guión bajo, pero no por un número. Por ejemplo, puedes llamar a una variable *mensaje_1* pero no *1_mensaje*.
- No se permiten espacios en los nombres de las variables, pero se pueden utilizar guiones bajos para separar palabras en los nombres de las variables. Por ejemplo, *greeting_message* funciona, pero *greeting message* provocará errores.
- Evite usar palabras clave de Python y nombres de funciones como nombres de variables; es decir, no use palabras que Python haya reservado para un propósito pro- gramatical particular, como la palabra `print`. (Vea "Palabras clave de Python y funciones incorporadas" en la página 489.)
- Los nombres de las variables deben ser cortos pero descriptivos. Por ejemplo, *nombre es mejor* que *n*, *nombre_alumno* es mejor que *s_n*, y *longitud_del_nombre* es mejor que *longitud_del_nombre_de_personas*.
- Tenga cuidado al utilizar la letra minúscula *l* y la mayúscula *O* porque podrían confundirse con los números *1* y *0*.

Puede llevar algo de práctica aprender a crear buenos nombres de variables, especialmente a medida que tus programas se vuelven más interesantes y complicados. A medida que escribas más programas y empieces a leer el código de otras personas, mejorarás en la creación de nombres significativos.

NOTE *Las variables Python que utilices en este punto deben estar en minúsculas. No tendrás errores si usas mayúsculas, pero es una buena idea evitar usarlas por ahora.*

Evitar errores de nombre al utilizar variables

Todo programador comete errores, y la mayoría los comete a diario. Aunque los buenos programadores pueden cometer errores, también saben cómo responder a esos errores de forma eficaz. Veamos un error que es probable

que cometas al principio y aprendamos a solucionarlo.

Escribiremos un código que genere un error a propósito. Introduce el siguiente código, incluyendo la palabra mal escrita *mesage* mostrada en negrita:

```
message = "¡Hola lector del Curso acelerado de
Python!" print(mesage)
```

Cuando se produce un error en su programa, el intérprete de Python hace todo lo posible para ayudarle a averiguar dónde está el problema. El intérprete proporciona una traza cuando un programa no puede ejecutarse con éxito. Un *rastreo* es un registro de dónde el intérprete tuvo problemas al intentar ejecutar tu código. Este es un ejemplo del rastreo que Python proporciona después de que accidentalmente hayas escrito mal el nombre de una variable:

```
Traceback (última llamada más reciente):
❶  File "hello_world.py", line 2, in <module>.
❷    print(mensaje)
❸ NameError: el nombre 'mesage' no está definido
```

La salida en ❶ informa de que se produce un error en la línea 2 del fichero *hello_world.py*. El intérprete muestra esta línea para ayudarnos a detectar el error rápidamente ❷ y nos dice qué tipo de error ha encontrado ❸. En este caso encontró un *error de nombre* e informa que la variable que se está imprimiendo, `mesage`, no ha sido definida. Python no puede identificar el nombre de la variable proporcionada. Un error de nombre generalmente significa que olvidamos definir el valor de una variable antes de usarla, o que cometimos un error ortográfico al ingresar el nombre de la variable.

Por supuesto, en este ejemplo omitimos la letra *s* en el `mensaje` del nombre de la variable en la segunda línea. El intérprete de Python no revisa la ortografía de tu código, pero se asegura de que los nombres de las variables estén escritos de forma consistente. Por ejemplo, observa lo que ocurre cuando escribimos *mensaje* incorrectamente también en otro lugar del código:

```
mesage = "¡Hola lector del Curso acelerado de
Python!" print(mesage)
```

En este caso, el programa se ejecuta correctamente.

```
¡Hola lector del Curso acelerado de Python!
```

Los ordenadores son estrictos, pero no tienen en cuenta la buena y la mala ortografía. En consecuencia, no es necesario tener en cuenta las reglas ortográficas y gramaticales inglesas a la hora de crear nombres de variables y escribir código.

Muchos errores de programación son simples erratas de un solo carácter en una línea de un programa. Si pasa mucho tiempo buscando uno

de estos errores, sepa que está en buena compañía. Muchos programadores experimentados y con talento pasan horas buscando este tipo de pequeños errores. Intenta reírte de ello y sigue adelante, sabiendo que ocurrirá con frecuencia a lo largo de tu vida como programador.

NOTE *La mejor manera de entender nuevos conceptos de programación es intentar utilizarlos en sus programas. Si se queda atascado mientras trabaja en un ejercicio de este libro, intente hacer otra cosa durante un rato. Si sigue atascado, repase la parte correspondiente de ese capítulo. Si sigue necesitando ayuda, consulte las sugerencias del Apéndice C.*

TRY IT yOURSELF

Escriba un programa separado para realizar cada uno de estos ejercicios. Guarde cada programa con un nombre de archivo que siga las convenciones estándar de Python, utilizando letras minúsculas y guiones bajos, como *simple_mensaje.py* y *simple_mensajes.py*.

2-1. Mensaje simple: Almacena un mensaje en una variable, y luego imprime ese mensaje.

2-2. Mensajes simples: Almacene un mensaje en una variable e imprima ese mensaje. Luego cambie el valor de su variable a un nuevo mensaje, e imprima el nuevo mensaje.

Cuerdas

Dado que la mayoría de los programas definen y recogen algún tipo de datos, y luego hacen algo útil con ellos, es útil clasificar los diferentes tipos de datos. El primer tipo de datos que veremos es la cadena. Las cadenas son bastante simples a primera vista, pero puedes utilizarlas de muchas maneras diferentes.

Una *cadena* es simplemente una serie de caracteres. Cualquier cosa dentro de comillas se considera una cadena en Python, y puedes usar comillas simples o dobles alrededor de tus cadenas de esta manera:

```
"Esto es una cuerda".
"Esto también es una
cuerda".
```

Esta flexibilidad le permite utilizar comillas y apóstrofes dentro de sus cadenas:

```
Le dije a mi amigo: "¡Python es mi lenguaje favorito!"'.
"El lenguaje 'Python' se llama así por Monty Python, no por la
serpiente". "Uno de los puntos fuertes de Python es su comunidad
diversa y solidaria".
```

Exploremos algunas de las formas en que se pueden utilizar las cadenas.

Cambio de mayúsculas y minúsculas en una cadena con métodos

Una de las tareas más sencillas que puedes hacer con cadenas es cambiar las mayúsculas y minúsculas de las palabras de una cadena. Mira el siguiente código e intenta determinar qué está pasando:

nombre.py

```
nombre = "ada lovelace"
print(nombre.titulo())
```

Guarde este archivo como *nombre.py* y ejecútelo. Usted debe ver esta salida:

```
Ada Lovelace
```

En este ejemplo, la cadena en minúsculas `"ada lovelace"` se almacena en el `nombre de la variable`. El método `title()` aparece después de la variable en el estado `print()`. Un *método* es una acción que Python puede realizar sobre un dato. El punto (`.`) después de `nombre` en `nombre.título()` le dice a Python que haga que el `método título()` actúe sobre la variable `nombre`. Cada método va seguido de un conjunto de paréntesis, porque los métodos a menudo necesitan información adicional para hacer su trabajo.
Esa información se proporciona dentro de los paréntesis. La función `title()` no necesita información adicional, por lo que sus paréntesis están vacíos.

`title()` muestra cada palabra en mayúscula inicial. Esto es útil porque a menudo querrá pensar en un nombre como una pieza de información. Por ejemplo, puede que quiera que su programa reconozca los valores de entrada `Ada`, `ADA` y `ada` como el mismo nombre, y los muestre todos como `Ada`.

También existen otros métodos útiles para tratar las mayúsculas y minúsculas. Por ejemplo, puede cambiar una cadena a mayúsculas o minúsculas de la siguiente manera:

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

Aparecerá lo siguiente:

```
ADA LOVELACE
ada lovelace
```

El método `lower()` es particularmente útil para almacenar datos. Muchas veces no querrá confiar en las mayúsculas que le proporcionen sus usuarios, por lo que convertirá las cadenas a minúsculas antes de almacenarlas. Luego, cuando quiera mostrar la información, utilizará el caso que tenga más sentido para cada cadena.

Combinar o concatenar cadenas

A menudo resulta útil combinar cadenas. Por ejemplo, es posible que desee almacenar un nombre y un apellido en variables separadas, y luego combinarlas cuando desee mostrar el nombre completo de alguien:

```
first_name = "ada"
last_name = "lovelace"
❶ full_name = first_name + " " + last_name

print(nombre_completo)
```

Python utiliza el símbolo más (+) para combinar cadenas. En este ejemplo, usamos `+` para crear un nombre completo combinando un `nombre`, un espacio y un `apellido` ❶, dando este resultado:

```
ada lovelace
```

Este método de combinar cadenas se denomina *concatenación*. Puedes utilizar la concatenación para componer mensajes completos utilizando la información que has almacenado en una variable. Veamos un ejemplo:

```
first_name = "ada"
last_name = "lovelace"
nombre_completo = nombre + " " + apellido

❶ print("¡Hola, " + nombre_completo.title() + "!")
```

Aquí, el nombre completo se utiliza ❶ en una frase que saluda al usuario, y el método `title()` se utiliza para formatear el nombre adecuadamente. Este código devuelve un saludo sencillo pero con un formato agradable:

```
¡Hola, Ada Lovelace!
```

Puede utilizar la concatenación para componer un mensaje y, a continuación, almacenar el mensaje completo en una variable:

```
first_name = "ada"
last_name = "lovelace"
nombre_completo = nombre + " " + apellido

❶ message = "¡Hola, " + nombre_completo.title() + "!"
❷ print(mensaje)
```

Este código también muestra el mensaje "¡Hola, Ada Lovelace!", pero almacenar el mensaje en una variable en ❶ hace que la sentencia `print` final en ❷ sea mucho más sencilla.

Añadir espacios en blanco a cadenas con tabulaciones o nuevas líneas

En programación, los espacios en *blanco* se refieren a cualquier carácter no impreso, como espacios, tabulaciones y símbolos de fin de línea. Puedes utilizar los espacios en blanco para organizar tu salida de forma que sea más fácil de leer para los usuarios.

Para añadir un tabulador al texto, utilice la combinación de caracteres `\t` como se muestra en ❶:

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

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

```
>>> print("Idiomas:PythonCavaScript")
Idiomas:
Python
C
JavaScript
```

También puedes combinar tabuladores y nuevas líneas en una sola cadena. La cadena `"\n\t"` le dice a Python que se mueva a una nueva línea, y comience la siguiente línea con un tabulador. El siguiente ejemplo muestra cómo se puede utilizar una cadena de una sola línea para generar cuatro líneas de salida:

```
>>> print("Idiomas:PythonJavaScript")
Idiomas:
    Python
    C
    JavaScript
```

Las nuevas líneas y los tabuladores serán muy útiles en los dos próximos capítulos, cuando empiece a producir muchas líneas de salida a partir de unas pocas líneas de código.

Eliminar espacios en blanco

Los espacios en blanco adicionales pueden crear confusión en los programas. Para los programadores, `'python'` y `'python '` son prácticamente iguales. Pero para un programa, son dos cadenas diferentes. Python detecta el espacio extra en `'python '` y lo considera significativo a menos que le digas lo contrario.

Es importante tener en cuenta los espacios en blanco, porque a menudo querrá comparar dos cadenas para determinar si son iguales. Por ejemplo, un caso importante podría ser la comprobación de los nombres de usuario de las personas que acceden a un sitio web. Los espacios en blanco adicionales t a m b i é n pueden confundir en situaciones mucho más

sencillas. Afortunadamente, Python facilita la eliminación de los espacios en blanco de los datos que se introducen.

Python puede buscar espacios en blanco adicionales en los lados derecho e izquierdo de una cadena. Para asegurarse de que no existen espacios en blanco en el extremo derecho de una cadena, utilice el método `rstrip()`.

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

El valor almacenado en `favorite_language` en ❶ contiene un espacio en blanco extra al final de la cadena. Cuando preguntas a Python por este valor en una sesión de terminal, puedes ver el espacio al final del valor ❷. Cuando el método `rstrip()` actúa sobre la variable `favorite_language` en ❸, este espacio extra es eliminado. Sin embargo, sólo se elimina temporalmente. Si se vuelve a pedir el valor de `favorite_language`, se puede ver que la cadena tiene el mismo aspecto que cuando se introdujo, incluido el espacio en blanco extra ❹.

Para eliminar los espacios en blanco de la cadena de forma permanente, tienes que volver a almacenar el valor eliminado en la variable:

```
>>> lenguaje_favorito = 'python '
❶ >>> idioma_favorito = idioma_favorito.rstrip()
>>> idioma_favorito
python
```

Para eliminar los espacios en blanco de la cadena, se eliminan los espacios en blanco del lado derecho de la cadena y luego se almacena ese valor de nuevo en la variable original, como se muestra en ❶. Cambiar el valor de una variable y luego almacenar el nuevo valor de nuevo en la variable original se hace a menudo en la programación. Así es como el valor de una variable puede cambiar a medida que se ejecuta un programa o en respuesta a la entrada del usuario.

También puede eliminar los espacios en blanco de la parte izquierda de una cadena utilizando la función `lstrip()` o eliminar los espacios en blanco de ambos lados a la vez con `strip()`:

```
❶ >>> favorite_language = ' python '
❷ >>> lengua_favorita.rstrip()
' python'
❸ >>> idioma_favorito.lstrip()
python
❹ >>> favorite_language.strip()
python
```

En este ejemplo, empezamos con un valor que tiene espacios en blanco al principio y al final ❶. A continuación, eliminamos el espacio sobrante del lado derecho

en ❷, desde el lado izquierdo en ❸, y desde ambos lados en ❹. Experimentar con estas funciones puede ayudarte a familiarizarte con la manipulación de cadenas. En el mundo real, estas funciones de extracción se utilizan más a menudo para limpiar la entrada del usuario antes de que se almacene en un programa.

Evitar errores de sintaxis con cadenas

Un tipo de error que puedes ver con cierta regularidad es un error de sintaxis. Un error de *sintaxis* ocurre cuando Python no reconoce una sección de tu programa como código Python válido. Por ejemplo, si usas un apóstrofe entre comillas simples, producirás un error. Esto ocurre porque Python interpreta todo lo que hay entre la primera comilla simple y el apóstrofo como una cadena. Luego intenta interpretar el resto del texto como código Python, lo que provoca errores.

He aquí cómo utilizar correctamente las comillas simples y dobles. Guarda este programa como *apostrophe.py* y ejecútalo:

apóstrofe.py

```
message = "Uno de los puntos fuertes de Python es su comunidad
diversa" print(message)
```

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

```
Uno de los puntos fuertes de Python es la diversidad de su comunidad.
```

Sin embargo, si utilizas comillas simples, Python no puede identificar dónde debe terminar la cadena:

```
message = 'Uno de los puntos fuertes de Python es su comunidad
diversa' print(message)
```

Verá el siguiente resultado:

```
  File "apostrophe.py", line 1
    message = 'Uno de los puntos fuertes de Python es su comunidad diversa'.
                                ^(1)
SyntaxError: sintaxis inválida
```

En la salida puedes ver que el error se produce en (1) justo después de la segunda comilla simple. Este *error de sintaxis* indica que el intérprete no reconoce algo en el código como código Python válido. Los errores pueden provenir de una variedad de fuentes, y señalaré algunas comunes a medida que surjan. Es posible que veas errores de sintaxis a menudo mientras aprendes a escribir código Python correcto. Los errores de sintaxis son también el tipo de error menos específico, por lo que pueden ser difíciles y frustrantes de identificar y corregir. Si te quedas atascado en un error particularmente estúpido, mira las sugerencias en el Apéndice C.

NOTE *La función de resaltado de sintaxis de tu editor debería ayudarte a detectar rápidamente algunos errores de sintaxis mientras escribes tus programas. Si ves código Python resaltado como si fuera inglés o inglés resaltado como si fuera código Python, probablemente tengas una comilla mal emparejada en algún lugar de tu archivo.*

Impresión en Python 2

La sentencia `print` tiene una sintaxis ligeramente diferente en Python 2:

```
>>> python2.7
>>> print "¡Hola mundo Python 2.7!"
¡Hola mundo Python 2.7!
```

En Python 2 no se necesitan paréntesis alrededor de la frase que quieres imprimir. Técnicamente, `print` es una función en Python 3, por lo que necesita paréntesis. Algunas sentencias `print` de Python 2 sí incluyen paréntesis, pero el comportamiento puede ser un poco diferente de lo que verás en Python 3. Básicamente, cuando veas código escrito en Python 2..,
es de esperar que aparezcan algunas sentencias `print` con paréntesis y otras sin ellos.

TRY IT yOURSELF

Guarda cada uno de los siguientes ejercicios como un archivo separado con un nombre como *nombre_casos.py*. Si te atascas, tómate un descanso o consulta las sugerencias del Apéndice C.

2-3. Mensaje personal: Almacene el nombre de una persona en una variable e imprima un mensaje para esa persona. Tu mensaje debe ser simple, como "Hola Eric, ¿te gustaría aprender algo de Python hoy?".

2-4. Casos de nombre: Almacena el nombre de una persona en una variable y luego imprime el nombre de esa persona en minúsculas, mayúsculas y mayúsculas de título.

2-5. Cita célebre: Busca una cita de una persona famosa a la que admires. Imprime la cita y el nombre de su autor. El resultado debe ser similar al siguiente, incluidas las comillas:

> *Albert Einstein dijo una vez: "Una persona que nunca cometió un error nunca intentó nada nuevo".*

2-6. Cita célebre 2: Repite el Ejercicio 2-5, pero esta vez guarda el nombre de la persona famosa en una variable llamada `persona_famosa`. Luego redacta tu mensaje y guárdalo en una nueva variable llamada `mensaje`. Imprime tu mensaje.

2-7. Eliminación de nombres: Guarde el nombre de una persona e incluya algunos caracteres de espacio en blanco al principio y al final del nombre. Asegúrate de utilizar cada combinación de caracteres, `"\t"` y `"\n"`, al menos una vez.

Imprima el nombre una vez, de forma que se muestren los espacios en blanco alrededor del nombre. A continuación, imprima el nombre utilizando cada una de las tres funciones de eliminación, `lstrip()`, `rstrip()` y `strip()`.

Números

Los números se utilizan muy a menudo en programación para llevar la puntuación en los juegos, representar datos en visualizaciones, almacenar información en aplicaciones web, etcétera. Python trata los números de diferentes maneras, dependiendo de cómo se utilicen. Veamos primero cómo Python maneja los números enteros, porque son los más sencillos con los que trabajar.

Entero

En Python puedes sumar (+), restar (-), multiplicar (*) y dividir (/) números enteros.

```
>>> 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.

Python utiliza dos símbolos de multiplicación para representar exponentes:

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

Python también soporta el orden de las operaciones, por lo que puedes usar múltiples operaciones en una expresión. También puedes usar paréntesis para modificar el orden de las operaciones, de forma que Python pueda evaluar tu expresión en el orden que especifiques. Por ejemplo:

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

El espaciado en estos ejemplos no tiene ningún efecto sobre cómo Python evalúa las expresiones; simplemente te ayuda a identificar más rápidamente las operaciones que tienen prioridad cuando lees el código.

Flotadores

Python llama *float* a cualquier número con punto decimal. Este término se

utiliza en la mayoría de los lenguajes de programación, y se refiere al hecho de que un punto decimal puede aparecer en cualquier posición de un número. Todo lenguaje de programación debe

diseñarse cuidadosamente para gestionar correctamente los números decimales, de modo que los números se comporten adecuadamente independientemente de dónde aparezca el punto decimal.

En su mayor parte, puede utilizar decimales sin preocuparse de cómo se comportan. Simplemente introduce los números que quieras usar y Python hará lo que esperas:

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

Pero ten en cuenta que a veces puedes obtener un número arbitrario de deci- malios en tu respuesta:

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004
```

Esto ocurre en todos los lenguajes y no tiene mayor importancia. Python intenta encontrar una forma de representar el resultado de la forma más precisa posible, lo que a veces es difícil dado cómo los ordenadores tienen que representar los números internamente. Ignora los decimales extra por ahora; aprenderás formas de tratar con los decimales extra cuando lo necesites en los proyectos de la Parte II.

Evitar errores tipográficos con la función str()

A menudo, querrás utilizar el valor de una variable dentro de un mensaje. Por ejemplo, digamos que quieres desearle a alguien un feliz cumpleaños. Puede escribir el código
así:

cumpleaños.py
```
edad = 23
message = "¡Feliz " + edad + "rd Cumpleaños!"

print(mensaje)
```

Podrías esperar que este código imprimiera la simple felicitación de cumpleaños, `¡Feliz 23 cumpleaños!` Pero si ejecutas este código, verás que genera un error:

```
Traceback (última llamada más reciente):
  File "cumpleaños.py", line 2, in <module>
    message = "¡Feliz " + edad + "rd
    Cumpleaños!"
❶ TypeError: No se puede convertir el objeto 'int' a str implícitamente
```

Esto es un *error de tipo*. Significa que Python no puede reconocer el tipo de información que estás usando. En este ejemplo Python ve en ❶ que estás usando una variable que tiene un valor entero (`int`), pero no está seguro de cómo interpretarlo.

valor. Python sabe que la variable puede representar tanto el valor numérico 23 como los caracteres *2* y *3*. Cuando usas enteros dentro de cadenas como esta, necesitas especificar explícitamente que quieres que Python use el entero como una cadena de caracteres. Puede hacerlo envolviendo la variable en la función `str()`, que le dice a Python que represente los valores que no son cadenas como cadenas:

```
edad = 23 años
message = "¡Feliz " + str(edad) + "rd Cumpleaños!"

print(mensaje)
```

Python sabe ahora que quieres convertir el valor numérico 23 en una cadena y mostrar los caracteres `2` y `3` como parte del mensaje de cumpleaños. Ahora obtienes el mensaje que esperabas, sin ningún error:

```
¡Feliz 23 cumpleaños!
```

Trabajar con números en Python es sencillo la mayor parte del tiempo. Si obtienes resultados inesperados, comprueba si Python está interpretando tus números de la forma que deseas, ya sea como un valor numérico o como un valor de cadena.

Números enteros en Python 2

Python 2 devuelve un resultado ligeramente diferente cuando se dividen dos enteros:

```
>>> python2.7
>>> 3 / 2
1
```

En lugar de `1,5`, Python devuelve `1`. La división de enteros en Python 2 da como resultado un entero con el resto truncado. Tenga en cuenta que el resultado no es un entero redondeado; el resto simplemente se omite.

Para evitar este comportamiento en Python 2, asegúrese de que al menos uno de los números es un float. Al hacerlo, el resultado también será un float:

```
>>> 3 / 2
1
>>> 3.0 / 2
1.5
>>> 3 / 2.0
1.5
>>> 3.0 / 2.0
1.5
```

Este comportamiento de división es una fuente común de confusión cuando la gente que está acostumbrada a Python 3 empieza a usar Python 2, o viceversa. Si usas o creas código que mezcla enteros y flotantes, ten cuidado con el comportamiento irregular.

TRY IT yOURSELF

2-8. Número Ocho: Escribe operaciones de suma, resta, multiplicación y división que den como resultado el número 8. Asegúrate de encerrar tus operaciones en sentencias `print` para ver los resultados. Debes crear cuatro líneas que parezcan así:

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

Su salida debería ser simplemente cuatro líneas con el número 8 apareciendo una vez en cada línea.

2-9. Número favorito: Guarda tu número favorito en una variable. Luego, usando esa variable, crea un mensaje que revele tu número favorito. Imprime ese mensaje.

Comentarios

Los comentarios son una característica extremadamente útil en la mayoría de los lenguajes de programación. Todo lo que has escrito en tus programas hasta ahora es código Python. A medida que tus programas se hacen más largos y complicados, debes añadir notas dentro de tus programas que describan tu enfoque general del problema que estás resolviendo. Un *comentario te permite* escribir notas en inglés dentro de tus programas.

¿Cómo se escriben los comentarios?

En Python, la almohadilla (#) indica un comentario. El intérprete de Python ignora todo lo que sigue a una almohadilla. Por ejemplo:

comentar.py

```
# Saluda a todos.
print("¡Hola gente de Python!")
```

Python ignora la primera línea y ejecuta la segunda.

```
¡Hola gente de Python!
```

¿Qué tipo de comentarios debe escribir?

La principal razón para escribir comentarios es explicar qué se supone que hace el código y cómo se hace que funcione. Cuando estás trabajando en un proyecto, entiendes cómo encajan todas las piezas. Pero cuando vuelvas al proyecto después de un tiempo, es probable que hayas olvidado algunos detalles. Siempre puedes estudiar tu código durante un rato y averiguar cómo se suponía que debían funcionar los segmentos, pero escribir buenos comentarios puede ahorrarte tiempo al resumir tu enfoque general en un inglés claro.

Si quieres convertirte en un programador profesional o colaborar con otros programadores, debes escribir comentarios con sentido. Hoy en día, la mayoría del software se escribe en colaboración, 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 ver comentarios en el código, así que lo mejor es que empieces ya a añadir comentarios descriptivos a tus programas. Escribir comentarios claros y concisos en tu código es uno de los hábitos más beneficiosos que puedes adquirir como nuevo programador.

A la hora de decidir si escribir o no un comentario, pregúntate si has tenido que considerar varios enfoques antes de dar con una forma razonable de hacer que algo funcione; si es así, escribe un comentario sobre tu solución. Es mucho más fácil borrar comentarios extra más tarde que volver atrás y escribir comentarios para un programa escasamente comentado. A partir de ahora, usaré comentarios en ejemplos a lo largo de este libro para ayudar a explicar secciones de código.

TRY IT yOURSELF

2-10. Añadir comentarios: Elija dos de los programas que ha escrito y añada al menos un comentario a cada uno. Si no tienes nada específico que escribir porque tus programas son demasiado simples en este punto, simplemente añade tu nombre y la fecha actual en la parte superior de cada archivo de programa. A continuación, escribe una frase que describa lo que hace el programa.

El zen de Python

Durante mucho tiempo, el lenguaje de programación Perl fue el pilar de Internet. La mayoría de los sitios web interactivos de los primeros tiempos funcionaban con scripts Perl. El lema de la comunidad Perl en aquella época era: "Hay más de una forma de hacerlo". A la gente le gustó esta mentalidad durante un tiempo, porque la flexibilidad escrita en el lenguaje hacía posible resolver la mayoría de los problemas de varias maneras. Este enfoque era aceptable mientras trabajabas en tu
Pero, con el tiempo, la gente se dio cuenta de que el énfasis en la flexibilidad dificultaba el mantenimiento de grandes proyectos durante largos periodos de tiempo. Era difícil, tedioso y llevaba mucho tiempo revisar el código e intentar averiguar qué estaba pensando otra persona cuando resolvía un problema complejo.

Los programadores experimentados de Python te animarán a evitar la complejidad y a buscar la simplicidad siempre que sea posible. La filosofía de la comunidad Python está contenida en "The Zen of Python" de Tim Peters. Puedes acceder a este breve conjunto de principios para escribir buen código Python introduciendo `import this` en tu intérprete. No voy a reproducir todo

el "Zen of

Python" aquí, pero compartiré algunas líneas para ayudarte a entender por qué deberían ser importantes para ti como programador principiante de Python.

```
>>> import this
El zen de Python, por Tim Peters

Lo bonito es mejor que lo feo.
```

Los programadores de Python abrazan la idea de que el código puede ser bello y elegante. En programación, la gente resuelve problemas. Los programadores siempre han respetado las soluciones bien diseñadas, eficientes e incluso bellas. A medida que aprendas más sobre Python y lo uses para escribir más código, puede que algún día alguien mire por encima de tu hombro y diga: "¡Vaya, qué código más bonito!".

```
Lo simple es mejor que lo complejo.
```

Si puedes elegir entre una solución simple y una compleja, y ambas funcionan, utiliza la solución simple. Tu código será más fácil de mantener, y será más fácil para ti y para otros construir sobre ese código más adelante.

```
Lo complejo es mejor que lo complicado.
```

La vida real es complicada y a veces no es posible encontrar una solución sencilla a un problema. En ese caso, utiliza la solución más sencilla que funcione.

```
La legibilidad cuenta.
```

Aunque tu código sea complejo, procura que sea legible. Cuando trabajes en un proyecto que implique una codificación compleja, céntrate en escribir comentarios informativos para ese código.

```
Debería haber una -y preferiblemente sólo una- forma obvia de hacerlo.
```

Si se pide a dos programadores de Python que resuelvan el mismo problema, deberían encontrar soluciones bastante compatibles. Esto no quiere decir que no haya lugar para la creatividad en la programación. Al contrario. Pero gran parte de la programación consiste en utilizar pequeños enfoques comunes a situaciones simples dentro de un proyecto más grande y creativo. Las tuercas y tornillos de tus programas deben tener sentido para otros programadores de Python.

```
Ahora es mejor que nunca.
```

Podrías pasarte el resto de tu vida aprendiendo todos los entresijos de Python y de la programación en general, pero entonces nunca completarías ningún proyecto. No intentes escribir código perfecto; escribe código que funcione, y luego decide si mejoras tu código para ese proyecto o pasas a

algo nuevo.

Cuando pases al siguiente capítulo y empieces a profundizar en temas más complicados, intenta mantener en mente esta filosofía de simplicidad y claridad. Los programadores experimentados respetarán más tu código y estarán encantados de darte su opinión y colaborar contigo en proyectos interesantes.

TRY IT yOURSELF

2-11. Zen de Python: Introduzca **import this** en una sesión terminal de Python y hojee los principios adicionales.

resumen

En este capítulo aprendiste a trabajar con variables. Aprendiste a usar nombres de variables descriptivos y cómo resolver errores de nombre y errores de sintaxis cuando surgen. Aprendiste qué son las cadenas y cómo mostrarlas usando minúsculas, mayúsculas y mayúsculas de título. Empezó a utilizar los espacios en blanco para organizar la salida de forma ordenada y aprendió a eliminar los espacios en blanco innecesarios de las distintas partes de una cadena. Empezaste a trabajar con números enteros y flotantes, y has leído acerca de algunos comportamientos inesperados a tener en cuenta cuando se trabaja con datos numéricos. También aprendiste a escribir comentarios explicativos para que tu código sea más fácil de leer para ti y para los demás. Por último, has leído sobre la filosofía de mantener el código lo más simple posible, siempre que sea posible.

En el Capítulo 3 aprenderás a almacenar colecciones de información en variables llamadas *listas*. Aprenderás a trabajar a través de una lista, manipulando cualquier información en esa lista.

3

ENTRODUCIENDOLOSISTES



En este capítulo y en el siguiente aprenderás qué son las listas y cómo empezar a trabajar con los elementos de una lista. Las listas permiten almacenar conjuntos de información en un solo lugar, tanto si tienes unos pocos elementos como millones de ellos. Las listas son una de las características más potentes de Python, fácilmente accesibles para los nuevos programadores, y unen muchos conceptos importantes en la programación.

¿Qué es una lista?

Una *lista* es una colección de elementos en un orden determinado. Puedes hacer una lista que incluya las letras del alfabeto, los dígitos del 0 al 9 o los nombres de todas las personas de tu familia. Puedes poner lo que

quieras en una lista, y

los elementos de tu lista no tienen por qué estar relacionados de ninguna manera en particular. Como una lista suele contener más de un elemento, es buena idea que el nombre de tu lista sea plural, como `letras`, `dígitos` o `nombres`.

En Python, los corchetes (`[]`) indican una lista, y los elementos individuales de la lista están separados por comas. He aquí un ejemplo sencillo de una lista que contiene algunos tipos de bicicletas:

bicicletas.py

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

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

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

Dado que esta no es la salida que quieres que vean tus usuarios, vamos a aprender cómo acceder a los elementos individuales de una lista.

Acceso a los elementos de una lista

Las listas son colecciones ordenadas, por lo que se puede acceder a cualquier elemento de una lista indicando a Python la posición, o *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, saquemos la primera bicicleta de la lista `bicicletas`:

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

La sintaxis para ello se muestra en ❶. Cuando pedimos un único elemento de una lista, Python devuelve sólo ese elemento sin corchetes ni comillas:

```
senderismo
```

Este es el resultado que desea que vean sus usuarios: un resultado limpio y con un formato ordenado.

También puedes utilizar los métodos de cadena del capítulo 2 en cualquier elemento de una lista. Por ejemplo, puede dar un formato más ordenado al elemento `'trek'` utilizando el método `title()`:

```
bicicletas = ['trek', 'cannondale', 'redline', 'specialized']
print(bicicletas[0].title())
```

Este ejemplo produce el mismo resultado que el anterior, salvo que `"Trek"` se escribe con mayúsculas.

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 posición 1. Esto es así en la mayoría de los lenguajes de programación. Esto es cierto en la mayoría de los lenguajes de programación, y la razón tiene que ver con cómo se implementan las operaciones de lista a un nivel inferior. Si estás recibiendo resultados inesperados, determina si estás cometiendo un simple error off-by-one.

El segundo elemento de una lista tiene un índice de 1. Utilizando este sencillo sistema de recuento, puede obtener cualquier elemento que desee de una lista restando uno a su posición en la lista. Por ejemplo, para acceder al cuarto elemento de una lista, se solicita el elemento situado en el índice 3.

A continuación se pregunta por las bicicletas de índice `1` e índice `3`:

```
bicicletas = ['trek', 'cannondale', 'redline', 'specialized']
print(bicicletas[1])
print(bicicletas[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. Al preguntar por el elemento en el índice `-1`, Python siempre devuelve el último elemento de la lista:

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

Este código devuelve el valor `'specialized'`. Esta sintaxis es bastante útil, porque a menudo querrá acceder a los últimos elementos de una lista sin saber exactamente cuán larga es la lista. Esta convención se extiende también a otros valores de índice negativos. El índice `-2` devuelve el segundo elemento desde el final de la lista, el índice -3 devuelve el tercer elemento desde el final, y así sucesivamente.

Utilización de valores individuales de una lista

Puede utilizar valores individuales de una lista como lo haría con cualquier otra variable. Por ejemplo, puede utilizar la concatenación para crear un mensaje basado en un valor de una lista.

Intentemos extraer la primera bicicleta de la lista y redactar un mensaje utilizando ese valor.

```
  bicicletas = ['trek', 'cannondale', 'redline', 'specialized']
❶ message = "Mi primera bicicleta fue una " + bicycles[0].title() + "."

  print(mensaje)
```

En ❶, construimos una sentencia usando el valor de `bicicletas[0]` y la almacenamos en la variable `mensaje`. La salida es una sentencia simple sobre la primera bicicleta de la lista:

```
Mi primera bicicleta fue una Trek.
```

TRY IT yOURSELF

Prueba estos programas cortos para obtener experiencia de primera mano con las listas de Python. Es posible que desees crear una nueva carpeta para los ejercicios de cada capítulo para mantenerlos organizados.

3-1. Nombres: Almacena los nombres de algunos de tus amigos en una lista llamada `nombres`. Imprime el nombre de cada persona accediendo a cada elemento de la lista, de uno en uno.

3-2. Saludos: Comience con la lista que utilizó en el Ejercicio 3-1, pero en lugar de imprimir sólo el nombre de cada persona, imprima un mensaje para ellos. El texto de cada mensaje debe ser el mismo, pero cada mensaje debe estar personalizado con el nombre de la persona.

3-3. Tu propia lista: Piensa en tu medio de transporte favorito, como una motocicleta o un coche, y haz una lista que almacene varios ejemplos. Utiliza tu lista para imprimir una serie de afirmaciones sobre estos elementos, como "Me gustaría tener una motocicleta Honda".

Modificar, añadir y eliminar elementos

La mayoría de las listas que crees serán dinámicas, lo que significa que construirás una lista y luego añadirás y quitarás elementos de ella a medida que tu programa siga su curso. Por ejemplo, puedes crear un juego en el que un jugador tenga que disparar a unos alienígenas desde el cielo. Podrías almacenar el conjunto inicial de alienígenas en una lista y luego eliminar un alienígena de la lista cada vez que se derriba uno. Cada vez que aparece un nuevo alienígena en la pantalla, lo añades a la lista. Tu lista de alienígenas disminuirá y aumentará en longitud a lo largo del juego.

Modificación de elementos de una lista

La sintaxis para modificar un elemento es similar a la sintaxis para acceder a un elemento de una lista. Para modificar un elemento, utilice el nombre de la lista seguido del índice del elemento que desea modificar y, a continuación, proporcione el nuevo valor que desea que tenga ese elemento.

Por ejemplo, supongamos que tenemos una lista de motocicletas, y el primer elemento de la lista es `'honda'`. ¿Cómo cambiaríamos el valor de este primer elemento?

motocicletas.py

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

❷ motocicletas[0] = 'ducati'
print(motocicletas)
```

El código de ❶ define la lista original, con `'honda'` como primer elemento. El código de ❷ cambia el valor del primer elemento a `'ducati'`. La salida muestra que efectivamente se ha cambiado el primer elemento, y el resto de la lista permanece igual:

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

Puede cambiar el valor de cualquier elemento de una lista, no sólo del primero.

Añadir elementos a una lista

Puedes querer añadir un nuevo elemento a una lista por muchas razones. Por ejemplo, puedes querer hacer que aparezcan nuevos aliens en un juego, añadir nuevos datos a una visualización, o añadir nuevos usuarios registrados a un sitio web que has construido. Python proporciona varias formas de añadir nuevos datos a listas existentes.

Añadir elementos al final de una lista

La forma más sencilla de añadir un nuevo elemento a una lista es añadirlo a la lista. Al añadir un elemento a una lista, el nuevo elemento se añade al final de la lista. Usando la misma lista que teníamos en el ejemplo anterior, añadiremos el nuevo elemento `'ducati'` al final de la lista:

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

❶ motorcycles.append('ducati')
print(motorcycles)
```

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

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

El método `append()` facilita la construcción dinámica de listas. Por ejemplo, puede empezar con una lista vacía y luego añadir elementos a la lista utilizando una serie de sentencias `append()`. Utilizando una `lista` vacía, añadamos los elementos `"honda"`, `"yamaha"` y `"suzuki"` a la lista:

```
motocicletas = []

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

print(motocicletas)
```

La lista resultante tiene exactamente el mismo aspecto que las listas de los ejemplos anteriores:

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

Construir listas de esta manera es muy común, porque a menudo no sabrás los datos que tus usuarios quieren almacenar en un programa hasta después de que el programa se esté ejecutando. Para que tus usuarios tengan el control, empieza por definir una lista vacía que contendrá los valores de los usuarios. A continuación, añada cada nuevo valor proporcionado a la lista que acaba de crear.

Insertar elementos en una lista

Puedes añadir un nuevo elemento en cualquier posición de tu lista utilizando el método `insert()`. Esto se hace especificando el índice del nuevo elemento y el valor del nuevo elemento.

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

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

En este ejemplo, el código en ❶ inserta el valor `'ducati'` al principio de la lista. El método `insert()` abre un espacio en la posición `0` y almacena el valor `"ducati"` en ese lugar. Esta operación desplaza todos los demás valores de la lista una posición a la derecha:

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

Eliminar elementos de una lista

A menudo, querrá eliminar un elemento o un conjunto de elementos de una lista. Por ejemplo, cuando un jugador derriba a un alienígena del cielo, lo más probable es que quiera eliminarlo de la lista de alienígenas activos. O cuando un usuario

decide cancelar su cuenta en una aplicación web creada por ti, querrás eliminar a ese usuario de la lista de usuarios activos. Puedes eliminar un elemento según su posición en la lista o según su valor.

Eliminar un elemento mediante la sentencia del

Si conoce la posición del elemento que desea eliminar de una lista, puede utilizar la sentencia `del`.

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

            ❶ del
   motocicletas[0]
print(motocicletas
                )
```

El código de ❶ utiliza `del` para eliminar el primer elemento, `'honda'`, de la lista de motocicletas:

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

Puedes eliminar un elemento de cualquier posición de una lista utilizando el estado `del` si conoces su índice. Por ejemplo, así se elimina el segundo elemento, `'yamaha'`, de la lista:

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

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

La segunda motocicleta se elimina de la lista:

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

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

Eliminar un elemento con el método pop()

A veces querrás utilizar el valor de un elemento después de eliminarlo de una lista. Por ejemplo, podrías querer obtener la posición *x* e *y* de un alienígena que acaba de ser derribado, para poder dibujar una explosión en esa posición. En una aplicación web, es posible que desee eliminar un usuario de una lista de miembros activos y, a continuación, añadir ese usuario a una lista de miembros inactivos.

El método `pop()` elimina el último elemento de una lista, pero te permite trabajar con ese elemento después de eliminarlo. El término "*pop*" viene de pensar en una lista como una pila de elementos y de quitar

un elemento de la parte superior de la pila. En esta analogía, la parte superior de una pila corresponde al final de una lista.

Pongamos una moto de la lista de motos:

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

❷ motocicleta_popped = motocicletas.pop()
❸ print(motocicletas)
❹ print(moto_aparecida)
```

Comenzamos definiendo e imprimiendo la lista `motocicletas` en ❶. En ❷ extraemos un valor de la lista y lo almacenamos en la variable motocicleta `extraída`. Imprimimos la lista en ❸ para mostrar que se ha eliminado un valor de la lista. Luego imprimimos el valor extraído en ❹ para demostrar que todavía tenemos acceso al valor que fue eliminado.

La salida muestra que el valor `'suzuki'` fue eliminado del final de la lista y ahora se almacena en la variable `popped_motorcycle`:

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

¿Qué utilidad puede tener este método `pop()`? Imagina que las motos de la lista están almacenadas en orden cronológico según la fecha en que las tuvimos. Si este es el caso, podemos utilizar el método `pop()` para imprimir una declaración sobre la última motocicleta que compramos:

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

last_owned = motorcycles.pop()
print("La última moto que tuve fue una " + last_owned.title() + ".")
```

La salida es una simple frase sobre la última motocicleta que hemos tenido:

```
La última moto que tuve fue una Suzuki.
```

Retirar elementos de cualquier posición de una lista

Puede utilizar `pop()` para eliminar un elemento de una lista en cualquier posición incluyendo entre paréntesis el índice del elemento que desea eliminar.

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

❶ primer_propietario = motocicletas.pop(0)
❷ print('La primera moto que tuve fue una ' + first_owned.title() + '.')
```

Empezamos haciendo saltar la primera moto de la lista en ❶, y luego imprimimos un mensaje sobre esa moto en ❷. La salida es una simple frase que describe la primera motocicleta que tuve:

```
La primera moto que tuve fue una Honda.
```

Recuerde que cada vez que utiliza `pop()`, el elemento con el que trabaja deja de estar almacenado en la lista.

Si no estás seguro de si utilizar la sentencia `del` o el método `pop()`, aquí tienes una forma sencilla de decidirte: cuando quieras eliminar un elemento de una lista y no utilizarlo de ninguna forma, utiliza la sentencia `del`; si quieres utilizar un elemento mientras lo eliminas, utiliza el método `pop()`.

Eliminar un elemento por valor

A veces no conocerás la posición del valor que quieres eliminar de una lista. Si sólo conoce el valor del elemento que desea eliminar, puede utilizar el método `remove()`.

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

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

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

El código de ❶ le dice a Python que averigüe dónde aparece `'ducati'` en la lista y elimine ese elemento:

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

También puedes usar el método `remove()` para trabajar con un valor que está siendo eliminado de una lista. Vamos a eliminar el valor `'ducati'` e imprimir una razón para eliminarlo de la lista:

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

❷ too_expensive = 'ducati'
❸ motocicletas.eliminar(demasiado_caras)
print(motocicletas)
❹ print("\nA " + too_expensive.title() + " es demasiado caro para mí.")
```

Tras definir la lista en ❶, almacenamos el valor `'ducati'` en una variable llamada `too_expensive` ❷. A continuación, utilizamos esta variable para decirle a Python qué valor

para eliminarlo de la lista en ❸. En ❹ el valor `'ducati'` se ha eliminado de la lista, pero sigue almacenado en la variable `too_expensive`, lo que nos permite imprimir una declaración sobre por qué hemos eliminado `'ducati'` de la lista de motocicletas:

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

Una Ducati es demasiado cara para mí.
```

NOTE *El método remove() elimina sólo la primera aparición del valor especificado. Si existe la posibilidad de que el valor aparezca más de una vez en la lista, deberá utilizar un bucle para determinar si se han eliminado todas las apariciones del valor. Aprenderás cómo hacerlo en el Capítulo 7.*

TRY IT yOURSELF

Los siguientes ejercicios son un poco más complejos que los del capítulo 2, pero te dan la oportunidad de utilizar las listas de todas las formas descritas.

3-4. Lista de invitados: Si pudieras invitar a cenar a cualquier persona, viva o fallecida, ¿a quién invitarías? Haz una lista con al menos tres personas a las que te gustaría invitar a cenar. Luego, utiliza tu lista para imprimir un mensaje dirigido a cada persona, invitándola a cenar.

3-5. Cambio de lista de invitados: Acabas de enterarte de que uno de tus invitados no puede asistir a la cena, así que tienes que enviar un nuevo juego de invitaciones. Tendrás que pensar en otra persona a la que invitar.

- Comience con su programa del Ejercicio 3-4. Añade una sentencia `print` al final de tu programa indicando el nombre del invitado que no puede venir.
- Modifica tu lista, sustituyendo el nombre del invitado que no puede venir por el de la nueva persona a la que invitas.
- Imprima un segundo juego de mensajes de invitación, uno para cada persona que siga en su lista.

3-6. Más invitados: Acabas de encontrar una mesa más grande para cenar, así que ahora hay más espacio disponible. Piensa en tres invitados más a los que invitar a cenar.

- Empieza con tu programa del Ejercicio 3-4 o Ejercicio 3-5. Añade una sentencia `print` al final de tu programa informando de que has encontrado una mesa más grande.
- Utilice `insert()` para añadir un nuevo invitado al principio de la lista.
- Utilice `insert()` para añadir un nuevo invitado al centro de la lista.
- Utilice `append()` para añadir un nuevo invitado al final de la lista.

- Imprima un nuevo juego de mensajes de invitación, uno para cada persona de su lista.

3-7. Lista de invitados menguante: Acabas de enterarte de que tu nueva mesa no llegará a tiempo para la cena y sólo tienes espacio para dos invitados.

- Comience con su programa del Ejercicio 3-6. Añada una nueva línea que imprima un mensaje diciendo que sólo puede invitar a cenar a dos personas.
- Utilice `pop()` para eliminar invitados de la lista de uno en uno hasta que sólo queden dos nombres en la lista. Cada vez que elimine un nombre de la lista, imprima un mensaje para esa persona haciéndole saber que lamenta no poder invitarle a cenar.
- Imprime un mensaje para cada una de las dos personas que siguen en tu lista, haciéndoles saber que siguen invitadas.
- Utiliza `del` para eliminar los dos últimos nombres de la lista y obtener una lista vacía. Imprima su lista para asegurarse de que realmente tiene una lista vacía al final de su programa.

Organizar una lista

A menudo, tus listas se crearán en un orden impredecible, porque no siempre puedes controlar el orden en que tus usuarios proporcionan sus datos. Aunque esto es inevitable en la mayoría de los casos, a menudo querrás presentar la información en un orden determinado. A veces querrás conservar el orden original de tu lista, y otras querrás cambiarlo. Python ofrece diferentes formas de organizar tus listas, dependiendo de la situación.

Ordenación permanente de una lista con el método sort()

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

coches.py

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

coches.orde
nar()
print(coche
s)
```

❶

El método `sort()`, mostrado en ❶, cambia el orden de la lista de forma perma- nente. Los coches están ahora en orden alfabético, y nunca podremos volver al orden original:

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

También 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:

```
coches = ['bmw', 'audi', 'toyota', 'subaru']
coches.ordenar(inverso=Verdadero)
print(coches)
```

De nuevo, el orden de la lista cambia permanentemente:

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

Ordenación temporal de una lista con la función sorted()

Para mantener el orden original de una lista pero presentarla ordenada, puede utilizar la función `sorted()`. La función `sorted()` permite mostrar la lista en un orden determinado, pero no afecta al orden real de la lista.

Probemos esta función con la lista de coches.

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

❶ print("Aquí está la lista
  original:") print(coches)

❷ print("\nAquí está la lista ordenada:")
  print(sorted(cars))

❸ print("\nAquí está de nuevo la lista
  original:") print(coches)
```

Primero imprimimos la lista en su orden original en ❶ y luego en orden alfabético en ❷. Después de mostrar la lista en el nuevo orden, demostramos que la lista sigue almacenada en su orden original en ❸.

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

Aquí está la lista ordenada:
['audi', 'bmw', 'subaru', 'toyota']

❹ Aquí está la lista original de
  nuevo: ['bmw', 'audi', 'toyota',
  'subaru']
```

Observe que la lista sigue existiendo en su orden original en ❹ después de utilizar la función sorted(). La función `sorted()` también puede aceptar un argumento `reverse=True` si desea mostrar una lista en orden alfabético inverso.

NOTE *Ordenar una lista alfabéticamente es un poco más complicado cuando todos los valores no están en minúsculas. Hay varias formas de interpretar las mayúsculas cuando se decide un orden de clasificación, y especificar el orden exacto puede ser más complejo de lo que queremos tratar en este momento. Sin embargo, la mayoría de las aproximaciones a la ordenación se basarán directamente en lo aprendido en esta sección.*

Imprimir una lista en orden inverso

Para invertir el orden original de una lista, puede utilizar el método `reverse()`.
Si originalmente almacenamos la lista de coches en orden cronológico según la fecha de propiedad, podemos reordenarla fácilmente en orden cronológico inverso:

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

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

Observe que `reverse()` no ordena alfabéticamente hacia atrás, 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 permanentemente, pero puede volver al orden original en cualquier momento aplicando `reverse()` a la misma lista por segunda vez.

Determinar la longitud de una lista

Puede averiguar rápidamente la longitud de una lista utilizando la función `len()`. La lista de este ejemplo tiene cuatro elementos, por lo que su longitud es `4`:

```
>>> coches = ['bmw', 'audi', 'toyota', 'subaru']
>>> len(coches)
4
```

Encontrará `len()` útil cuando necesite identificar el número de alienígenas que aún quedan por derribar en un juego, determinar la cantidad de datos que tiene que gestionar en una visualización o calcular el número de usuarios registrados en un sitio web, entre otras tareas.

NOTE *Python cuenta los elementos de una lista empezando por uno, por lo que no deberías encontrarte con ningún error de uno en uno al determinar la longitud de una lista.*

TRY IT yOURSELF

3-8. Ver el mundo: Piensa en al menos cinco lugares del mundo que te gustaría visitar.

- Guarde las ubicaciones en una lista. Asegúrate de que la lista no esté en orden alfabético.
- Imprime tu lista en su orden original. No se preocupe por imprimir la lista ordenadamente, simplemente imprímala como una lista Python sin procesar.
- Utilice `sorted()` para imprimir su lista en orden alfabético sin modificar la lista real.
- Demuestre que su lista sigue en su orden original imprimiéndola.
- Utilice `sorted()` para imprimir su lista en orden alfabético inverso sin cambiar el orden de la lista original.
- Demuestra que tu lista sigue en su orden original imprimiéndola de nuevo.
- Utilice `reverse()` para cambiar el orden de la lista. Imprima la lista para mostrar que su orden ha cambiado.
- Utilice `reverse()` para volver a cambiar el orden de la lista. Imprime la lista para mostrar que ha vuelto a su orden original.
- Utilice `sort()` para cambiar su lista de modo que se almacene en orden alfabético. Imprima la lista para mostrar que se ha cambiado su orden.
- Utilice `sort()` para cambiar su lista de modo que se almacene en orden alfabético inverso. Imprima la lista para mostrar que su orden ha cambiado.

3-9. Invitados a Cenar: Trabajando con uno de los programas de los Ejercicios 3-4 a 3-7 (página 46), utilice `len()` para imprimir un mensaje indicando el número de personas que está invitando a cenar.

3-10. Cada función: Piensa en algo que puedas almacenar en una lista. Por ejemplo, podrías hacer una lista de montañas, ríos, países, ciudades, idiomas o cualquier otra cosa que te apetezca. Escribe un programa que cree una lista que contenga estos elementos y luego utilice cada función introducida en este capítulo al menos una vez.

Evitar errores de índice al trabajar con listas

Un tipo de error es habitual cuando se trabaja con listas por primera vez. Supongamos que tienes una lista con tres elementos y preguntas por el cuarto:

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

Este ejemplo da lugar a un *error de índice*:

```
Traceback (última llamada más reciente):
  File "motos.py", line 3, in <module> print(motos[3])
IndexError: índice de lista fuera de rango
```

Python intenta darle el elemento en el índice 3. Pero cuando busca en la lista, ningún elemento de motocicletas tiene el índice 3. Este error es típico. Pero cuando busca en la lista, ningún elemento en `motocicletas` tiene un índice 3. Debido a la naturaleza de la indexación en listas, este error es típico. La gente piensa que el tercer elemento es el número 3, porque empiezan a contar en 1. Pero en Python el tercer elemento es el número 2, porque empieza a indexar en 0.

Un error de índice significa que Python no puede averiguar el índice que has solicitado. Si se produce un error de índice en tu programa, intenta ajustar en uno el índice que estás pidiendo. Luego ejecuta el programa de nuevo para ver si los resultados son correctos.

Ten en cuenta que siempre que quieras acceder al último elemento de una lista debes utilizar el índice `-1`. Esto siempre funcionará, incluso si tu lista ha cambiado de tamaño desde la última vez que accediste a ella:

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

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

```
suzuki
```

La única vez que este enfoque causará un error es cuando se solicita el último elemento de una lista vacía:

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

No hay elementos en `motocicletas`, por lo que Python devuelve otro error de índice:

```
Traceback (última llamada más reciente):
  File "motos.py", line 3, in <module>
    print(motos[-1])
IndexError: índice de lista fuera de rango
```

NOTE *Si se produce un error de índice y no puede averiguar cómo resolverlo, intente imprimir su lista o simplemente imprimir la longitud de su lista. Es posible que su lista tenga un aspecto muy diferente al que usted pensaba, especialmente si ha sido gestionada dinámicamente por su programa. Ver la lista real, o el número exacto de elementos de tu lista, puede ayudarte a resolver estos errores lógicos.*

TRY IT yOURSELF

3-11. Error intencionado: Si todavía no ha recibido un error de índice en uno de sus programas, intente que se produzca uno. Cambie un índice en uno de sus programas para producir un error de índice. Asegúrese de corregir el error antes de cerrar el programa.

resumen

En este capítulo has aprendido qué son las listas y cómo trabajar con los elementos individuales de una lista. Ha aprendido a definir una lista y a añadir y eliminar elementos. Has aprendido a ordenar listas de forma permanente y temporal con fines de visualización. También aprendiste a encontrar la longitud de una lista y a evitar errores de índice cuando trabajas con listas.

En el Capítulo 4 aprenderás a trabajar con los elementos de una lista de forma más eficiente. Recorriendo en bucle cada elemento de una lista utilizando unas pocas líneas de código podrás trabajar de forma eficiente, incluso cuando tu lista contenga miles o millones de elementos.

4

COMPARTIR CON LOS LISTES



En el capítulo 3 aprendiste a hacer una lista sencilla y a trabajar con los elementos individuales de una lista. En este capítulo aprenderás a recorrer en *bucle* una lista entera.

lista utilizando sólo unas pocas líneas de código independientemente de cómo

la longitud de la lista. Los bucles permiten realizar la misma acción, o conjunto de acciones, con cada elemento de una lista. Como resultado, podrás trabajar eficazmente con listas de cualquier longitud, incluidas las que tienen miles o incluso millones de elementos.

Recorrer una lista completa

A menudo querrás recorrer todas las entradas de una lista, realizando la misma tarea con cada elemento. Por ejemplo, en un juego puedes querer mover todos los elementos de la pantalla la misma cantidad, o en una lista de números puedes querer realizar la misma operación estadística en cada

elemento. O quizá quieras mostrar cada titular de una lista de artículos de un sitio web. Cuando quieras realizar la misma acción con cada elemento de una lista, puedes utilizar el bucle `for` de Python.

Supongamos que tenemos una lista de nombres de magos y queremos imprimir cada nombre de la lista. Podríamos hacerlo recuperando cada nombre de la lista individualmente, pero este enfoque podría causar varios problemas. Por un lado, sería repetitivo hacerlo con una lista larga de nombres. Además, tendríamos que
cambiar nuestro código cada vez que cambiara la longitud de la lista. Un bucle `for` evita a m b o s problemas dejando que Python los gestione internamente.

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

magicians.py

```
❶ magicians = ['alice', 'david', 'carolina']
❷ para mago en magos:
❸     print(mago)
```

Comenzamos definiendo una lista en ❶, igual que hicimos en el capítulo 3. En ❷ definimos un bucle `for`. Esta línea le dice a Python que saque un nombre de la lista `magos`, y lo almacene en la variable `mago`. En ❸ le decimos a Python que imprima el nombre que se acaba de almacenar en `mago`. Python repite entonces las líneas ❷ y ❸, una vez por cada nombre de la lista. Podría ayudar leer este código como "Para cada mago en la lista de magos, imprime el nombre del mago." La salida es una simple impresión de cada nombre de la l i s t a :

```
alice
david
carolina
```

Una mirada más de cerca al bucle

El concepto de bucle es importante porque es una de las formas más comunes que tiene un ordenador de automatizar tareas repetitivas. Por ejemplo, en un bucle simple como el que usamos en *magicians.py*, Python lee inicialmente la primera línea del bucle:

```
de mago en los magos:
```

Esta línea le dice a Python que recupere el primer valor de la lista `magos` y lo almacene en la variable `mago`. Este primer valor es `'alice'`. A continuación, Python lee la siguiente línea:

```
    print(mago)
```

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

```
de mago en los magos:
```

Python recupera el siguiente nombre de la lista, `'david'`, y almacena ese valor en `magician`. A continuación, Python ejecuta la línea:

```
print(mago)
```

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

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

Cuando escriba sus propios bucles `for`, tenga en cuenta que puede elegir el nombre que desee para la variable temporal que contiene cada valor de la lista. Sin embargo, es útil elegir un nombre significativo que represente un único elemento de la lista. Por ejemplo, ésta es una buena forma de iniciar un bucle `for` para una lista de gatos, una lista de perros y una lista general de elementos:

```
para gato en gatos:
para perro en perros:
para elemento en lista_de_elementos:
```

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

Hacer más trabajo dentro de un bucle for

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

```
magos = ['alice', 'david', 'carolina'] for
mago in magos:
    print(mago.titulo() + ", ¡qué buen truco!")
```

La única diferencia en este código está en ❶ donde componemos un mensaje para cada mago, empezando con el nombre de ese mago. La primera vez a través del bucle el valor de mago es `'alice'`, por lo que Python comienza el primer mensaje con el nombre `'Alice'`. La segunda vez el mensaje comenzará con `'David'`, y la tercera vez el mensaje comenzará con `'Carolina'`.

El resultado muestra un mensaje personalizado para cada mago de la lista:

```
Alice, ¡qué buen truco! ¡David, eso
fue un gran truco! ¡Carolina, eso
fue un gran truco!
```

También puede escribir tantas líneas de código como desee en el bucle `for`. Cada línea sangrada que sigue a la línea `for magician en magicians` se considera *dentro del bucle*, y cada línea sangrada se ejecuta una vez por cada

de la lista. Por lo tanto, puedes hacer todo el trabajo que quieras con cada valor de la lista.

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

```
magos = ['alice', 'david', 'carolina'] for
mago in magos:
    print(mago.titulo() + ", ¡qué buen truco!")
❶   print("Estoy deseando ver tu próximo truco, " + magician.title() + ".\n")
```

Como hemos sangrado ambas sentencias de `impresión`, cada línea se ejecutará una vez por cada mago de la lista. La nueva línea (`"\n"`) en la segunda sentencia `print` ❶ inserta una línea en blanco después de cada paso por el bucle. Esto crea un conjunto de mensajes que se agrupan ordenadamente para cada persona en la lista:

```
Alice, ¡qué buen truco!
Estoy deseando ver tu próximo truco, Alice.

David, ¡qué buen truco!
Estoy deseando ver tu próximo truco, David.

Carolina, ¡qué buen truco!
Estoy deseando ver tu próximo truco, Carolina.
```

Puede utilizar tantas líneas como desee en sus bucles `for`. En la práctica, a menudo le resultará útil realizar varias operaciones diferentes con cada elemento de una lista cuando utilice un bucle `for`.

Hacer algo después de un bucle for

¿Qué sucede una vez que un bucle `for` ha terminado de ejecutarse? Normalmente, querrá resumir un bloque de salida o pasar a otro trabajo que su programa debe realizar.

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

```
magos = ['alice', 'david', 'carolina'] for
mago in magos:
    print(mago.titulo() + ", ¡qué buen truco!")
    print("Estoy deseando ver tu próximo truco, " + magician.title() + ".\n")

❶ print("Gracias a todos. ¡Ha sido un gran espectáculo de magia!")
```

Las dos primeras sentencias de `impresión se` repiten una vez por cada mago de la lista, como has visto antes. Sin embargo, como la línea de ❶ no tiene sangría, sólo se imprime una vez:

```
Alice, ¡qué buen truco!
Estoy deseando ver tu próximo truco, Alice.

David, ¡qué buen truco!
Estoy deseando ver tu próximo truco, David.

Carolina, ¡qué buen truco!
Estoy deseando ver tu próximo truco, Carolina.

Gracias a todos. Ha sido un gran espectáculo de magia.
```

Cuando proceses datos utilizando un `bucle for`, verás que es una buena forma de resumir una operación que se ha realizado en un conjunto de datos completo. Por ejemplo, puedes usar un `bucle for` para inicializar un juego recorriendo una lista de personajes y mostrando cada personaje en la pantalla. A continuación, puede escribir un bloque sin sangría después de este bucle que muestre un botón Reproducir ahora después de que todos los caracteres se hayan dibujado en la pantalla.

Evitar errores de sangría

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

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

Examinemos algunos de los errores de sangría más comunes.

Olvidar las sangrías

Siempre indenta la línea después de la sentencia `for` en un bucle. Si lo olvidas, Python te lo recordará:

magos.py

```
magos = ['alice', 'david', 'carolina'] for
mago in magos:
❶ print(mago)
```

www.allitebooks.com

La sentencia `print` en ❶ debería estar sangrada, pero no lo está. Cuando Python espera un bloque sangrado y no lo encuentra, te indica con qué línea ha tenido un problema.

```
  File "magos.py", line 3 print(mago)
        ^
IndentationError: se esperaba un bloque sangrado
```

Por lo general, puede resolver este tipo de error de indentación indentando la línea o líneas inmediatamente posteriores a la sentencia `for`.

Olvido de sangrar líneas adicionales

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

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

```
magos = ['alice', 'david', 'carolina'] for
mago in magos:
    print(mago.titulo() + ", ¡qué buen truco!")
❶ print("No puedo esperar a ver tu próximo truco, " + magician.title() + ".\n")
```

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

```
Alice, ¡qué buen truco! ¡David,
eso fue un gran truco! ¡Carolina,
eso fue un gran truco!
Estoy deseando ver tu próximo truco, Carolina.
```

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

Sangrías innecesarias

Si accidentalmente indentas una línea que no necesita ser indentada, Python te informa sobre la indentación inesperada:

hola_mundo.py

```
mensaje = "¡Hola mundo Python!"
    print(mensaje)
```

❶

No necesitamos sangrar la sentencia `print` en ❶, porque no lo hace *pertenecen* a la línea superior; por lo tanto, Python informa de ese error:

```
  File "hola_mundo.py", line 2
    print(mensaje)
    ^
IndentationError: sangría inesperada
```

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

Sangría innecesaria después del bucle

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

Por ejemplo, veamos qué ocurre si accidentalmente sangramos la línea en la que se agradece a los magos como grupo que hayan ofrecido un buen espectáculo:

```
magos = ['alice', 'david', 'carolina'] for
mago in magos:
    print(mago.titulo() + ", ¡qué buen truco!")
    print("Estoy deseando ver tu próximo truco, " + magician.title() + ".\n")

    print("¡Gracias a todos, ha sido un gran espectáculo de magia!")
```

❶

Como la línea de ❶ tiene sangría, se imprime una vez por cada persona de la lista, como puedes ver en ❷:

```
Alice, ¡qué buen truco!
Estoy deseando ver tu próximo truco, Alice.

❷ ¡Gracias a todos, ha sido un gran espectáculo de magia!
David, ¡qué buen truco!
Estoy deseando ver tu próximo truco, David.

❷ ¡Gracias a todos, ha sido un gran espectáculo de magia!
Carolina, ¡qué buen truco!
Estoy deseando ver tu próximo truco, Carolina.
```

❷ ¡Gracias a todos, ha sido un gran espectáculo de magia!

Este es otro error lógico, similar al de "Olvidar sangrar líneas adicionales" en la página 58. Como Python no sabe lo que estás intentando conseguir con tu código, ejecutará todo el código que esté escrito en sintaxis válida. Si una acción se repite muchas veces cuando debería ejecutarse sólo una vez, determina si sólo necesitas desentintar el código para esa acción.

Olvidar el colon

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

```
magos = ['alice', 'david', 'carolina']
❶ for mago in magos print(mago)
```

Si accidentalmente olvidas los dos puntos, como se muestra en ❶, obtendrás un error de sintaxis porque Python no sabe lo que estás tratando de hacer. Aunque este es un error fácil de corregir, no siempre es fácil de encontrar. Te sorprendería la cantidad de tiempo que los programadores pasan buscando errores de carácter único como éste. Estos errores son difíciles de encontrar porque a menudo sólo vemos lo que esperamos ver.

TRY IT yOURSELF

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

- Modifique su bucle `for` para imprimir una frase utilizando el nombre de la pizza en lugar de imprimir sólo el nombre de la pizza. Para cada pizza debe tener una línea de salida que contenga una declaración simple como *Me gusta la pizza de pepperoni*.
- Añade una línea al final de tu programa, fuera del bucle `for`, que diga cuánto te gusta la pizza. La salida debe consistir en tres o más líneas sobre los tipos de pizza que te gustan y luego una frase adicional, como *¡Me encanta la pizza!*

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

- Modifique su programa para imprimir una declaración sobre cada animal, como por ejemplo
  Un perro sería una gran mascota.
- Añade una línea al final de tu programa indicando qué tienen en común estos animales. Podrías imprimir una frase como *¡Cualquiera de estos animales sería una gran mascota!*

Hacer listas numéricas

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

Las listas son ideales para almacenar conjuntos de números, y Python proporciona un número de herramientas para ayudarte a trabajar eficientemente con listas de números. Una vez que entiendas cómo usar estas herramientas de forma efectiva, tu código funcionará bien incluso cuando tus listas contengan millones de elementos.

Uso de la función range()

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

números.py

```
para valor en rango(1,5):
    print(valor)
```

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

```
1
2
3
4
```

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

Para imprimir los números del 1 al 5, se utilizaría `range(1,6)`:

```
para valor en rango(1,6):
    print(valor)
```

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

```
1
2
3
4
5
```

Si la salida es diferente de lo que espera cuando se utiliza `range()`, intente ajustar su valor final en 1.

Uso de range() para hacer una lista de números

Si desea hacer una lista de números, puede convertir los resultados de range() directamente en una lista utilizando la función `list()`. Cuando se envuelve `list()` alrededor de una llamada a la función range(), la salida será una lista de números.

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

```
números = list(rango(1,6))
print(números)
```

Y éste es el resultado:

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

También podemos utilizar la función `range()` para indicar a Python que omita los números de un rango determinado. Por ejemplo, así es como listaríamos los números pares entre 1 y 10:

números_pares.py

```
números_pares = list(rango(2,11,2))
print(números_pares)
```

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

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

Puede crear casi cualquier conjunto de números que desee utilizando la función `range()`. Por ejemplo, considere cómo podría hacer una lista de los 10 primeros números cuadrados (es decir, el cuadrado de cada número entero del 1 al 10). En Python, dos asteriscos (`**`) representan exponentes. Así es como podrías poner los 10 primeros números cuadrados en una lista:

cuadrados.py

```
❶ cuadrados = []
❷ para valor en rango(1,11):
❸     cuadrado = valor**2
❹     cuadrados.append(cuadrado)

❺ print(cuadrados)
```

Empezamos con una lista vacía llamada `cuadrados` en ❶. En ❷, le decimos a Python que recorra cada valor de 1 a 10 utilizando la función `range()`. Dentro del bucle, el valor actual se eleva a la segunda potencia y se

almacena en la variable

variable `cuadrado` en ❸. En ❹, cada nuevo valor de `cuadrado` se añade a la lista de `cuadrados`. Finalmente, cuando el bucle ha terminado de ejecutarse, la lista de cuadrados se imprime en ❺:

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

> Para escribir este código de forma más concisa, omita la variable temporal `cuadrada`

y añadir cada nuevo valor directamente a la lista:

```
cuadrados = []
para valor en rango(1,11):
❶    cuadrados.append(valor**2)

print(cuadrados)
```

El código en ❶ hace el mismo trabajo que las líneas en ❸ y ❹ en *squares.py*. Cada valor del bucle se eleva a la segunda potencia y se añade inmediatamente a la lista de cuadrados.

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

Estadísticas sencillas con una lista de números

Algunas funciones de Python son específicas para listas de números. Por ejemplo, puedes encontrar fácilmente el mínimo, el máximo y la suma de una lista de números:

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

NOTE *Los ejemplos de esta sección utilizan listas cortas de números para que quepan fácilmente en la página. Funcionarían igual de bien si su lista contuviera un millón de números o más.*

Comprensión de listas

El enfoque descrito anteriormente para generar `los cuadrados de la lista` consistía en utilizar tres o cuatro líneas de código. Una comprensión de *lista permite* generar esta misma lista en una sola línea de código. Una

comprensión de lista combina el bucle `for` y la creación de nuevos elementos en una sola línea, y añade automáticamente cada nuevo elemento. Las comprensiones de lista no siempre se presentan a los principiantes, pero las he incluido aquí porque lo más probable es que las veas en cuanto empieces a mirar el código de otras personas.

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

cuadrados.py

```
cuadrados = [valor**2 para valor en
rango(1,11)] print(cuadrados)
```

Para utilizar esta sintaxis, comience con un nombre descriptivo para la lista, como `cuadrados`. A continuación, abra un conjunto de corchetes y defina la expresión para los valores que desea almacenar en la nueva lista. En este ejemplo, la expresión es `valor**2`, que eleva el valor a la segunda potencia. A continuación, escribe un bucle `for` para generar los números que desea introducir en la expresión y cierre los corchetes. El bucle `for` de este ejemplo es `for value in range(1,11)`, que introduce los valores del 1 al 10 en la expresión `value**2`. Observe que no se utilizan dos puntos al final de la sentencia `for`.

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

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

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

TRY IT yOURSELF

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

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

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

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

4-7. De tres en tres: Haz una lista de los múltiplos de 3 del 3 al 30. Utiliza un bucle `for` para imprimir los números de tu lista.

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

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

Trabajar con parte de una lista

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

Rebanar una lista

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

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

jugadores.py

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

El código de ❶ imprime un trozo de esta lista, que incluye sólo los tres primeros jugadores. La salida conserva la estructura de la lista e incluye los tres primeros jugadores de la lista:

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

Puede generar cualquier subconjunto de una lista. Por ejemplo, si desea los elementos segundo, tercero y cuarto de una lista, inicie el corte en el índice `1` y termine en el índice `4`:

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

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

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

Si omite el primer índice en una rebanada, Python inicia automáticamente su rebanada al principio de la lista:

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

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

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

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

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

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

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

Esta sintaxis permite mostrar todos los elementos desde cualquier punto de la lista hasta el final, independientemente de la longitud de la lista. Recuerde que un índice negativo devuelve un elemento a cierta distancia del final de la lista; por lo tanto, puede mostrar cualquier segmento desde el final de la lista. Por ejemplo, si queremos obtener los tres últimos jugadores de la lista, podemos utilizar el corte `jugadores[-3:]`:

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

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

Recorrer un corte en bucle

Puede utilizar un slice en un bucle `for` si desea recorrer un subconjunto de los elementos de una lista. En el siguiente ejemplo recorremos los tres primeros jugadores e imprimimos sus nombres como parte de una lista simple:

```
jugadores = ['charles', 'martina', 'michael', 'florence',

'eli'] print("Aquí están los tres primeros jugadores de mi

equipo:")
❶ para jugador en jugadores[:3]:
    print(jugador.titulo())
```

En lugar de hacer un bucle a través de toda la lista de jugadores en ❶, Python hace un bucle a través de sólo los tres primeros nombres:

```
Aquí están los tres primeros jugadores de mi equipo:
Charles
Martina
Michael
```

Las rebanadas son muy útiles en varias situaciones. Por ejemplo, al crear un juego, puedes añadir la puntuación final de un jugador a una lista cada vez

que termine de jugar. A continuación, puedes obtener las tres mejores puntuaciones de un jugador ordenando la lista en orden decreciente y tomando una porción que incluya sólo las tres primeras puntuaciones. Cuando se trabaja con datos, se pueden utilizar trozos para procesar

tus datos en trozos de un tamaño específico. O, si estás creando una aplicación web, puedes utilizar trozos para mostrar la información en una serie de páginas con una cantidad adecuada de información en cada una de ellas.

Copiar una lista

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

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

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

foods.py
```
❶ my_foods = ['pizza', 'falafel', 'tarta de zanahoria'].
❷ amigo_alimentos = mi_alimentos[:]

print("Mis comidas favoritas son:")
print(mis_comidas)

print("Las comidas favoritas de mi amigo
son:") print(amigo_comidas)
```

En ❶ hacemos una lista de los alimentos que nos gustan llamada `mis_alimentos`. En ❷ hacemos una nueva lista llamada `alimentos_amigos`. Hacemos una copia de `my_foods` pidiendo un trozo de `my_foods` sin especificar ningún índice y guardamos la copia en `friend_foods`. Cuando imprimimos cada lista, vemos que ambas contienen los mismos alimentos:

```
Mis comidas favoritas son:
['pizza', 'falafel', 'tarta de zanahoria']

Las comidas favoritas de mi amigo son:
['pizza', 'falafel', 'tarta de zanahoria']
```

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

```
mis_alimentos = ['pizza', 'falafel', 'tarta de zanahoria']
❶ amigo_alimentos = mi_alimentos[:]

❷ mi_comida.append('cannoli')
❸ friend_foods.append('helado')

print("Mis comidas favoritas son:")
```

```
print(mis_comidas)
```

```
print("Las comidas favoritas de mi amigo
son:") print(amigo_comidas)
```

En ❶ copiamos los elementos originales de `my_foods` a la nueva lista `friend_foods`, como hicimos en el ejemplo anterior. A continuación, añadimos un nuevo alimento a cada lista: en ❷ añadimos `'cannoli'` a `mi_alimentos`, y en ❸ añadimos `'helado'` a `amigo_alimentos`. A continuación, imprimimos las dos listas para ver si cada uno de estos alimentos está en la lista correspondiente.

```
Mis comidas favoritas son:
❹ ['pizza', 'falafel', 'tarta de zanahoria', 'cannoli']

Las comidas favoritas de mi amigo son:
❺ ['pizza', 'falafel', 'tarta de zanahoria', 'helado']
```

La salida en ❹ muestra que `'cannoli'` aparece ahora en nuestra lista de comidas favoritas pero `'helado'` no. En ❺ podemos ver que `'helado'` aparece ahora en la lista de nuestros amigos, pero `'cannoli'` no. Si simplemente hubiéramos establecido `friend_foods` igual a `my_foods`, no produciríamos dos listas separadas. Por ejemplo, esto es lo que ocurre cuando se intenta copiar una lista sin utilizar un corte:

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

# Esto no funciona:
❶ amigo_alimentos = mis_alimentos

mi_comida.append('cannoli')
amigo_comida.append('helado')

print("Mis comidas favoritas son:")
print(mis_comidas)

print("Las comidas favoritas de mi amigo
son:") print(amigo_comidas)
```

En lugar de almacenar una copia de `mi_comida` en `amigo_comida` en ❶, establecemos `amigo_comida` igual a `mi_comida`. Esta sintaxis en realidad le dice a Python que conecte la nueva variable `friend_foods` a la lista que ya está contenida en `my_foods`, por lo que ahora ambas variables apuntan a la misma lista. Como resultado, cuando añadimos `'cannoli'` a `mi_comida`, también aparecerá en `amigo_comida`. Del mismo modo, `"helado"` aparecerá en ambas listas, aunque parezca que sólo se ha añadido a `alimentos_amigos`.

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

```
Mis comidas favoritas son:
['pizza', 'falafel', 'tarta de zanahoria', 'cannoli', 'helado']

Las comidas favoritas de mi amigo son:
['pizza', 'falafel', 'tarta de zanahoria', 'cannoli', 'helado']
```

NOTE *No te preocupes por los detalles de este ejemplo por ahora. Básicamente, si estás tratando de trabajar con una copia de una lista y ves un comportamiento inesperado, asegúrate de que estás copiando la lista usando un slice, como hicimos en el primer ejemplo.*

TRY IT yOURSELF

4-10. Rebanadas: Usando uno de los programas que escribiste en este capítulo, agrega varias líneas al final del programa que hagan lo siguiente:

- Imprime el mensaje *Los tres primeros elementos de la lista son:*. A continuación, utilice una rebanada para imprimir los tres primeros elementos de la lista de ese programa.
- Imprime el mensaje *Tres elementos de la mitad de la lista son:*. Utilice un corte para imprimir tres elementos de la mitad de la lista.
- Imprime el mensaje *Los tres últimos elementos de la lista son:*. Utilice un corte para imprimir los tres últimos elementos de la lista.

4-11. Mis Pizzas, Tus Pizzas: Comience con su programa del Ejercicio 4-1 (página 60). Haz una copia de la lista de pizzas, y llámala `amigo_pizzas`. Luego, haz lo siguiente:

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

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

Tuplas

Las listas funcionan bien para almacenar conjuntos de elementos que pueden cambiar a lo largo de la vida de un programa. La posibilidad de modificar listas es especialmente importante cuando s e trabaja con una lista de usuarios en un sitio web o una lista de personajes en un juego. Sin embargo, a veces querrá crear una lista de elementos que no puede cambiar. Las tuplas te permiten hacer precisamente eso. Python se refiere a los valores que no pueden cambiar como *inmutables*, y una lista inmutable se llama *tupla*.

Definición de una tupla

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

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

dimensiones.py
```
❶ dimensiones = (200, 50)
❷ print(dimensiones[0])
print(dimensiones[1])
```

Definimos las `dimensiones` de la tupla en ❶, utilizando paréntesis en lugar de corchetes. En ❷ imprimimos cada elemento de la tupla individualmente, utilizando la misma sintaxis que hemos estado utilizando para acceder a los elementos de una lista:

```
200
50
```

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

```
dimensiones = (200, 50)
❶ dimensiones[0] = 250
```

El código de ❶ intenta cambiar el valor de la primera dimensión, pero Python devuelve un error de tipo. Básicamente, porque estamos tratando de alterar una tupla, lo que no se puede hacer a ese tipo de objeto, Python nos dice que no podemos asignar un nuevo valor a un elemento de una tupla:

```
Traceback (última llamada más reciente):
  File "dimensiones.py", line 3, in <module>
    dimensiones[0] = 250
TypeError: El objeto 'tuple' no admite la asignación de elementos
```

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

Recorrer en bucle todos los valores de una tupla

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

```
dimensiones = (200, 50)
para dimensión en dimensiones:
    print(dimensión)
```

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

```
200
50
```

Escribir sobre una tupla

Aunque no se puede modificar una tupla, sí se puede asignar un nuevo valor a una variable que contenga una tupla. Así que si quisiéramos cambiar nuestras dimensiones, podríamos redefinir toda la tupla:

```
❶ dimensiones = (200, 50)
print("Dimensiones
originales:") for dimensión in
dimensiones:
    print(dimensión)

❷ dimensiones = (400, 100)
❸ print("\nDimensiones
modificadas:") for dimensión in
dimensiones:
    print(dimensión)
```

El bloque ❶ define la tupla original e imprime las dimensiones iniciales. En ❷ almacenamos una nueva tupla en las `dimensiones` variables. Luego imprimimos las nuevas dimensiones en ❸. Python no produce ningún error esta vez, porque sobrescribir una variable es válido:

```
Dimensiones originales:
200
50

Dimensiones modificadas:
400
100
```

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

TRY IT yOURSELF

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

- Utiliza un bucle `for` para imprimir cada comida que ofrece el restaurante.
- Intente modificar uno de los elementos y asegúrese de que Python rechaza el cambio.
- El restaurante cambia su menú, sustituyendo dos de los elementos por alimentos diferentes. Añade un bloque de código que reescriba la tupla y, a continuación, utiliza un bucle `for` para imprimir cada uno de los elementos del menú revisado.

Estilizar su código

Ahora que estás escribiendo programas más largos, vale la pena conocer ideas sobre cómo dar estilo a tu código. Tómate el tiempo necesario para que tu código sea lo más fácil de leer posible. Escribir código fácil de leer te ayuda a seguir la pista de lo que están haciendo tus pro- gramas y ayuda a otros a entender tu código también.

Los programadores de Python se han puesto de acuerdo en una serie de convenciones de estilo para asegurar que el código de todo el mundo está estructurado más o menos de la misma manera. Una vez que hayas aprendido a escribir código Python limpio, deberías ser capaz de entender la estructura general del código Python de cualquier otra persona, siempre y cuando sigan las mismas pautas. Si esperas convertirte en un programador profesional en algún momento, deberías empezar a seguir estas pautas lo antes posible para desarrollar buenos hábitos.

Guía de estilo

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

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

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

Indentación

PEP 8 recomienda utilizar cuatro espacios por nivel de sangría. El uso de cuatro espacios mejora la legibilidad a la vez que deja espacio para múltiples niveles de sangría en cada línea.

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

Mezclar tabuladores y espacios en un archivo puede causar problemas difíciles de diagnosticar. Si crees que tienes una mezcla de tabuladores y

espacios, puedes convertir todos los tabuladores de un archivo en espacios en la mayoría de los editores.

Longitud de línea

Muchos programadores de Python recomiendan que cada línea tenga menos de 80 caracteres. Históricamente, esta directriz se desarrolló porque en la mayoría de los ordenadores sólo cabían 79 caracteres en una sola línea de una ventana de terminal.
Actualmente, en las pantallas caben líneas mucho más largas, pero existen otras razones para respetar la longitud de línea estándar de 79 caracteres. Los programadores profesionales a menudo tienen varios archivos abiertos en la misma pantalla, y usar la longitud de línea estándar les permite ver líneas enteras en dos o tres archivos que están abiertos uno al lado del otro en la pantalla. PEP 8 también recomienda que limite todos sus comentarios a 72 caracteres por línea, porque algunas de las herramientas que generan documentación automática para proyectos grandes añaden caracteres de formato al principio de cada línea comentada.

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

NOTE *El Apéndice B muestra cómo configurar el editor de texto para que inserte siempre cuatro espacios cada vez que pulse la tecla* TAB *y muestra una pauta vertical para ayudarle a respetar el límite de 79 caracteres.*

Líneas en blanco

Para agrupar partes de su programa visualmente, utilice líneas en blanco. Deberías utilizar líneas en blanco para organizar tus archivos, pero no lo hagas en exceso. Siguiendo los ejemplos proporcionados en este libro, debería encontrar el equilibrio adecuado. Por ejemplo, si tienes cinco líneas de código que construyen una lista, y luego otras tres líneas que hacen algo con esa lista, es apropiado colocar una línea en blanco entre las dos secciones. Sin embargo, no 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. El intérprete de Python utiliza la sangría horizontal para interpretar el significado de tu código, pero no tiene en cuenta el espaciado vertical.

Otras normas de estilo

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

PEP 8.

TRY IT yOURSELF

4-14. PEP 8: Revisa la guía de estilo original de PEP 8 en *https://python.org/dev/peps/pep-0008/*. No usarás mucho de ella ahora, pero puede ser interesante ojearla.

4-15. Revisión del Código: Elija tres de los programas que ha escrito en este capítulo y modifique cada uno para cumplir con PEP 8:

- Utilice cuatro espacios para cada nivel de sangría. Configure su editor de texto para que inserte cuatro espacios cada vez que pulse TAB, si aún no lo ha hecho (consulte el Apéndice B para obtener instrucciones sobre cómo hacerlo).
- Utilice menos de 80 caracteres en cada línea y configure su editor para que muestre una línea vertical en la posición del carácter 80.
- No utilice excesivamente líneas en blanco en los archivos de programa.

Resumen

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

En el Capítulo 5, aprenderá a responder adecuadamente a diferentes condiciones utilizando sentencias `if`. Aprenderá a encadenar conjuntos relativamente complejos de pruebas condicionales para responder adecuadamente al tipo exacto de situación o información que esté buscando. También aprenderá a utilizar `if`
para realizar acciones específicas con los elementos seleccionados de una lista.

5

IF STATEmENTS



Programar suele implicar examinar un conjunto de condiciones y decidir qué acción tomar en función de ellas.

La sentencia `if` de Python permite examinar el estado actual de un programa y responder adecuadamente a ese estado.

En este capítulo aprenderá a escribir pruebas condicionales, que le permitirán comprobar cualquier condición de interés. Aprenderás a escribir sentencias `if` simples, y aprenderás a crear una serie más compleja de sentencias `if` para identificar cuándo se dan las condiciones exactas que deseas. Luego aplicarás este concepto a las listas, de modo que podrás escribir un bucle `for` que maneje la mayoría de los elementos de una lista de una manera, pero que maneje ciertos elementos con valores específicos de una manera diferente.

Un ejemplo sencillo

El siguiente breve ejemplo muestra cómo las pruebas `if` le permiten responder correctamente a situaciones especiales. Imagine que tiene una lista de coches y quiere imprimir el nombre de cada uno de ellos. Los nombres de los coches son nombres propios, por lo que los nombres de la mayoría de los coches deberían imprimirse en mayúsculas. Sin embargo, el valor `'bmw'` debería imprimirse en mayúsculas. El siguiente código recorre una lista de coches
y busca el valor `'bmw'`. Cuando el valor es `'bmw'`, se imprime en mayúsculas en lugar de en mayúsculas:

coches.py

```
coches = ['audi', 'bmw', 'subaru',

'toyota'] para coche en coches:
    if coche == 'bmw':
        print(coche.superi
        or())
    si no:
        print(coche.titulo())
```

❶

El bucle de este ejemplo comprueba primero si el valor actual de `coche` es `'bmw'` ❶. Si lo es, el valor se imprime en mayúsculas. Si el valor de `car` es cualquier otra cosa que no sea `'bmw', se imprime en mayúsculas`:

```
Audi
BMW
Subaru
Toyota
```

Este ejemplo combina varios de los conceptos que aprenderás en este capítulo. Empecemos por ver los tipos de pruebas que puedes utilizar para examinar las condiciones de tu programa.

Pruebas condicionales

En el corazón de cada sentencia `if` hay una expresión que puede ser evaluada como `Verdadero` o `Falso` y que se llama *prueba condicional*. Python utiliza los valores `True` y `False` para decidir si el código de una sentencia `if` debe ejecutarse. Si una prueba condicional se evalúa como `True`, Python ejecuta el código que sigue a la sentencia `if`. Si el resultado es `Falso`, Python ignora el código que sigue a la `sentencia if`.

Comprobación de 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 al valor de interés:

```
❶ >>> coche = 'bmw'
```

❷ >>> **coche == 'bmw'**
Verdadero

La línea ❶ establece el valor de `car` en `'bmw'` utilizando un solo signo igual, como ya has visto muchas veces. La línea ❷ comprueba si el valor de `car` es `'bmw'` con un signo igual doble (`==`). Este operador *de igualdad* devuelve `True` si los valores a la izquierda y a la derecha del operador coinciden, y `False` si no coinciden. Los valores de este ejemplo coinciden, por lo que Python devuelve `True`.

Cuando el valor de `coche` es distinto de `"bmw"`, esta prueba devuelve `Falso`:

```
❶ >>> coche = 'audi'
❷ >>> coche == 'bmw'
Falso
```

Un signo igual simple es realmente una declaración; podrías leer el código en ❶ como "Establece el valor de car igual a `'audi'`". Por otro lado, un signo igual doble, como el de ❷, hace una pregunta: "¿Es el valor de coche igual a `'bmw'`?". La mayoría de los lenguajes de programación utilizan los signos igual de esta forma.

Ignorar mayúsculas y minúsculas al comprobar la igualdad

En Python, las pruebas de igualdad distinguen entre mayúsculas y minúsculas. Por ejemplo, dos valores con mayúsculas diferentes no se consideran iguales:

```
>>> coche = 'Audi'
>>> coche == 'audi'
Falso
```

Si las mayúsculas y minúsculas importan, este comportamiento es ventajoso. Pero si el caso no importa y en su lugar sólo desea probar el valor de una variable, puede convertir el valor de la variable a minúsculas antes de hacer la comparación:

```
>>> coche = 'Audi'
>>> car.lower() == 'audi'
Verdadero
```

Esta prueba devolverá `True` independientemente de cómo se formatee el valor `"Audi"`, porque ahora la prueba no distingue entre mayúsculas y minúsculas. La función `lower()` no cambia el valor almacenado originalmente en `car`, por lo que puede realizar este tipo de comparación sin afectar a la variable original:

```
❶ >>> coche = 'Audi'
❷ >>> car.lower() == 'audi'
Verdadero
❸ >>> coche
Audi
```

En ❶ almacenamos la cadena `'Audi'` en mayúsculas en la variable `coche`. En ❷ convertimos el valor de `car` a minúsculas y comparamos el valor de minúsculas

a la cadena 'audi'. Las dos cadenas coinciden, por lo que Python devuelve True. En ❸ podemos ver que el valor almacenado en coche no se ha visto afectado por la prueba condicional.

Los sitios web aplican ciertas reglas a los datos que introducen los usuarios de forma similar a ésta. Por ejemplo, un sitio puede utilizar una prueba condicional como ésta para asegurarse de que cada usuario tiene un nombre de usuario realmente único, y no sólo una variación de las mayúsculas del nombre de usuario de otra persona. Cuando alguien introduce un nuevo nombre de usuario, éste se convierte a minúsculas y se compara con las versiones en minúsculas de todos los nombres de usuario existentes. Durante esta comprobación, un nombre de usuario como "Juan" será rechazado si ya se utiliza cualquier variación de "Juan".

Comprobación de la desigualdad

Cuando desee determinar si dos valores no son iguales, puede combinar un signo de exclamación y un signo igual (!=). El signo de exclamación representa *no*, como ocurre en muchos lenguajes de programación.

Utilicemos otra sentencia if para examinar cómo utilizar el operador de desigualdad. Almacenaremos un ingrediente de pizza solicitado en una variable y luego imprimiremos un mensaje si la persona no pidió anchoas:

toppings.py

```
tapa_solicitada = 'champiñones'

❶ if tapa_solicitada != 'anchoas':
    print("¡Detenga las anchoas!")
```

La línea en ❶ 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 los dos valores coinciden, Python devuelve False y no ejecuta el código que sigue a la sentencia if.

Como el valor de requested_topping no es 'anchoas', se ejecuta el estado de impresión:

```
¡Sujeta las anchoas!
```

La mayoría de las expresiones condicionales que escriba comprobarán la igualdad, pero a veces le resultará más eficaz comprobar la desigualdad.

Comparaciones numéricas

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

```
>>> edad = 18
>>> edad == 18
Verdadero
```

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

magic_número.py

```
respuesta = 17

❶ si respuesta != 42:
    print("Esa no es la respuesta correcta. ¡Inténtalo de nuevo!")
```

La prueba condicional en ❶ pasa, porque el valor de la `respuesta` (17) no es igual a 42. Como la prueba pasa, se ejecuta el bloque de código sangrado:

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

También puede incluir varias comparaciones matemáticas en sus sentencias condicionales, como menos que, menos que o igual a, mayor que y mayor que o igual a:

```
>>> edad = 19
>>> edad < 21
Verdadero
>>> edad <= 21
Verdadero
>>> edad > 21
Falso
>>> edad >= 21
Falso
```

Cada comparación matemática puede utilizarse como parte de una sentencia `if`, que puede ayudarle a detectar las condiciones exactas de interés.

Comprobación de condiciones múltiples

Es posible que desee comprobar varias condiciones al mismo tiempo. Por ejemplo, a veces puede necesitar que dos condiciones sean `Verdaderas` para realizar una acción. Otras veces puede estar satisfecho con que sólo una condición sea `Verdadera`. Las palabras clave `and` y `or` pueden ayudarle en estas situaciones.

Uso de y para comprobar varias condiciones

Para comprobar si dos condiciones son `verdaderas` simultáneamente, utilice la palabra clave `y` para combinar las dos pruebas condicionales; si cada prueba se supera, la expresión global se evalúa como `Verdadero`. Si una de las pruebas falla o ambas fallan, la expresión se evalúa como `Falso`.

Por ejemplo, puedes comprobar si dos personas tienen más de 21 años utilizando la siguiente prueba:

```
❶ >>> edad_0 = 22
>>> edad_1 = 18
```

❷ >>> **edad_0 >= 21 y edad_1 >= 21**
Falso

```
❸ >>> edad_1 = 22
>>> edad_0 >= 21 y edad_1 >= 21
Verdadero
```

En ❶ definimos dos edades, `edad_0` y `edad_1`. En ❷ comprobamos si ambas edades tienen 21 años o más. La prueba de la izquierda pasa, pero la de la derecha falla, por lo que la expresión condicional global se evalúa como `False`. En ❸ cambiamos `edad_1` a 22. El valor de `edad_1` es ahora mayor que 21, por lo que ambas pruebas individuales pasan, haciendo que la expresión condicional global se evalúe como `Verdadero`.

Para mejorar la legibilidad, puede utilizar paréntesis alrededor de las pruebas individuales, pero no es obligatorio. Si utiliza paréntesis, su prueba tendría el siguiente aspecto:

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

Uso de o para comprobar varias condiciones

La palabra clave `or` también permite comprobar varias condiciones, pero se supera cuando se supera una o las dos pruebas individuales. Una expresión `or` sólo falla cuando fallan las dos pruebas individuales.

Volvamos a considerar dos edades, pero esta vez buscaremos que sólo un per- sonal tenga más de 21 años:

```
❶ >>> edad_0 = 22
  >>> edad_1 = 18
❷ >>> edad_0 >= 21 o edad_1 >= 21
Verdadero
❸ >>> edad_0 = 18
>>> edad_0 >= 21 o edad_1 >= 21
Falso
```

Volvemos a empezar con dos variables de edad en ❶. Debido a que la prueba para `age_0` en ❷ pasa, la expresión global se evalúa como `True`. A continuación, bajamos `age_0` a 18. En la prueba de ❸, ahora ambas pruebas fallan y la expresión global se evalúa como `Falso`.

Comprobar si un valor está en una lista

A veces es importante comprobar si una lista contiene un determinado valor antes de realizar una acción. Por ejemplo, es posible que desee comprobar si un nuevo nombre de usuario ya existe en una lista de nombres de usuario actuales antes de completar el registro de alguien en un sitio web. En un proyecto de cartografía, es posible que desee comprobar si una ubicación enviada ya existe en una lista de ubicaciones conocidas.

Para saber si un determinado valor ya está en una lista, utilice la palabra clave `in`. Veamos un ejemplo de código para una pizzería. Haremos una lista de los ingredientes que un cliente ha pedido para una pizza y luego comprobaremos si ciertos ingredientes están en la lista.

```
>>> requested_toppings = ['champiñones', 'cebollas', 'piña']
❶ >>> 'champiñones' en requested_toppings
Verdadero
❷ >>> 'pepperoni' in requested_toppings
Falso
```

En ❶ y ❷, la palabra clave `in` le dice a Python que compruebe la existencia de `'champiñones'` y `'pepperoni'` en la lista `requested_toppings`. Esta técnica es bastante potente porque puedes crear una lista de valores esenciales y, a continuación, comprobar fácilmente si el valor que estás comprobando coincide con alguno 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. Puede utilizar la palabra clave `not` en esta situación. Por ejemplo, considere una lista de usuarios que tienen prohibido comentar en un foro. Puede comprobar si un usuario ha sido vetado antes de permitir que esa persona envíe un comentario:

baneados_usuarios.py

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

❶ si el usuario no está en banned_users:
    print(user.title() + ", puede publicar una respuesta si lo desea.")
```

La línea de ❶ se lee bastante clara. Si el valor de `usuario` no está en la lista `banned_users`, Python devuelve `True` y ejecuta la línea sangrada.

El usuario `'marie'` no está en la lista `banned_users`, por lo que ve un mensaje invitándola a enviar una respuesta:

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

Expresiones booleanas

A medida que aprendas más sobre programación, escucharás el término expresión *booleana* en algún momento. Una expresión booleana es sólo otro nombre para una prueba condicional. Un *valor booleano* es `Verdadero` o `Falso`, igual que el valor de una expresión condicional después de haber sido evaluada.

Los valores booleanos se utilizan a menudo para realizar un seguimiento de ciertas condiciones, como si un juego se está ejecutando o si un usuario puede editar cierto contenido en un sitio web:

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

Los valores booleanos proporcionan una forma eficaz de realizar un seguimiento del estado de un programa o de una condición concreta que

sea importante en su programa.

TRY IT yOURSELF

5-1. Pruebas condicionales: Escriba una serie de pruebas condicionales. Imprima una declaración describiendo cada prueba y su predicción para los resultados de cada prueba. Su código debe ser algo como esto

```
coche = 'subaru'
print("¿Es coche == 'subaru'? predigo
Verdadero.") print(coche == 'subaru')

print("¿es coche == 'audi'? predigo Falso.")
print(coche == 'audi')
```

- Fíjate bien en los resultados y asegúrate de que entiendes por qué cada línea se evalúa como `Verdadero` o `Falso`.
- Crea al menos 10 pruebas. Haz que al menos 5 pruebas evalúen `Verdadero` y otras 5 pruebas evalúen `Falso`.

5-2. Más pruebas condicionales: No tienes que limitar el número de pruebas que crees a 10. Si quieres probar más comparaciones, escribe más pruebas y añádelas a *conditional_tests.py*. Tenga al menos un resultado `Verdadero` y uno `Falso` para cada uno de los siguientes:

- Pruebas de igualdad y desigualdad con cadenas
- Pruebas con la función `lower()`
- Pruebas numéricas de 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`
- Comprobar si un elemento está en una lista
- Comprobar si un elemento no está en una lista

Declaraciones if

Cuando entienda las pruebas condicionales, podrá empezar a escribir sentencias `if`. Existen varios tipos de sentencias `if`, y su elección dependerá del número de condiciones que necesite probar. Ha visto varios ejemplos de sentencias `if` en la discusión sobre pruebas condicionales, pero ahora vamos a profundizar en el tema.

Sentencias if sencillas

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

```
if
    prueba_condicion
    al: hacer algo
```

Puedes poner cualquier prueba condicional en la primera línea y casi cualquier acción en el bloque sangrado que sigue a la prueba. Si el resultado de la prueba condicional es `True`, Python ejecuta el código que sigue a la sentencia `if`. Si el resultado es `Falso`, Python ignora el código que sigue a la sentencia `if`.

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

votar.py

```
edad = 19
❶ si edad >= 18 años:
❷     print("¡Tienes edad para votar!")
```

En ❶ Python comprueba si el valor de la `edad` es mayor o igual que 18 años. Lo es, por lo que Python ejecuta la sentencia `print` sangrada en ❷:

```
Ya tienes edad para votar.
```

La sangría desempeña el mismo papel en las sentencias `if` que en los bucles `for`. Todas las líneas sangradas después de una sentencia `if` se ejecutarán si la prueba pasa, y todo el bloque de líneas sangradas se ignorará si la prueba no pasa.
no pasar.

Puede tener tantas líneas de código como desee en el bloque que sigue a la sentencia `if`. Añadamos otra línea de salida si la persona tiene edad suficiente para votar, preguntando si ya se ha registrado para votar:

```
edad = 19 años
si edad >= 18:
    print("¡Tienes edad para votar!")
    print("¿Ya te has registrado para
    votar?")
```

La prueba condicional pasa, y ambas sentencias de `impresión` tienen sangría, por lo que se imprimen ambas líneas:

```
Ya tienes edad para votar.
¿Ya se ha inscrito para votar?
```

Si el valor de la `edad` es inferior a 18 años, este programa no produciría ningún resultado.

Sentencias if-else

A menudo, querrá tomar una acción cuando una prueba condicional pasa y una acción diferente en todos los demás casos. La sintaxis `if-else` de Python lo hace posible. Un bloque `if-else` es similar a una simple sentencia `if`, pero la sentencia `else permite` definir una acción o conjunto de acciones que se ejecutan cuando la prueba condicional falla.

Mostraremos el mismo mensaje que teníamos anteriormente si la persona tiene edad suficiente para votar, pero esta vez añadiremos un mensaje para cualquiera que no tenga edad suficiente para votar:

```
edad = 17 años
❶ si edad >= 18 años:
      print("¡Tienes edad para votar!")
      print("¿Ya te has registrado para
      votar?")
❷ si no:
      print("Lo sentimos, eres demasiado joven para votar")
      print("¡Por favor, regístrate para votar en cuanto cumplas 18 años!")
```

Si la prueba condicional en ❶ pasa, se ejecuta el primer bloque de sentencias de `impresión` indentadas. Si la prueba es `falsa`, se ejecuta el bloque `else` en ❷. Como esta vez la `edad es` menor de 18 años, la prueba condicional falla y se ejecuta el código del bloque `else`:

```
Lo siento, eres demasiado joven para votar.
Inscríbete para votar en cuanto cumplas 18 años.
```

Este código funciona porque sólo tiene dos situaciones posibles que evaluar: una persona tiene edad suficiente para votar o no tiene edad suficiente para votar. El `if-else`
funciona bien en situaciones en las que se desea que Python ejecute siempre una de dos acciones posibles. En una simple cadena `if-else como` esta, siempre se ejecutará una de las dos acciones.

La cadena if-elif-else

A menudo, necesitarás probar más de dos situaciones posibles, y para evaluarlas puedes utilizar la sintaxis `if-elif-else` de Python. Python ejecuta sólo un bloque en una cadena `if-elif-else`. Ejecuta cada prueba condicional en orden hasta que una pasa. Cuando una prueba pasa, el código que sigue a esa prueba se ejecuta y Python se salta el resto de las pruebas.

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

- La entrada para los menores de 4 años es gratuita.
- La entrada para cualquier persona de entre 4 y 18 años cuesta 5 $.
- La entrada para mayores de 18 años cuesta 10 $.

¿Cómo podemos utilizar una sentencia `if` para determinar el precio de admisión de una persona? El siguiente código comprueba el grupo de edad de una persona y luego imprime un mensaje de precio de admisión:

diversión_aparcar.py

```
edad = 12
```

```
❶ si edad < 4 años:
      print("Su coste de admisión es de 0$")
```

```
❷ elif edad < 18:
      print("El precio de la entrada es de 5 dólares")
❸ si no:
      print("El precio de la entrada es de 10 dólares")
```

La prueba `if` en ❶ comprueba si una persona tiene menos de 4 años. Si pasa la prueba, se imprime un mensaje apropiado y Python se salta el resto de las pruebas. La línea `elif` en ❷ es en realidad otra prueba `if`, que se ejecuta sólo si la prueba anterior falló. En este punto de la cadena, sabemos que la persona es al menos
4 años porque la primera prueba falló. Si la persona es menor de 18 años, se imprime un mensaje apropiado y Python se salta el bloque `else`. Si ambas pruebas `if` y `elif` fallan, Python ejecuta el código en el bloque `else` en ❸.

En este ejemplo la prueba en ❶ evalúa a `Falso, por lo que` su bloque de código no se ejecuta. Sin embargo, la segunda prueba es `verdadera` (12 es menor que 18), por lo que se ejecuta su código. El resultado es una frase que informa al usuario del coste de la entrada:

```
El precio de la entrada es de 5 $.
```

Cualquier edad superior a 17 años provocaría el fallo de las dos primeras pruebas. En estas situaciones, se ejecutaría el bloque `else` y el precio de la entrada sería de 10 $.

En lugar de imprimir el precio de admisión dentro del bloque `if-elif-else`, sería más conciso establecer sólo el precio dentro de la cadena `if-elif-else` y luego tener una simple sentencia `print` que se ejecute después de que la cadena haya sido evaluada:

```
edad = 12 años

si edad < 4 años:
❶     precio = 0
elif edad <
18:
❷     precio =
5 else:
❸     precio = 10

❹ print("El precio de tu entrada es de $" + str(precio) + ".")
```

Las líneas ❶, ❷ y ❸ establecen el valor del `precio` según la edad de la persona, como en el ejemplo anterior. Después de que la cadena `if-elif-else` establezca el precio, una sentencia `print` sin sangría ❹ utiliza este valor para mostrar un mensaje informando del precio de admisión de la persona.

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

mensaje de salida, sólo tendría que cambiar una sentencia `print` en lugar de tres sentencias `print` separadas.

Uso de varios bloques elif

Puede utilizar tantos bloques `elif` en su código como desee. Por ejemplo, si el parque de atracciones implementara un descuento para personas mayores, podría añadir una prueba condicional más al código para determinar si alguien tiene derecho al descuento. Digamos que cualquier persona de 65 años o más paga la mitad de la entrada normal, o 5$:

```
edad = 12 años

si edad < 4 años:
    precio = 0
elif edad <
18:
   precio = 5
❶ elif edad < 65:
     precio = 10
❷ si no:
     precio = 5

print("Su precio de entrada es de $" + str(precio) + ".")
```

La mayor parte de este código no ha cambiado. El segundo bloque `elif` en ❶ ahora comprueba que la persona es menor de 65 años antes de asignarle la tarifa de admisión completa de 10 $. Observa que el valor asignado en el bloque `else` en ❷ debe cambiarse a $5, porque las únicas edades que llegan a este bloque son las personas de 65 años o más.

Omitir el bloque else

Python no requiere un bloque `else` al final de una cadena `if-elif`. A veces un bloque `else` es útil; a veces es más claro usar una sentencia `elif` adicional que capture la condición específica de interés:

```
edad = 12 años

si edad < 4 años:
    precio = 0
elif edad <
18:
    precio = 5
elif edad <
65:
   precio = 10
❶ elif edad >= 65:
     precio = 5

print("El precio de su entrada es de $" + str(precio) + ".")
```

El bloque `elif` adicional en ❶ asigna un precio de 5 dólares cuando la persona tiene 65 años o más, lo que es un poco más claro que el bloque `else` general. Con este cambio, cada bloque de código debe superar una prueba específica para poder ejecutarse.

El bloque `else` es una sentencia catchall. Coincide con cualquier condición que no coincida con una prueba `if` o `elif` específica, y que a veces puede incluir datos no válidos o incluso maliciosos. Si tiene una condición final específica que está comprobando, considere la posibilidad de utilizar un bloque `elif` final y omitir el bloque `else`. Como resultado, ganará confianza extra en que su código se ejecutará sólo bajo las condiciones correctas.

Comprobación de múltiples condiciones

La cadena `if-elif-else` es potente, pero sólo es apropiada cuando sólo necesitas que pase una prueba. Tan pronto como Python encuentra una prueba que pasa, se salta el resto de las pruebas. Este comportamiento es beneficioso, porque es eficiente y te permite probar una condición específica.

Sin embargo, a veces es importante comprobar todas las condiciones de interés. En este caso, debe utilizar una serie de sentencias `if` simples sin bloques `elif` o `else`. Esta técnica tiene sentido cuando más de una condición podría ser `True`, y desea actuar sobre cada condición que sea `True`.

Volvamos al ejemplo de la pizzería. Si alguien pide una pizza con dos ingredientes, tendrá que asegurarse de incluir los dos ingredientes en su pizza:

toppings.py

```
❶ requested_toppings = ['champiñones', 'queso extra']

❷ if 'setas' in setas_solicitadas:
      print("Añadiendo setas.")
❸ if 'pepperoni' in requested_toppings:
      print("Añadiendo pepperoni.")
❹ if 'extra cheese' in requested_toppings:
      print("Añadiendo queso extra.")

print("¡Has terminado de hacer tu pizza!")
```

Empezamos en ❶ con una lista que contiene los ingredientes solicitados. La sentencia `if` en ❷ comprueba si la persona ha pedido champiñones en su pizza. Si es así, se imprime un mensaje confirmando ese ingrediente. La prueba para pepperoni en ❸ es otra sentencia `if` simple, no una sentencia `elif` o `else`, por lo que esta prueba se ejecuta independientemente de si la prueba anterior pasó o no. El código en ❹ comprueba si se ha pedido queso extra independientemente de los resultados de las dos primeras pruebas. Estas tres pruebas independientes se ejecutan cada vez que se ejecuta el programa.

Como se evalúan todas las condiciones de este ejemplo, se añaden a la pizza tanto champiñones como queso extra:

```
Añadir champiñones.
Añadiendo queso extra.

¡Terminaste de hacer tu pizza!
```

Este código no funcionaría correctamente si utilizáramos un bloque `if-elif-else`, porque el código dejaría de ejecutarse después de pasar una sola prueba. Así es como quedaría:

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

if 'champiñones' in requested_toppings:
    print("Añadiendo setas.")
elif 'pepperoni' in ingredientes_solicitados:
    print("Añadiendo pepperoni.")
elif 'extra cheese' in requested_toppings:
    print("Añadiendo queso extra.")

print("¡Has terminado de hacer tu pizza!")
```

La prueba de `"champiñones"` es la primera que se supera, por lo que se añaden champiñones a la pizza. Sin embargo, los valores `"queso extra"` y `"pepperoni"` nunca se comprueban, porque Python no ejecuta ninguna prueba más allá de la primera prueba que pasa en una cadena `if-elif-else`. Se añadirá el primer ingrediente del cliente, pero se omitirán todos los demás:

```
Añadir setas.

¡Terminaste de hacer tu pizza!
```

En resumen, si sólo quieres que se ejecute un bloque de código, utiliza una cadena `if-elif- else`. Si necesita que se ejecute más de un bloque de código, utilice una serie de sentencias `if` independientes.

TRY IT yOURSELF

5-3. Alien Colors #1: Imagina que un alien acaba de ser derribado en un juego. Crea una variable llamada `color_alien` y asígnale un valor de `'verde'`, `'amarillo'`, o `'rojo'`.

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

5-4. Colores de los Alienígenas #2: Elija un color para un alienígena como hizo en el Ejercicio 5-3, y escriba una cadena `if-else`.

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

5-5. Colores alienígenas #3: Convierte tu cadena `if-else` del Ejercicio 5-4 en una cadena `if-elif- else`.

- Si el alienígena es verde, imprime un mensaje indicando que el jugador ha ganado 5 puntos.
- Si el alienígena es amarillo, imprime un mensaje indicando que el jugador ha ganado 10 puntos.
- Si el alienígena es rojo, imprime un mensaje indicando que el jugador ha ganado 15 puntos.
- Escriba tres versiones de este programa, asegurándose de que cada mensaje se imprime para el alienígena de color apropiado.

5-6. Etapas de la vida: Escribe una cadena `if-elif-else` que determine la etapa de la vida de una persona. Establezca un valor para la variable `edad`, y luego:

- Si la persona tiene menos de 2 años, imprime un mensaje que indique que la persona es un bebé.
- Si la persona tiene al menos 2 años pero menos de 4, imprime un mensaje que indique que la persona es un niño pequeño.
- Si la persona tiene al menos 4 años pero menos de 13, imprime un mensaje que indique que la persona es un niño.
- Si la persona tiene al menos 13 años pero menos de 20, imprime un mensaje que indique que se trata de un adolescente.
- Si la persona tiene al menos 20 años pero menos de 65, imprima un mensaje que indique que la persona es adulta.
- Si la persona tiene 65 años o más, imprima un mensaje indicando que se trata de un anciano.

5-7. Fruta favorita: Haz una lista de tus frutas favoritas y, a continuación, escribe una serie de sentencias `if` independientes que comprueben la presencia de determinadas frutas en tu lista.

- Haz una lista de tus tres frutas favoritas y llámala `frutas_favoritas`.
- Escriba cinco sentencias `if`. Cada una de ellas debe comprobar si un determinado tipo de fruta está en tu lista. Si la fruta está en tu lista, el bloque `if` debe imprimir una declaración, como *¡Te gustan mucho los plátanos!*

Uso de sentencias if con listas

La combinación de listas y estados `if` puede resultar muy interesante. Puede vigilar valores especiales que deban tratarse de forma diferente a otros valores de la lista. Puede gestionar condiciones cambiantes de forma eficaz, como la disponibilidad de determinados artículos en un restaurante a lo largo de un turno. También puede empezar a probar que su código funciona como

espera en todas las situaciones posibles.

Comprobación de artículos especiales

Este capítulo comenzó con un ejemplo sencillo que mostraba cómo manejar un valor especial como `'bmw'`, que debía imprimirse en un formato diferente al de otros valores de la lista. Ahora que tiene un conocimiento básico de las pruebas condicionales y las sentencias `if`, veamos más de cerca cómo puede buscar valores especiales en una lista y manejarlos adecuadamente.

Sigamos con el ejemplo de la pizzería. La pizzería muestra un mensaje cada vez que se añade un topping a la pizza, mientras se está haciendo. El código para esta acción se puede escribir de forma muy eficiente haciendo una lista de los ingredientes que el cliente ha solicitado y utilizando un bucle para anunciar cada ingrediente a medida que se añade a la pizza:

toppings.py

```
requested_toppings = ['champiñones', 'pimientos verdes', 'queso

extra'] for requested_topping in requested_toppings:
    print("Añadiendo " + copete_solicitado + ".")

print("¡Has terminado de hacer tu pizza!")
```

La salida es sencilla porque este código es un simple bucle `for`:

```
Añadir champiñones.
Añadir pimientos
verdes. Añadir más
queso.

¡Terminaste de hacer tu pizza!
```

Pero, ¿y si la pizzería se queda sin pimientos verdes? Una sentencia `if` dentro del bucle `for` puede manejar esta situación adecuadamente:

```
requested_toppings = ['champiñones', 'pimientos verdes', 'queso

extra'] for requested_topping in requested_toppings:
(1)     if requested_topping == 'pimientos verdes':
            print("Lo sentimos, no nos quedan pimientos verdes en este momento.")
(2)     más:
            print("Añadiendo " + copete_solicitado + ".")

print("¡Has terminado de hacer tu pizza!")
```

Esta vez comprobamos cada elemento solicitado antes de añadirlo a la `pizza`. El código en (1) comprueba si la persona solicitó `pimientos` verdes. Si es así, mostramos un mensaje informándole de por qué no puede tener pimientos verdes. El bloque `else` en (2) asegura que todos los demás ingredientes se añadan a la pizza.

La salida muestra que cada topping solicitado se gestiona adecuadamente.

```
Añadir setas.
Lo siento, no tenemos pimientos verdes en este
momento. Añadiendo queso extra.

¡Terminaste de hacer tu pizza!
```

Comprobar que una lista no está vacía

Hemos hecho una suposición simple sobre cada lista con la que hemos trabajado hasta ahora; hemos asumido que cada lista tiene al menos un elemento en ella. Pronto dejaremos que los usuarios proporcionen la información que se almacena en una lista, por lo que no podremos asumir que una lista tiene algún elemento cada vez que se ejecuta un bucle. En esta situación, es útil comprobar si una lista está vacía antes de ejecutar un bucle `for`.

Por ejemplo, comprobemos si la lista de ingredientes solicitados está vacía antes de crear la pizza. Si la lista está vacía, preguntaremos al usuario si desea una pizza normal. Si la lista no está vacía, construiremos la pizza tal y como hicimos en los ejemplos anteriores:

```
❶ requested_toppings = []

❷ si se solicita_toppings:
      for tapa_solicitada in tapas_solicitadas:
          print("Añadiendo " + tapa_solicitada +
          ".")
      print("¡Has terminado de hacer tu pizza!")
❸ si no:
      print("¿Seguro que quieres una pizza normal?")
```

Esta vez empezamos con una lista vacía de coberturas solicitadas en ❶. En lugar de saltar directamente a un bucle `for`, hacemos una comprobación rápida en ❷. Cuando se utiliza el nombre de una lista en una sentencia `if`, Python devuelve `True` si la lista contiene al menos un elemento; una lista vacía se evalúa como `False`. Si `requested_toppings` pasa la prueba condicional, ejecutamos el mismo bucle `for` que usamos en el ejemplo anterior. Si la prueba condicional falla, imprimimos un mensaje preguntando al cliente si realmente quiere una pizza normal sin ingredientes ❸.

La lista está vacía en este caso, por lo que la salida pregunta si el usuario realmente quiere una pizza normal:

```
¿Seguro que quieres una pizza normal?
```

Si la lista no está vacía, la salida mostrará cada uno de los ingredientes solicitados añadidos a la pizza.

Uso de varias listas

La gente pide cualquier cosa, sobre todo si se trata de ingredientes para una pizza. ¿Y si un cliente quiere patatas fritas en la pizza? Puede utilizar listas y afirmaciones `del tipo "si"` para asegurarse de que su aportación tiene sentido antes de actuar en consecuencia.

Tengamos cuidado con las peticiones de ingredientes inusuales antes de construir una pizza. El siguiente ejemplo define dos listas. La primera es una lista de ingredientes disponibles en la pizzería, y la segunda es la lista de ingredientes solicitados por el usuario. Esta vez, cada elemento de `requested_toppings` se compara con la lista de ingredientes disponibles antes de añadirlo a la pizza:

```
❶ available_toppings = ['setas', 'aceitunas', 'pimientos verdes',
                        pepperoni", "piña", "queso extra"].

❷ requested_toppings = ['champiñones', 'patatas fritas', 'queso extra']

❸ para tapa_solicitada en tapas_solicitadas:
❹     if tapa_solicitada in tapas_disponibles:
          print("Añadiendo " + tapa_solicitada +
          ".")
❺     si no:
          print("Lo sentimos, no tenemos " + tapa_solicitada + ".")

print("¡Has terminado de hacer tu pizza!")
```

En ❶ definimos una lista de los toppings disponibles en esta pizzería. Tenga en cuenta que esto podría ser una tupla si la pizzería tiene una selección estable de coberturas. En ❷, hacemos una lista de los toppings que ha pedido un cliente. Nótese lo inusual petición, `"patatas fritas"`. En ❸ hacemos un bucle con la lista de ingredientes solicitados.
Dentro del bucle, primero comprobamos si cada ingrediente solicitado está en la lista de ingredientes disponibles ❹. En caso afirmativo, se añade el ingrediente a la pizza. Si el ingrediente solicitado no está en la lista de ingredientes disponibles, se ejecuta `el bloque else` ❺. El `bloque else` imprime un mensaje indicando al usuario qué ingredientes no están disponibles.

Esta sintaxis de código produce una salida limpia e informativa:

```
Añadir setas.
Lo siento, no tenemos patatas
fritas. Añadiendo queso extra.

¡Terminaste de hacer tu pizza!
```

En unas pocas líneas de código, hemos resuelto una situación del mundo real con bastante eficacia.

TRY IT yOURSELF

5-8. Hola Admin: Haz una lista de cinco o más nombres de usuario, incluyendo el nombre `'admin'`. Imagina que estás escribiendo un código que imprimirá un saludo a cada usuario después de que se conecten a un sitio web. Recorre la lista e imprime un saludo a cada usuario:

- Si el nombre de usuario es `"admin"`, imprime un saludo especial, como *Hola admin, ¿desea ver un informe de estado?*
- En caso contrario, imprima un saludo genérico, como *Hola Eric, gracias por volver a conectarte.*

5-9. No hay usuarios: Añade un test `if` a *hello_admin.py* para asegurarte de que la lista de usuarios no está vacía.

- Si la lista está vacía, imprime el mensaje *¡Necesitamos encontrar algunos usuarios!*
- Elimine todos los nombres de usuario de su lista y asegúrese de que se imprime el mensaje correcto.

5-10. Comprobación de nombres de usuario: Haz lo siguiente para crear un programa que simule cómo los sitios web se aseguran de que todo el mundo tiene un nombre de usuario único.

- Haz una lista de cinco o más nombres de usuario llamada `usuarios_actuales`.
- Haz otra lista de cinco nombres de usuario llamada `nuevos_usuarios`. Asegúrate de que uno o dos de los nuevos nombres de usuario también están en la lista `usuarios_actuales`.
- Recorre la lista `new_users` para ver si ya se ha utilizado cada nuevo nombre de usuario. En caso afirmativo, imprime un mensaje indicando que la persona deberá introducir un nuevo nombre de usuario. Si un nombre de usuario no se ha utilizado, imprime un mensaje diciendo que el nombre de usuario está disponible.
- Asegúrese de que la comparación no distingue entre mayúsculas y minúsculas. Si se ha utilizado `'Juan'`, `"JUAN"` no debe aceptarse.

5-11. Números ordinales: Los números ordinales indican su posición en una lista, como *1º* o *2º*. La mayoría de los números ordinales terminan en *th*, excepto 1, 2 y 3.

- Guarda los números del 1 al 9 en una lista.
- Recorre la lista.
- Use una cadena `if-elif-else` dentro del bucle para imprimir el final ordinal apropiado para cada número. La salida debe ser `"1ro 2do 3ro 4to 5to 6to 7mo 8vo 9no"`, y cada resultado debe estar en una línea separada.

Estilo de las sentencias if

En cada ejemplo de este capítulo, has visto buenos hábitos de estilo. La única recomendación que PEP 8 proporciona para estilizar las pruebas condicionales es usar un solo espacio alrededor de los operadores de comparación, como ==, >=, <=. Por ejemplo:

```
si edad < 4 años:
```

es mejor que:

```
si edad<4:
```

Este espaciado no afecta a la forma en que Python interpreta tu código; sólo hace que tu código sea más fácil de leer para ti y para los demás.

TRY IT yOURSELF

5-12. Estilizando las sentencias if: Revise los programas que escribió en este capítulo, y asegúrese de que le dio el estilo apropiado a sus pruebas condicionales.

5-13. Tus ideas: En este punto, usted es un programador más capaz de lo que era cuando comenzó este libro. Ahora que tiene un mejor sentido de cómo las situaciones del mundo real son modeladas en programas, podría estar pensando en algunos problemas que podría resolver con sus propios programas. Anote cualquier nueva idea que tenga sobre problemas que quiera resolver a medida que sus habilidades de programación mejoren. Piensa en juegos que te gustaría escribir, conjuntos de datos que te gustaría explorar y aplicaciones web que te gustaría crear.

Resumen

En este capítulo ha aprendido a escribir pruebas condicionales, que siempre se evalúan como `True` o `False`. Aprendió a escribir sentencias `if` simples, cadenas `if-else`, y cadenas `if-elif-else`. Empezaste a usar estas estructuras para identificar condiciones particulares que necesitabas probar y para saber cuando esas condiciones se han cumplido en tus programas. Aprendiste a manejar ciertos elementos de una lista de manera diferente a todos los demás elementos, sin dejar de utilizar la eficiencia de un bucle `for`. También revisaste las recomendaciones de estilo de Python para asegurarte de que tus programas, cada vez más complejos, siguen siendo relativamente fáciles de leer y entender.

En el Capítulo 6 aprenderás sobre los diccionarios de Python. Un diccionario es similar a una lista, pero te permite conectar piezas de información. Aprenderás a construir diccionarios, recorrerlos y usarlos en combinación con listas y sentencias `if`. El aprendizaje de los diccionarios le permitirá modelar una variedad aún mayor de situaciones del mundo

real.

6

DICTIONARIES



En este capítulo aprenderás a usar los diccionarios de Python, que te permiten conectar piezas de información relacionada. Aprenderás

aprender a acceder a la información una vez que está en un diccionario y a modificarla. Dado que los diccionarios pueden almacenar una cantidad casi ilimitada de

cantidad de información, te mostraré cómo hacer un bucle a través de los datos de un diccionario. Además, aprenderás a anidar diccionarios dentro de listas, listas dentro de diccionarios e incluso diccionarios dentro de otros diccionarios.

Comprender los diccionarios te permitirá modelar con mayor precisión diversos objetos del mundo real. Podrás crear un diccionario que represente a una persona y almacenar toda la información que quieras sobre ella. Puedes almacenar su nombre, edad, ubicación, profesión y cualquier otro aspecto de una persona que puedas describir. Podrás almacenar dos tipos de información que se puedan emparejar, como una lista de palabras y sus significados, una lista de nombres de personas y sus

números favoritos, una lista de montañas y sus elevaciones, etcétera.

Un diccionario sencillo

Consideremos un juego con alienígenas que pueden tener diferentes colores y valores de puntos. Este sencillo diccionario almacena información sobre un alienígena concreto:

alien.py

```
alien_0 = {'color': 'verde', 'puntos': 5}

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

El diccionario `alien_0` almacena el color y el valor en puntos del alien. Los dos `print` acceden a esa información y la muestran, como se muestra aquí:

```
verde
5
```

Como ocurre con la mayoría de los nuevos conceptos de programación, el uso de diccionarios requiere práctica. Una vez que hayas trabajado un poco con los diccionarios, pronto verás la eficacia con la que pueden modelar situaciones del mundo real.

Trabajar con diccionarios

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

En Python, un diccionario se encierra entre llaves, `{}`, con una serie de pares clave-valor dentro de las llaves, como se muestra en el ejemplo anterior:

```
alien_0 = {'color': 'verde', 'puntos': 5}
```

Un *par clave-valor* es un conjunto de valores asociados entre sí. Cuando se proporciona una clave, Python devuelve el valor asociado a esa clave. Cada clave está unida a su valor por dos puntos, y los pares clave-valor se separan por comas. Puede almacenar tantos pares clave-valor como desee en un diccionario.

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

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

Este diccionario almacena una información sobre `alien_0`, a saber, el color del alien. La cadena `"color"` es una clave de este diccionario, y su valor asociado es `"verde"`.

Acceso a los valores de un diccionario

Para obtener el valor asociado a una clave, indique el nombre del diccionario y, a continuación, coloque la clave dentro de un conjunto de corchetes, como se muestra aquí:

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

Devuelve el valor asociado a la clave `"color"` del diccionario `alien_0`:

```
verde
```

En un diccionario puede haber un número ilimitado de pares clave-valor. Por ejemplo, aquí está el diccionario original `alien_0` con dos pares clave-valor:

```
alien_0 = {'color': 'verde', 'puntos': 5}
```

Ahora puedes acceder al color o al valor en puntos `del alien_0`. Si un jugador derriba a este alienígena, puedes ver cuántos puntos debería ganar usando un código como este:

```
alien_0 = {'color': 'verde', 'puntos': 5}

❶ nuevos_puntos = alien_0['puntos']
❷ print("¡Acabas de ganar " + str(nuevos_puntos) + " puntos!")
```

Una vez definido el diccionario, el código de ❶ extrae del diccionario el valor asociado a la clave `'puntos'`. Este valor se almacena en la variable `nuevos_puntos`. La línea en ❷ convierte este valor entero en una cadena e imprime una declaración sobre cuántos puntos acaba de ganar el jugador:

```
Acabas de ganar 5 puntos.
```

Si ejecutas este código cada vez que un alien es derribado, se recuperará el valor en puntos del alien.

Añadir nuevos pares clave-valor

Los diccionarios son estructuras dinámicas, y puedes añadir nuevos pares clave-valor a un diccionario en cualquier momento. Por ejemplo, para añadir un nuevo par clave-valor, debe indicar el nombre del diccionario seguido de la nueva clave entre corchetes junto con el nuevo valor.

Añadamos dos nuevos datos al diccionario `alien_0`: las coordenadas x e y del alien, que nos ayudarán a mostrarlo en una posición determinada de la pantalla. Coloquemos al alienígena en el borde izquierdo de la pantalla, a 25 píxeles de la parte superior. Como las coordenadas de pantalla suelen empezar en la esquina superior izquierda de la pantalla, colocaremos al alienígena en el borde izquierdo de la pantalla, a 25 píxeles de la parte superior.

borde de la pantalla fijando la coordenada x en 0 y a 25 píxeles de la parte superior fijando su coordenada y en 25 positivo, como se muestra aquí:

```
alien_0 = {'color': 'verde', 'puntos': 5}
print(alien_0)

❶ alien_0['x_position'] = 0
❷ alien_0['y_position'] = 25
print(alien_0)
```

Empezamos definiendo el mismo diccionario con el que hemos estado trabajando. A continuación, imprimimos este diccionario, mostrando una instantánea de su información. En ❶ añadimos un nuevo par clave-valor al diccionario: clave `'x_position'` y valor `0`. Hacemos lo mismo con la clave `"posición_y"` en ❷. Cuando imprimimos el diccionario modificado, vemos los dos pares clave-valor adicionales:

```
{'color': 'verde', 'puntos': 5}
{'color': 'verde', 'puntos': 5, 'y_position': 25, 'x_position': 0}
```

La versión final del diccionario contiene cuatro pares clave-valor. Los dos originales especifican el color y el valor del punto, y dos más especifican la posición del alienígena. Observe que el orden de los pares clave-valor no coincide con el orden en que los hemos añadido. A Python no le importa el orden en el que almacenas cada par clave-valor; sólo le importa la conexión entre cada clave y su valor.

Empezar con un diccionario vacío

A veces es conveniente, o incluso necesario, empezar con un diccionario vacío y luego añadirle cada nuevo elemento. Para comenzar a llenar un diccionario vacío, defina un diccionario con un conjunto vacío de llaves y luego agregue cada par clave-valor en su propia línea. Por ejemplo, así es como se construye el diccionario `alien_0` utilizando este método:

```
alien_0 = {}

alien_0['color'] = 'verde'
alien_0['puntos'] = 5

print(alien_0)
```

Aquí definimos un diccionario `alien_0` vacío, y luego le añadimos valores de color y puntos. El resultado es el diccionario que hemos estado utilizando en ejemplos anteriores:

```
{'color': 'verde', 'puntos': 5}
```

Normalmente, utilizará diccionarios vacíos cuando almacene datos

proporcionados por el usuario en un diccionario o cuando escriba código que genere automáticamente un gran número de pares clave-valor.

Modificación de valores en un diccionario

Para modificar un valor de un diccionario, indique el nombre del diccionario con la clave entre corchetes y, a continuación, el nuevo valor que desea asociar a esa clave. Por ejemplo, considere un alienígena que cambia de verde a amarillo a medida que avanza el juego:

```
alien_0 = {'color': 'verde'}
print("El alien es " + alien_0['color'] + ".")

alien_0['color'] = 'amarillo'
print("El alien es ahora " + alien_0['color'] + ".")
```

Primero definimos un diccionario para `alien_0` que contiene sólo el color del alien; luego cambiamos el valor asociado con la clave `'color'` a `'amarillo'`. La salida muestra que, efectivamente, el alienígena ha cambiado de verde a amarillo:

```
El alienígena es verde.
El alienígena es ahora amarillo.
```

Para un ejemplo más interesante, vamos a seguir la posición de un alienígena que puede moverse a diferentes velocidades. Almacenaremos un valor que represente la velocidad actual del alienígena y luego lo usaremos para determinar qué tan a la derecha debe moverse el alienígena:

```
alien_0 = {'posición_x': 0, 'y_position': 25, 'speed': 'medium'}
print("Original x-position: " + str(alien_0['posición_x']))

# Mueve al alien hacia la derecha.
# Determina qué tan lejos mover al alienígena basado en su velocidad actual.
❶ if alien_0['velocidad'] ==
    'lento': x_incremento = 1
elif alien_0['velocidad'] ==
    'media': x_incremento = 2
si no:
    # Este 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("Nueva posición x: " + str(alien_0['posición_x']))
```

Empezaremos definiendo un alienígena con una posición *x* y una posición *y* iniciales, y una velocidad `"media"`. Hemos omitido los valores de color y punto para simplificar, pero este ejemplo funcionaría igual si incluyeras también esos pares clave-valor. También imprimimos el valor original de `x_position` para ver cuánto se desplaza el alienígena hacia la derecha.

En ❶, una cadena `if-elif-else` determina cuánto debe moverse el alien hacia la derecha y almacena este valor en la variable `x_incremento`. Si la velocidad del alienígena es `'lenta'`, se mueve una unidad a la derecha; si la

velocidad es `'media'`, se mueve dos

unidades a la derecha; y si es `'rápido'`, se desplaza tres unidades a la derecha.

Una vez calculado el incremento, se suma al valor de `x_position` en ❷, y el resultado se almacena en `x_position` del diccionario.

Como se trata de un alienígena de velocidad media, su posición se desplaza dos unidades a la derecha:

```
Posición x original: 0
Nueva posición x: 2
```

Esta técnica es genial: cambiando un valor en el diccionario del alienígena, puedes cambiar el comportamiento general del alienígena. Por ejemplo, para convertir este alienígena de velocidad media en un alienígena rápido, se añadiría la línea:

```
alien_0['velocidad'] = rápido
```

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

Eliminación de pares clave-valor

Cuando ya no necesites una información almacenada en un diccionario, puedes utilizar la sentencia `del` para eliminar por completo un par clave-valor. Todo lo que `del` necesita es el nombre del diccionario y la clave que desea eliminar.

Por ejemplo, eliminemos la clave `"puntos"` del diccionario `alien_0` junto con su valor:

```
alien_0 = {'color': 'verde', 'puntos': 5}
print(alien_0)

❶ del alien_0['puntos']
print(alien_0)
```

La línea ❶ le dice a Python que borre la clave `'puntos'` del diccionario `alien_0` y que borre también el valor asociado a esa clave. La salida muestra que la clave `"puntos"` y su valor de `5` se eliminan del diccionario, pero el resto del diccionario no se ve afectado:

```
{'color': 'verde', 'puntos': 5}
{'color': 'verde'}
```

NOTE *Tenga en cuenta que el par clave-valor eliminado se elimina de forma permanente.*

Diccionario de objetos similares

En el ejemplo anterior se almacenaban distintos tipos de información sobre un objeto, un alienígena en un juego. También puedes utilizar un diccionario para almacenar un tipo de información sobre varios objetos.

Por ejemplo, digamos que quieres sondear un

número de personas y preguntarles cuál es su lenguaje de programación favorito. Un diccionario es útil para almacenar los resultados de una encuesta sencilla, como ésta:

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

Como puede ver, hemos dividido un diccionario más grande en varias líneas. Cada clave es el nombre de una persona que respondió a la encuesta, y cada valor es su elección de idioma. Cuando sepa que va a necesitar más de una línea para definir un diccionario, pulse INTRO después del corchete de apertura. A continuación, aplique una sangría de un nivel (cuatro espacios) a la línea siguiente y escriba el primer par clave-valor, seguido de una coma. A partir de este momento, cuando pulse INTRO, su editor de texto sangrará automáticamente todos los pares clave-valor subsiguientes para que coincidan con el primer par clave-valor.

Una vez que hayas terminado de definir el diccionario, añade una llave de cierre en una nueva línea después del último par clave-valor y haz una sangría de un nivel para que se alinee con las claves del diccionario. Es una buena práctica incluir también una coma después del último par clave-valor, de modo que esté listo para añadir un nuevo par clave-valor en la línea siguiente.

NOTE *La mayoría de los editores tienen alguna función que le ayuda a formatear listas y diccionarios extensos de manera similar a este ejemplo. También existen otras formas aceptables de formatear diccionarios largos, por lo que es posible que vea un formato ligeramente diferente en su editor o en otras fuentes.*

Para utilizar este diccionario, dado el nombre de una persona que haya participado en la encuesta, puedes buscar fácilmente su lengua favorita:

favorito_lenguajes.py

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

❶ print("El idioma favorito de Sarah es " +
❷     favorite_languages['sarah'].title() +
❸     ".")
```

Para ver qué idioma eligió Sarah, le pedimos el valor en:

```
favorite_languages['sarah']
```

Esta sintaxis se utiliza en la sentencia `print` en ❷, y la salida muestra el

idioma favorito de Sarah:

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

Este ejemplo también muestra cómo puede dividir una sentencia `print` larga en varias líneas. La palabra `print` es más corta que la mayoría de los nombres de diccionario, por lo que tiene sentido incluir la primera parte de lo que se desea imprimir justo después del paréntesis de apertura ❶. Elija un punto apropiado en el que romper lo que se está imprimiendo, y añada un operador de concatenación (`+`) al final de la primera línea ❷. Pulsa ENTER y luego pulsa TAB para alinear todas las líneas subsiguientes a un nivel de sangría bajo la sentencia de `impresión`. Cuando hayas terminado de componer tu salida, puedes colocar el paréntesis de cierre en la última línea del bloque de `impresión` ❸.

TRY IT yOURSELF

6-1. Persona: Utiliza un diccionario para almacenar información sobre una persona que conozcas. Almacena su nombre, apellido, edad y la ciudad en la que vive. Debes tener claves como `nombre`, `apellido`, `edad` y `ciudad`. Imprime cada dato almacenado en tu diccionario.

6-2. Números favoritos: Utiliza un diccionario para almacenar los números favoritos de la gente. Piensa en cinco nombres y úsalos como claves en tu diccionario. Piensa en un número favorito para cada persona y almacena cada uno como un valor en tu diccionario. Imprime el nombre de cada persona y su número favorito. Para divertirte aún más, haz una encuesta entre algunos amigos y obtén datos reales para tu programa.

6-3. Glosario: Un diccionario Python puede usarse para modelar un diccionario real. Sin embargo, para evitar confusiones, vamos a llamarlo glosario.

- Piensa en cinco palabras de programación que hayas aprendido en los capítulos anteriores. Utiliza estas palabras como claves de tu glosario y guarda sus significados como valores.
- Imprime cada palabra y su significado con un formato claro. Puede imprimir la palabra seguida de dos puntos y luego su significado, o imprimir la palabra en una línea y luego imprimir su significado sangrado en una segunda línea. Utilice el carácter de nueva línea (`\n`) para insertar una línea en blanco entre cada par palabra-significado en la salida.

Recorrer un diccionario

Un diccionario Python puede contener unos pocos pares clave-valor o millones de pares. Dado que un diccionario puede contener grandes cantidades de datos, Python permite realizar bucles a través de un diccionario. Los diccionarios pueden usarse para almacenar información d e varias maneras; por lo tanto, existen varias formas de recorrerlos. Puedes recorrer todos los pares clave-valor de un diccionario, sus claves o sus

valores.

Recorrer todos los pares clave-valor

Antes de explorar los distintos enfoques del bucle, consideremos un nuevo diccionario diseñado para almacenar información sobre un usuario en un sitio web. El siguiente diccionario almacenaría el nombre de usuario, el nombre y los apellidos de una persona:

```
usuario_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }
```

Puedes acceder a cualquier información sobre el `usuario_0` basándote en lo que ya has aprendido en este capítulo. ¿Pero qué pasaría si quisieras ver todo lo almacenado en el diccionario de este usuario? Para hacerlo, podrías recorrer el diccionario usando un bucle `for`:

usuario.py

```
usuario_0 = {
    'username': 'efermi',
    'first': 'enrico',
    'last': 'fermi',
    }

❶ para clave, valor en usuario_0.items():
❷     print("\nTecla: " + tecla)
❸     print("Valor: " + valor)
```

Como se muestra en ❶, para escribir un bucle `for` para un diccionario, crea nombres para las dos variables que contendrán la clave y el valor en cada par clave-valor. Puedes elegir los nombres que quieras para estas dos variables. Este código funcionaría igual de bien si hubieras utilizado abreviaturas para los nombres de las variables, así:

```
para k, v en usuario_0.items()
```

La segunda mitad de la sentencia `for` en ❶ incluye el nombre del diccionario seguido del método `items()`, que devuelve una lista de pares clave-valor. A continuación, el bucle `for` almacena cada uno de estos pares en las dos variables proporcionadas. En el ejemplo anterior, utilizamos las variables para imprimir cada `clave` ❷, seguida del `valor` asociado ❸. El `"\n"` en la primera sentencia de `impresión` asegura que se inserte una línea en blanco antes de cada par clave-valor en la salida:

```
Clave: last
Valor: fermi

Clave:
primero
Valor: enrico

Clave: nombre
```

```
de usuario
Valor: efermi
```

Observe de nuevo que los pares clave-valor no se devuelven en el orden en que fueron almacenados, incluso cuando se realiza un bucle a través de un diccionario. A Python no le importa el orden en que se almacenan los pares clave-valor; sólo rastrea las conexiones entre las claves individuales y sus valores.

El bucle a través de todos los pares clave-valor funciona particularmente bien para diccionarios como el ejemplo *favorite_languages.py* en la página 101, que almacena el mismo tipo de información para muchas claves diferentes. Si recorre el diccionario `favorite_languages`, obtendrá el nombre de cada persona en el diccionario y su lenguaje de programación favorito. Como las claves siempre se refieren al nombre de una persona y el valor siempre es un idioma, utilizaremos las variables `nombre` e `idioma` en el bucle en lugar de `clave` y `valor`. Así será más fácil seguir lo que ocurre dentro del bucle:

favorito_lenguajes.py

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

❶ para nombre, idioma en favorite_languages.items():
❷     print(nombre.title() + " el idioma favorito de es
          " + idioma.title() + ".")
```

El código de ❶ le dice a Python que recorra en bucle cada par clave-valor del diccionario. A medida que recorre cada par, la clave se almacena en la varian-
y el valor se almacena en el `lenguaje de` variables. Estos nombres descriptivos hacen mucho más fácil ver lo que está haciendo la sentencia `print` en ❷.

Ahora, con unas pocas líneas de código, podemos mostrar toda la información de la encuesta:

```
El lenguaje favorito de Jen es
Python. El lenguaje favorito de
Sarah es C. El lenguaje favorito de
Phil es Python. El lenguaje
favorito de Edward es Ruby.
```

Este tipo de bucle funcionaría igual de bien si nuestro diccionario almacenara los resultados de encuestar a mil o incluso a un millón de personas.

Recorrer en bucle todas las claves de un diccionario

El método `keys()` es útil cuando no necesita trabajar con todos los valores de un diccionario. Recorramos el diccionario `idiomas_favoritos` e imprimamos los nombres de todos los participantes en la encuesta:

```
favorite_languages = {
    'jen': 'python',
```

```
'sarah': 'c',
'edward': 'ruby',
'phil': 'python',
}
```

```
❶ for nombre en idiomas_favoritos.keys():
    print(nombre.titulo())
```

La línea ❶ indica a Python que extraiga todas las claves del diccionario `favorite_languages` y las almacene de una en una en la variable `name`. La salida muestra los nombres de todos los que participaron en la encuesta:

```
Jen
Sarah
Phil
Edward
```

El bucle a través de las claves es en realidad el comportamiento por defecto cuando se realiza un bucle a través de un diccionario, por lo que este código tendría exactamente la misma salida si se escribiera . . .

```
para nombre en idiomas_favoritos:
```

en lugar de. . .

```
para nombre en idiomas_favoritos.keys():
```

Puede elegir utilizar el método `keys()` explícitamente si hace que su código sea más fácil de leer, o puede omitirlo si lo desea.

Puedes acceder al valor asociado a cualquier clave que te interese dentro del bucle utilizando la clave actual. Imprimamos un mensaje a un par de amigos sobre los idiomas que han elegido. Recorreremos los nombres del diccionario como hicimos anteriormente, pero cuando el nombre coincida con uno de nuestros amigos, mostraremos un mensaje sobre su idioma favorito:

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

❶ amigos = ['phil', 'sarah']
  for nombre in idiomas_favoritos.claves():
      print(nombre.titulo())

❷     si nombre en amigos:
          print(" Hola " + nombre.titulo() +
                ", veo que tu idioma favorito es " +
❸             favorite_languages[name].title() + "!")
```

En ❶ hacemos una lista de amigos a los que queremos imprimir un mensaje. Dentro del bucle, imprimimos el nombre de cada persona. A continuación, en ❷ comprobamos si el `nombre` con el que estamos trabajando está en la lista de `amigos`. Si es así, imprimimos un saludo especial, incluyendo una referencia a su elección de idioma. Para acceder a

el idioma favorito de ❸, utilizamos el nombre del diccionario y el valor actual del `nombre` como clave. Se imprime el nombre de todos, pero nuestros amigos reciben un mensaje especial:

```
Edward
Phil
  Hola Phil, ¡veo que tu lenguaje favorito es
Python! Sarah
  Hola Sarah, ¡veo que tu lenguaje favorito es
C! Jen
```

También puedes utilizar el método `keys()` para averiguar si una persona en particular fue encuestada. En esta ocasión, vamos a averiguar si Erin participó en la encuesta:

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

❶ if 'erin' not in favorite_languages.keys():
    print("¡Erin, por favor, haz nuestra
    encuesta!")
```

El método `keys()` no es sólo para hacer bucles: En realidad devuelve una lista de todas las claves, y la línea en ❶ simplemente comprueba si `'erin'` está en esta lista. Si no está, se imprime un mensaje invitándola a participar en la encuesta:

```
Erin, ¡responde a nuestra encuesta!
```

Recorrer por orden las claves de un diccionario

Un diccionario siempre mantiene una conexión clara entre cada clave y su valor asociado, pero nunca se obtienen los elementos de un diccionario en un orden predecible. Eso no es un problema, porque normalmente sólo querrás obtener el valor correcto asociado a cada clave.

Una forma de devolver elementos en un orden determinado es ordenar las claves a medida que se devuelven en el bucle `for`. Puede utilizar la función `sorted()` para obtener una copia de las claves en orden:

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

```
    }

for nombre in ordenados(idiomas_favoritos.claves()):
    print(nombre.titulo() + ", gracias por participar en la
    encuesta.")
```

Esta sentencia `for` es como otras sentencias `for` excepto que hemos envuelto la función `sorted()` alrededor del método `dictionary.keys()`. Esto le dice a Python que liste todas las claves del diccionario y ordene la lista antes de recorrerla. La salida muestra a todos los participantes en la encuesta con los nombres en orden:

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

Recorrer todos los valores de un diccionario

Si está interesado principalmente en los valores que contiene un diccionario, puede utilizar el método `values()` para devolver una lista de valores sin ninguna clave. Por ejemplo, digamos que simplemente queremos una lista de todos los idiomas elegidos en nuestra encuesta de idiomas de pro- gramación sin el nombre de la persona que eligió cada idioma:

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

print("Se han mencionado los siguientes idiomas:") for
idioma in idiomas_favoritos.valores():
    print(idioma.título())
```

La sentencia `for` extrae cada valor del diccionario y lo almacena en la variable `idioma`. Cuando se imprimen estos valores, obtenemos una lista de todos los idiomas elegidos:

```
Se han mencionado las siguientes lenguas:
Python
C
Python
Rubí
```

Este método extrae todos los valores del diccionario sin comprobar si hay repeticiones. Esto puede funcionar bien con un pequeño número de valores, pero en una encuesta con un gran número de encuestados, esto daría lugar a una lista muy repetitiva. Para ver cada idioma elegido sin repeticiones, podemos utilizar un *conjunto*.
Un conjunto es similar a una lista, excepto en que cada elemento del conjunto debe ser único:

```
favorite_languages = {
    'jen': 'python',
```

```
'sarah': 'c',
'edward': 'ruby',
```

```
    'phil': 'python',
    }

print("Se han mencionado las siguientes lenguas:")
❶ for idioma in set(idiomas_favoritos.valores()):
    print(idioma.titulo())
```

Cuando se envuelve set() alrededor de una lista que contiene elementos duplicados, Python identifica los elementos únicos de la lista y construye un conjunto a partir de esos elementos. En ❶ usamos set() para sacar los idiomas únicos en `favorite_languages.values()`.

El resultado es una lista no repetitiva de las lenguas mencionadas por los participantes en la encuesta:

```
Se han mencionado las siguientes lenguas:
Python
C
Ruby
```

A medida que vayas aprendiendo Python, a menudo encontrarás una función integrada en el lenguaje que te ayudará a hacer exactamente lo que quieres con tus datos.

TRY IT yOURSELF

6-4. Glosario 2: Ahora que sabe cómo hacer un bucle a través de un diccionario, limpie el código del Ejercicio 6-3 (página 102) reemplazando su serie de sentencias `print` por un bucle que recorra las claves y valores del diccionario.
Cuando estés seguro de que tu bucle funciona, añade cinco términos más de Python a tu glosario. Cuando vuelvas a ejecutar tu programa, estas nuevas palabras y significados deberían incluirse automáticamente en la salida.

6-5. Ríos: Crea un diccionario que contenga tres ríos principales y el país que atraviesa cada río. Un par clave-valor podría ser `'nilo': "egipto"`.

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

6-6. Sondeo: Usa el código en *favorite_languages.py* (página 104).

- Haz una lista de las personas que deberían participar en la encuesta sobre lenguas favoritas. Incluye algunos nombres que ya estén en el diccionario y otros que no.
- Recorre la lista de personas que deben responder a la encuesta. Si ya han participado en la encuesta, imprime un mensaje agradeciéndoles su respuesta. Si aún no han respondido, imprime un mensaje invitándoles a hacerlo.

Nido

A veces querrá almacenar un conjunto de diccionarios en una lista o una lista de elementos como valor en un diccionario. Esto se denomina *anidamiento*. Puede anidar un conjunto de diccionarios dentro de una lista, una lista de ítems dentro de un diccionario, o incluso un dentro de otro diccionario. El anidamiento es una potente función, como demostrarán los siguientes ejemplos.

Lista de diccionarios

El diccionario `alien_0` contiene información variada sobre un alienígena, pero no tiene espacio para almacenar información sobre un segundo alienígena, y mucho menos sobre una pantalla llena de alienígenas. ¿Cómo se puede gestionar una flota de alienígenas? Una forma es hacer una lista de alienígenas en la que cada alienígena es un diccionario de información sobre ese alienígena. Por ejemplo, el siguiente código construye una lista de tres alienígenas:

aliens.py

```
alien_0 = {'color': 'verde', 'puntos': 5}
alien_1 = {'color': 'amarillo', 'puntos': 10}
alien_2 = {'color': 'rojo', 'puntos': 15}

❶ aliens = [alien_0, alien_1, alien_2]

for alien in aliens:
    print(extranjero)
```

Primero creamos tres diccionarios, cada uno representando un alienígena diferente. En ❶ empaquetamos cada uno de estos diccionarios en una lista llamada `aliens`. Finalmente, recorremos la lista e imprimimos cada alienígena:

```
{'color': 'verde', 'puntos': 5}
{'color': 'amarillo', 'puntos': 10}
{'color': 'rojo', 'puntos': 15}
```

Un ejemplo más realista implicaría más de tres alienígenas con código que genere automáticamente cada alienígena. En el siguiente ejemplo utilizamos `range()` para crear una flota de 30 alienígenas:

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

# Haz 30 alienígenas verdes.
❶ para numero_extranjero en rango(30):
❷     new_alien = {'color': 'verde', 'puntos': 5, 'velocidad': 'lento'}
❸     aliens.append(nuevo_alien)

# Mostrar los primeros 5 alienígenas:
❹ for alien in aliens[:5]:
    print(alien)
```

```
print("...")

# Muestra cuántos alienígenas se han creado.
❺ print("Número total de extranjeros: " + str(len(extranjeros)))
```

Este ejemplo comienza con una lista vacía para contener todos los alienígenas que se crearán. En ❶ `range()` devuelve un conjunto de números, que sólo indica a Python cuántas veces queremos que se repita el bucle. Cada vez que el bucle se ejecuta creamos un nuevo alien ❷ y luego añadimos cada nuevo alien a la lista `aliens` ❸. En ❹ usamos un slice para imprimir los cinco primeros alienígenas, y luego en ❺ imprimimos la longitud de la lista para probar que realmente hemos generado la flota completa de 30 alienígenas:

```
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
...

Número total de extranjeros: 30
```

Todos estos alienígenas tienen las mismas características, pero Python considera a cada uno un objeto independiente, lo que nos permite modificar cada alienígena individualmente.

¿Cómo podría trabajar con un conjunto de alienígenas como éste? Imagina que un aspecto de un juego tiene algunos alienígenas que cambian de color y se mueven más rápido a medida que avanza el juego. Cuando llegue el momento de cambiar los colores, podemos utilizar un bucle `for` y una sentencia `if` para cambiar el color de los alienígenas. Por ejemplo, para cambiar los tres primeros aliens a amarillo, aliens de velocidad media que valen 10 puntos cada uno, podríamos hacer esto:

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

# Haz 30 alienígenas verdes.
para numero_extranjero en rango (0,30):
    nuevo_alien = {'color': 'verde', 'puntos': 5, 'velocidad': 'lento'}
    aliens.append(nuevo_alien)

para alien en aliens[0:3]:
    if alien['color'] == 'green':
        alien['color'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

# Mostrar los primeros 5 alienígenas:
for alien in aliens[0:5]:
    print(alien)
print("...")
```

Como queremos modificar los tres primeros alienígenas, hacemos un bucle a través de una porción que incluye sólo los tres primeros alienígenas. Ahora todos los alienígenas son verdes, pero no siempre será así, así que

escribimos una sentencia `if` para asegurarnos de que

sólo modificamos los alienígenas verdes. Si el alienígena es verde, cambiamos el color a `'amarillo'`, la velocidad a `'media'`, y el valor de puntos a `10`, como se muestra en la siguiente salida:

```
{'velocidad': 'media', 'color': 'amarillo', 'puntos': 10}
{'velocidad': 'media', 'color': 'amarillo', 'puntos': 10}
{'velocidad': 'media', 'color': 'amarillo', 'puntos': 10}
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
{'velocidad': 'lento', 'color': 'verde', 'puntos': 5}
...
```

Podrías ampliar este bucle añadiendo un bloque `elif` que convierta a los alienígenas amarillos en rojos y rápidos que valen 15 puntos cada uno. Sin mostrar todo el programa de nuevo, ese bucle se vería así:

```
para alien en 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 común almacenar varios diccionarios en una lista cuando cada diccionario contiene muchos tipos de información sobre un objeto. Por ejemplo, puedes crear un diccionario para cada usuario de un sitio web, como hicimos en *usuario.py* en la página 103, y almacenar los diccionarios individuales en una lista llamada `usuarios`. Todos los diccionarios de la lista deben tener una estructura idéntica para que puedas recorrer la lista y trabajar con cada objeto diccionario de la misma manera.

Una lista en un diccionario

En lugar de poner un diccionario dentro de una lista, a veces es útil poner una lista dentro de un diccionario. Por ejemplo, piensa en cómo describir una pizza que alguien va a pedir. Si sólo utilizaras una lista, lo único que podrías almacenar sería una lista de los ingredientes de la pizza. Con un diccionario, la lista de ingredientes puede ser sólo un aspecto de la pizza que estás describiendo.

En el siguiente ejemplo, se almacenan dos tipos de información para cada pizza: un tipo de masa y una lista de ingredientes. La lista de ingredientes es un valor asociado a la clave `"ingredientes"`. Para utilizar los elementos de la lista, le damos el nombre del diccionario y la clave "ingredientes", como haríamos con cualquier valor del diccionario. En lugar de devolver un único valor, obtenemos una lista de coberturas:

pizza.py

```
# Almacena información sobre la pizza que se pide.
❶ pizza = {
    'corteza': 'grueso',
    'toppings': ['champiñones', 'queso extra'],
```

```
}
```

```
    # Resumir la orden.
❷ print("Ha pedido una " + pizza['corteza'] + "pizza de corteza "
      + "con los siguientes ingredientes:")

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

Comenzamos en ❶ con un diccionario que contiene información sobre una pizza que se ha pedido. Una de las claves del diccionario es `"corteza"` y el valor asociado es la cadena `"gruesa"`. La siguiente clave, `"ingredientes"`, tiene como valor una lista que almacena todos los ingredientes solicitados. En ❷ resumimos el pedido antes de construir la pizza. Para imprimir los toppings, escribimos un bucle `for` ❸. Para acceder a la lista de toppings, usamos la clave `'toppings'`, y Python coge la lista de toppings del diccionario.

La siguiente salida resume la pizza que planeamos construir:

```
Has pedido una pizza de masa gruesa con los siguientes
    ingredientes: champiñones
    queso extra
```

Puede anidar una lista dentro de un diccionario siempre que desee asociar más de un valor a una única clave de un diccionario. En el ejemplo anterior de los lenguajes de programación favoritos, si almacenáramos las respuestas de cada persona en una lista, las personas podrían elegir más de un lenguaje favorito. Al recorrer el diccionario, el valor asociado a cada persona sería una lista de idiomas en lugar de un único idioma. Dentro del bucle `for del diccionario`, utilizamos otro bucle `for` para recorrer la lista de idiomas asociados a cada persona:

favoritos_idiomas.py

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

❷ para nombre, idiomas en favorite_languages.items():
      print("\n" + nombre.title() + " idiomas favoritos de:")
❸     para la lengua en las lenguas:
          print("\t" + language.title())
```

Como puedes ver en ❶ el valor asociado a cada nombre es ahora una lista. Observa que algunas personas tienen un idioma favorito y otras tienen varios favoritos. Cuando hacemos un bucle a través del diccionario en ❷, utilizamos la variable nombre `idiomas` para mantener cada valor del diccionario, porque sabemos que cada valor será una lista. Dentro del bucle principal del diccionario, utilizamos

otro bucle `for` ❸ para recorrer la lista de idiomas favoritos de cada persona. Ahora cada persona puede enumerar tantos idiomas favoritos como quiera:

```
Los lenguajes favoritos de Jen
    son: Python
    Ruby

Los idiomas favoritos de Sarah
    son: C

Los lenguajes favoritos de Phil
    son: Python
    Haskell

Los lenguajes favoritos de Edward
    son: Ruby
    Vaya a
```

Para refinar aún más este programa, podría incluir una sentencia `if` al principio del bucle `for` del diccionario para ver si cada persona tiene más de un idioma favorito examinando el valor de `len(languages)`. Si una persona tiene más de un idioma favorito, la salida sería la misma. Si la persona sólo tiene un idioma favorito, podría cambiar la redacción para reflejarlo. Por ejemplo, podría decir que el `lenguaje favorito de Sarah es C`.

NOTE *No debes anidar listas y diccionarios demasiado profundamente. Si estás anidando elementos mucho más profundamente de lo que se ve en los ejemplos anteriores o estás trabajando con el código de otra persona con niveles significativos de anidamiento, lo más probable es que exista una forma más sencilla de resolver el problema.*

Un diccionario en un diccionario

Puedes anidar un diccionario dentro de otro diccionario, pero tu código puede complicarse rápidamente cuando lo haces. Por ejemplo, si tiene varios usuarios para un sitio web, cada uno con un nombre de usuario único, puede utilizar los nombres de usuario como claves en un diccionario. A continuación, puede almacenar información sobre cada usuario utilizando un diccionario como valor asociado a su nombre de usuario. En la siguiente lista, almacenamos tres datos sobre cada usuario: su nombre de usuario y su contraseña.
nombre, apellidos y ubicación. Accederemos a esta información recorriendo los nombres de usuario y el diccionario de información asociado a cada nombre de usuario:

muchos_usuarios.py

```
usuarios = {
'aeinstein': {'first':
    'albert',
    'apellido': 'einstein',
    'ubicación': 'princeton',
    },
```

```
        'mcurie': {
            'primero': 'marie',
            'último': 'curie',
            ubicación": "parís",
            },

        }

❶ para nombre_usuario, usuario_info en usuarios.items():
❷     print("Nombre de usuario: " + nombre de usuario)
❸     full_name = user_info['first'] + " " + user_info['last']
      location = user_info['location']

❹     print("\tNombre completo: " +
      nombre_completo.title())
      print("\tLocalización: " +
      lugar.titulo())
```

Primero definimos un diccionario llamado `usuarios` con dos claves: una para cada uno de los nombres de usuario `'aeinstein'` y `'mcurie'`. El valor asociado a cada clave es un diccionario que incluye el nombre, el apellido y la ubicación de cada usuario. En ❶ hacemos un bucle a través del diccionario de `usuarios`. Python almacena cada clave en la variable `nombre_usuario`, y el diccionario asociado a cada nombre de usuario va a la variable `user_info`. Una vez dentro del bucle principal del diccionario, imprimimos el nombre de usuario en ❷.

En ❸ empezamos a acceder al diccionario interno. 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 con formato ordenado y la ubicación de cada persona, y luego imprimimos un resumen de lo que sabemos de cada usuario ❹:

```
Nombre de usuario: aeinstein
    Nombre completo: Albert
    Einstein Lugar: Princeton

Nombre de usuario: mcurie
    Nombre completo: Marie Curie
    Ubicación: París
```

Observe que la estructura del diccionario de cada usuario es idéntica. Aunque no es requerido por Python, esta estructura hace que los diccionarios anidados sean más fáciles de trabajar. Si cada diccionario de usuario tuviera claves diferentes, el código dentro del bucle `for` sería más complicado.

TRY IT yOURSELF

6-7. Personas: Comience con el programa que escribió para el Ejercicio 6-1 (página 102). Crea dos nuevos diccionarios que representen diferentes personas, y almacena los tres diccionarios en una lista llamada `personas`. Haz un bucle a través de tu lista de personas. A medida que recorre la lista, imprima todo lo que sabe sobre cada persona.

6-8. Mascotas: Haz varios diccionarios, donde el nombre de cada diccionario sea el nombre de una mascota. En cada diccionario, incluye el tipo de animal y el nombre del dueño. Guarda estos diccionarios en una lista llamada `mascotas`. A continuación, recorre la lista y, mientras lo haces, imprime todo lo que sepas sobre cada mascota.

6-9. Lugares favoritos: Crea un diccionario llamado `lugares_favoritos`. Piensa en tres nombres para utilizarlos como claves en el diccionario, y almacena de uno a tres lugares favoritos para cada persona. Para hacer este ejercicio un poco más interesante, pide a algunos amigos que nombren algunos de sus lugares favoritos. Recorre el diccionario e imprime el nombre de cada persona y sus lugares favoritos.

6-10. Números Favoritos: Modifica tu programa del Ejercicio 6-2 (página 102) para que cada persona pueda tener más de un número favorito. Luego imprime el nombre de cada persona junto con sus números favoritos.

6-11. Ciudades: Haz un diccionario llamado `ciudades`. Utiliza los nombres de tres ciudades como claves del diccionario. Crea un diccionario de información sobre cada ciudad e incluye el país en el que se encuentra, su población aproximada y un dato sobre esa ciudad. Las claves del diccionario de cada ciudad deben ser algo así como `país`, `población` y `dato`. Imprime el nombre de cada ciudad y toda la información que has almacenado sobre ella.

6-12. Ampliaciones: Ahora estamos trabajando con ejemplos que son lo suficientemente complejos como para que puedan ser extendidos de muchas maneras. Utilice uno de los programas de ejemplo de este capítulo y amplíelo añadiendo nuevas claves y valores, cambiando el contexto del programa o mejorando el formato de la salida.

Resumen

En este capítulo aprendió a definir un diccionario y a trabajar con la información almacenada en un diccionario. Aprendiste cómo acceder y modificar elementos individuales en un diccionario, y cómo recorrer toda la información en un diccionario. Aprendió a recorrer los elementos pares clave-valor, sus claves y sus valores. También ha aprendido a anidar varios diccionarios en una lista, a anidar listas en un diccionario y a anidar un diccionario dentro de un diccionario.
un diccionario.

En el próximo capítulo aprenderás sobre bucles `while` y cómo aceptar entradas de personas que estén usando tus programas. Este será un capítulo emocionante, porque aprenderás a hacer que todos tus programas sean interactivos: serán capaces de responder a las entradas del usuario.

7

USE R IN P U T Y WH IL E LOOOPS



La mayoría de los programas se escriben para resolver un problema del usuario final. Para ello, suelen necesitar obtener cierta información del usuario. Por ejemplo

Un ejemplo sencillo: supongamos que alguien quiere saber si tiene edad suficiente para votar. Si escribe un

para responder a esta pregunta, necesita conocer la edad del usuario antes de poder proporcionar una respuesta. El programa tendrá que pedir al usuario que introduzca, o *introduzca*, su edad; una vez que el programa tiene esta entrada, puede compararla con la edad de votación para determinar si el usuario es lo suficientemente mayor y, a continuación, informar del resultado.

En este capítulo aprenderás a aceptar entradas del usuario para que tu programa pueda trabajar con ellas. Cuando tu programa necesite un nombre, podrás pedírselo al usuario. Cuando tu programa necesite una lista de nombres,
podrás pedir al usuario una serie de nombres. Para ello, utilizarás la función `input()`.

También aprenderás a mantener los programas en ejecución tanto tiempo como los usuarios quieran, para que puedan introducir tanta

información como necesiten; entonces, tu programa podrá trabajar con esa información. Utilizarás el bucle `while` de Python para mantener los programas en ejecución mientras se cumplan ciertas condiciones.

www.allitebooks.com

Gracias a la posibilidad de trabajar con entradas de usuario y de controlar el tiempo de ejecución de los programas, podrá escribir programas totalmente interactivos.

Cómo funciona la función input()

La función `input()` pausa tu programa y espera a que el usuario introduzca algún texto. Una vez que Python recibe la entrada del usuario, la almacena en una variable para que te resulte más cómodo trabajar con ella.

Por ejemplo, el siguiente programa pide al usuario que introduzca un texto y, a continuación, le devuelve el mensaje:

loro.py

```
message = input("Dime algo y te lo repetiré: ") print(message)
```

La función `input()` toma un argumento: el *prompt*, o instrucciones, que queremos mostrar al usuario para que sepa qué hacer. En este ejemplo, cuando Python ejecuta la primera línea, el usuario ve el prompt `Dime algo, y te lo repetiré:` . El programa espera mientras el usuario introduce su respuesta y continúa después de que el usuario pulse ENTER. La respuesta se almacena en la variable `mensaje`, luego `print(mensaje)` muestra la entrada al usuario:

```
Dime algo y te lo repetiré: ¡Hola a todos!
Hola a todos.
```

NOTA *Sublime Text no ejecuta programas que pidan entrada al usuario. Puedes usar Sublime Text para escribir programas que pregunten al usuario, pero necesitarás ejecutar estos programas desde un terminal. Mira "Ejecutar programas Python desde un terminal" en la página 16.*

Escribir con claridad

Cada vez que utilice la función `input()`, debe incluir un prompt claro y fácil de seguir que indique al usuario exactamente qué tipo de información está buscando. Cualquier sentencia que le diga al usuario qué introducir debería funcionar. Por ejemplo:

saludo.py

```
nombre = input("Por favor, introduzca su
nombre: ") print("¡Hola, " + nombre +
"!")
```

Añada un espacio al final de las instrucciones (después de los dos puntos en el ejemplo anterior) para separar la instrucción de la respuesta del usuario y dejarle claro dónde debe introducir el texto. Por ejemplo:

```
Escribe tu nombre: Eric
¡Hola, Eric!
```

A veces querrás escribir un prompt más largo que una línea. Por ejemplo, es posible que desee decirle al usuario por qué está pidiendo cierta entrada.

Puedes almacenar tu prompt en una variable y pasar esa variable a la función input(). Esto le permite construir su prompt en varias líneas, y luego escribir una sentencia input() limpia.

greeter.py

```
prompt = "Si nos dice quién es, podemos personalizar los mensajes que ve."
prompt += "\n¿Cuál es su nombre de pila? "

nombre = input(prompt)
print("\n¡Hola, " + nombre +
"!")
```

Este ejemplo muestra una forma de construir una cadena multilínea. La primera línea almacena la primera parte del mensaje en la variable `prompt`. En la segunda línea, el operador `+=` toma la cadena almacenada en `prompt` y añade la nueva cadena al final.

La pregunta ahora ocupa dos líneas, con un espacio después del signo de interrogación para mayor claridad:

```
Si nos dices quién eres, podremos personalizar los mensajes que
veas. ¿Cuál es tu nombre de pila? Eric

¡Hola, Eric!
```

Uso de int() para aceptar entradas numéricas

Cuando usas la función `input()`, Python interpreta todo lo que el usuario introduce como una cadena. Considere la siguiente sesión del intérprete, que pregunta por la edad del usuario:

```
>>> edad = input("¿Cuántos años tienes? ")
¿Cuántos años tienes? 21
>>> edad
'21'
```

El usuario introduce el número 21, pero cuando pedimos a Python el valor de la edad, nos devuelve `'21'`, la representación en cadena del valor numérico introducido. Sabemos que Python interpretó la entrada como una cadena porque el número está ahora entre comillas. Si todo lo que quieres hacer es imprimir la entrada, esto funciona bien. Pero si intentas usar la entrada como un número, obtendrás un error:

```
>>> edad = input("¿Cuántos años tienes? ")
¿Cuántos años tienes? 21
❶ >>> edad >= 18 años
Traceback (última llamada más reciente):
  File "<stdin>", line 1, in <module>.
❷ TypeError: tipos desordenables: str() >= int()
```

Cuando intentas usar la entrada para hacer una comparación numérica ❶, Python produce un error porque no puede comparar una cadena con un

entero: la cadena `'21'` que está almacenada en `age` no puede compararse con el valor numérico `18` ❷.

Podemos resolver este problema utilizando la función `int()`, que indica a Python que debe tratar la entrada como un valor numérico. La función `int()` convierte una representación de cadena de un número en una representación numérica, como se muestra aquí:

```
>>> edad = input("¿Cuántos años tienes? ")
¿Cuántos años tienes? 21
❶ >>> edad = int(edad)
>>> edad >= 18
Verdadero
```

En este ejemplo, cuando introducimos `21` en el prompt, Python interpreta el número como una cadena, pero el valor se convierte a una representación numérica mediante `int()` ❶. Ahora Python puede ejecutar la prueba condicional: compara la `edad` (que ahora contiene 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 utiliza la función `int()` en un programa real? Considere un programa que determina si las personas son lo suficientemente altas como para montar en una montaña rusa:

montaña rusa.py

```
altura = input("¿Cuánto mides, en pulgadas?")
altura = int(altura)

si altura >= 36:
    print("\n¡Eres lo suficientemente alto
para montar!") else:
    print("\nPodrás montar cuando seas un poco mayor.")
```

El programa puede comparar la `altura` con `36` porque `altura = int(altura)` convierte el valor introducido en una representación numérica antes de realizar la comparación. Si el número introducido es mayor o igual que 36, le decimos al usuario que es lo suficientemente alto:

```
¿Cuánto mide, en centímetros? 71

¡Eres lo suficientemente alto para montar!
```

Cuando utilice entradas numéricas para realizar cálculos y comparaciones, asegúrese de convertir primero el valor de entrada a una representación numérica.

El operador módulo

Una herramienta útil para trabajar con información numérica es el *operador módulo* (`%`), que divide un número por otro y devuelve el resto:

```
>>> 4 % 3
1
>>> 5 % 3
```

```
2
>>> 6 % 3
0
```

```
>>> 7 % 3
1
```

El operador módulo no te dice cuántas veces cabe un número dentro de otro; sólo te dice cuál es el resto.

Cuando un número es divisible por otro número, el resto es 0, por lo que el operador módulo siempre devuelve 0. Puedes utilizar este hecho para determinar si un número es par o impar:

par_o_impar.py

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

si número % 2 == 0:
    print("\nEl número " + str(número) + " es par.")
else:
    print("El número " + str(número) + " es impar.")
```

Los números pares siempre son divisibles por dos, por lo que si el módulo de un número y dos es cero (aquí, `si número % 2 == 0`) el número es par. En caso contrario, es impar.

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

El número 42 es par.
```

Aceptar entradas en Python 2.7

Si estás usando Python 2.7, deberías usar la función `raw_input()` cuando pidas información al usuario. Esta función interpreta toda la entrada como una cadena, igual que hace `input()` en Python 3.

Python 2.7 también tiene una función `input()`, pero esta función interpreta la entrada del usuario como código Python e intenta ejecutar la entrada. En el mejor de los casos obtendrás un error de que Python no entiende la entrada; en el peor, ejecutarás código que no tenías intención de ejecutar. Si estás usando Python 2.7, usa `raw_input()` en lugar de `input()`.

TRY IT yOURSELF

7-1. Coche de alquiler: Escriba un programa que pregunte al usuario qué tipo de coche de alquiler le gustaría. Imprima un mensaje sobre ese coche, como por ejemplo "Déjeme ver si puedo encontrarle un Subaru".

7-2. Asientos en un restaurante: Escriba un programa que pregunte al usuario cuántas personas hay en su grupo para cenar. Si la respuesta es más de ocho, imprima un mensaje diciendo que tendrán que esperar por una mesa. De lo contrario, informe que su mesa está lista.

7-3. Múltiplos de diez: Pregunta al usuario por un número, y luego informa si el número es múltiplo de 10 o no.

Introducción a los bucles while

El bucle `for` toma una colección de elementos y ejecuta un bloque de código una vez por cada elemento de la colección. Por el contrario, el bucle `while` se ejecuta *mientras se* cumpla una determinada condición.

El bucle while en acción

Puede utilizar un bucle `while` para contar una serie de números. Por ejemplo, el siguiente bucle `while` cuenta del 1 al 5:

contando.py

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

En la primera línea, empezamos a contar desde 1 estableciendo el valor de `número_actual` en 1. A continuación, el bucle `while` se configura para que siga ejecutándose mientras el valor de `número_actual` sea menor o igual que 5. El código dentro del bucle imprime el valor de `número_actual` y luego añade 1 a ese `valor` con `número_actual += 1`. (El operador `+=` es la abreviatura de `número_actual = número_actual + 1`.)

Python repite el bucle mientras la condición `número_actual <= 5` sea cierta. Como 1 es menor que 5, Python imprime `1` y luego añade 1, haciendo que el número actual sea 2. Como 2 es menor que 5, Python imprime `2` y añade 1 de nuevo, haciendo que el número actual sea 3, y así sucesivamente. Una vez que el valor de `número_actual` es mayor que 5, el bucle deja de ejecutarse y el programa termina:

```
1
2
3
4
5
```

Lo más probable es que los programas que utilizas a diario contengan `bucles while`. Por ejemplo, un juego necesita un bucle `while` para seguir ejecutándose mientras quieras seguir jugando, y para que pueda dejar de ejecutarse en cuanto le pidas que salga. Los programas no serían divertidos si dejaran de ejecutarse antes de que se lo pidiéramos o siguieran ejecutándose incluso después de que quisiéramos salir, así que los bucles `while` son muy útiles.

Dejar que el usuario elija cuándo abandonar

Podemos hacer que el programa *parrot.py* se ejecute tanto como el usuario quiera poniendo la mayor parte del programa dentro de un bucle `while`. Vamos a definir un *valor de salir* y luego mantener el programa en ejecución, siempre y cuando el usuario no ha introducido el valor de salir:

parrot.py

```
❶ prompt = "\nDime algo, y yo te lo repetiré:" prompt +=
```

```
                "\nIntroduzca 'quit' para finalizar el programa. "
```

```
❷ mensaje = ""
❸ while mensaje != 'quit':
      mensaje = input(prompt)
      print(mensaje)
```

En ❶, definimos un prompt que indica al usuario sus dos opciones: introducir un mensaje o introducir el valor de abandono (en este caso, `'quit'`). Luego establecemos una variable `mensaje` ❷ para almacenar cualquier valor que introduzca el usuario. Definimos `mensaje` como una cadena vacía, `""`, para que Python tenga algo que comprobar la primera vez que llegue a la línea `while`. La primera vez que el programa se ejecuta y Python llega a la sentencia while, necesita comparar el valor de `message` con `'quit'`, pero todavía no se ha introducido ninguna entrada de usuario. Si Python no tiene nada que comparar, no podrá continuar ejecutando el programa. Para resolver este problema, nos aseguramos de darle a `mensaje` un valor inicial. Aunque es sólo una cadena vacía, tendrá sentido para Python y le permitirá realizar la comparación que hace que el bucle `while` funcione. Este bucle `while` ❸ se ejecuta mientras el valor de `mensaje` no sea `'quit'`.

La primera vez que se pasa por el bucle, el `mensaje` es una cadena vacía, por lo que Python entra en el bucle. En `message = input(prompt)`, Python muestra el prompt y espera a que el usuario introduzca sus datos. Lo que introduzca se almacena en el `mensaje` y se imprime; entonces, Python reevalúa la condición en la sentencia `while`. Mientras el usuario no haya introducido la palabra `'quit'`, el prompt se muestra de nuevo y Python espera más datos. Cuando el usuario finalmente introduce 'quit', Python deja de ejecutar el bucle `while` y el programa termina:

```
Dime algo y te lo repetiré: Introduce 'quit' para
terminar el programa. Hola a todos.
Hola a todos.

Dime algo y te lo repetiré: Introduce 'quit' para
terminar el programa. Hola de nuevo.
Hola de nuevo.

Dime algo y te lo repetiré:
Introduce 'quit' para finalizar el programa. quit
abandone
```

Este programa funciona bien, excepto que imprime la palabra `'quit'` como si fuera un mensaje real. Una simple prueba `if` arregla esto:

```
prompt = "\nDígame algo, y yo se lo repetiré:" prompt +=
"\nIntroduzca 'quit' para finalizar el programa. "

mensaje = ""
while mensaje != 'quit':
    mensaje = input(prompt)

    if mensaje != 'quit':
        print(mensaje)
```

Ahora el programa hace una comprobación rápida antes de mostrar el mensaje y sólo imprime el mensaje si no coincide con el valor de salida:

```
Dime algo y te lo repetiré:
Introduce 'quit' para finalizar el programa. Hola a todos
Hola a todos.

Dime algo y te lo repetiré:
Introduce 'quit' para finalizar el programa. Hola de nuevo.
Hola de nuevo.

Dime algo y te lo repetiré:
Introduce 'quit' para finalizar el programa. quit
```

Utilizar una bandera

En el ejemplo anterior, hicimos que el programa realizara ciertas tareas mientras se cumplía una determinada condición. Pero ¿qué pasa con los programas más complicados en los que muchos eventos diferentes podrían hacer que el programa dejara de ejecutarse?

Por ejemplo, en un juego, varios acontecimientos diferentes pueden poner fin a la partida. Cuando el jugador se queda sin naves, se le acaba el tiempo o todas las ciudades que debía proteger son destruidas, el juego debería terminar. Tiene que terminar si ocurre cualquiera de estos eventos. Si muchos eventos posibles pueden ocurrir para detener el programa, tratar de probar todas estas condiciones en una sentencia while se vuelve complicado y difícil.

Para un programa que debe ejecutarse sólo mientras muchas condiciones sean verdaderas, puede 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 nuestros programas de forma que se ejecuten mientras la bandera esté en `True` y dejen de ejecutarse cuando alguno de los eventos cambie el valor de la bandera a `False`. Como resultado, nuestra sentencia `while` sólo necesita comprobar una condición: si la bandera es `True` o no. Entonces, todas nuestras otras pruebas (para ver si ha ocurrido un evento que debería poner la bandera en `Falso`) pueden organizarse ordenadamente en el resto del programa.

Vamos a añadir una bandera a *parrot.py* de la sección anterior. Esta bandera, que llamaremos `activa` (aunque puedes llamarla como quieras), controlará si el programa debe continuar ejecutándose o no:

```
prompt = "\nDígame algo, y yo se lo repetiré:" prompt +=
"\nIntroduzca 'quit' para finalizar el programa. "

❶ activo = Verdadero
❷ mientras está activo:
      mensaje = input(prompt)

❸     si mensaje == 'quit':
          activo = Falso
❹     else:
```

```
print(mensaje)
```

Establecemos la variable `active` a `True` ❶ para que el programa comience en un estado activo. Esto hace que la sentencia `while` sea más sencilla porque no se realiza ninguna comparación en la propia sentencia `while`; la lógica se realiza en otras partes del programa. Mientras la variable `activa` siga siendo `True`, el bucle continuará ejecutándose ❷.

En la sentencia `if` dentro del bucle `while`, comprobamos el valor de `message` una vez que el usuario introduce su entrada. Si el usuario introduce `'quit'` ❸, establecemos `active` a `False`, y el bucle `while` se detiene. Si el usuario introduce cualquier otra cosa que no sea `'quit'` ❹, imprimimos su entrada como un mensaje.

Este programa tiene la misma salida que el ejemplo anterior donde colocamos la prueba condicional directamente en la sentencia `while`. Pero ahora que tenemos una bandera para indicar si el programa en general está en un estado activo, sería fácil añadir más pruebas (como sentencias `elif`) para eventos que deberían causar que `active` se convierta en `False`. Esto es útil en programas complicados como juegos en los que puede haber muchos eventos que deberían hacer que el programa dejara de ejecutarse. Cuando cualquiera de estos eventos cause que la bandera active se convierta en `False`, el bucle principal del juego saldrá, un mensaje de *Game Over* puede ser mostrado, y al jugador se le puede dar la opción de jugar de nuevo.

Uso de break para salir de un bucle

Para salir de un bucle `while` inmediatamente sin ejecutar ningún código restante en el bucle, independientemente de los resultados de cualquier prueba condicional, utilice la sentencia `break`. La sentencia `break` dirige el flujo de su programa; puede usarla para controlar qué líneas de código se ejecutan y cuáles no, de forma que el programa sólo ejecute el código que usted quiera, cuando usted quiera.

Por ejemplo, consideremos un programa que pregunta al usuario sobre los lugares que ha visitado. Podemos detener el bucle `while` de este programa llamando a `break` en cuanto el usuario introduzca el valor `'quit'`:

ciudades.py

```
prompt = "\nPor favor, introduzca el nombre de una ciudad que
haya visitado:" prompt += "\n(Introduzca 'quit' cuando haya
terminado.) "

❶ mientras True:
    ciudad = input(prompt)

    if ciudad == 'quit':
        break
    si no:
        print("¡Me encantaría ir a " + city.title() + "!")
```

Un bucle que comienza con `while True` ❶ se ejecutará para siempre a menos que llegue a una sentencia `break`. El bucle de este programa sigue pidiendo al usuario que introduzca los nombres de las ciudades en las que ha estado hasta que introduzca `'quit'`. Cuando introducen 'quit', se ejecuta la

sentencia break, haciendo que Python salga del bucle:

```
Introduce el nombre de una ciudad que hayas
visitado: (Introduce "abandonar" cuando hayas
terminado.) Nueva York ¡Me encantaría ir a Nueva
York!
```

```
Introduzca el nombre de una ciudad que haya visitado:
(Introduce 'quit' cuando hayas terminado.) San Francisco
Me encantaría ir a San Francisco.

Introduzca el nombre de una ciudad que haya visitado:
(Introduce 'quit' cuando hayas terminado.) quit
```

NOTE *Puedes utilizar la sentencia* `break` *en cualquiera de los bucles de Python. Por ejemplo, puede utilizar* `break` *para salir de un bucle* `for` *que está trabajando a través de una lista o un diccionario.*

Uso de continue en un bucle

En lugar de salir completamente de un bucle sin ejecutar el resto de su código, puede utilizar la sentencia `continue` para volver al principio del bucle basándose en el resultado de una prueba condicional. Por ejemplo, considere un bucle que cuenta del 1 al 10 pero imprime sólo los números impares en ese rango:

contando.py

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

    print(número_actual)
```

❶

Primero ponemos `número_actual` a 0. Como es menor que 10, Python entra en el `bucle while`. Una vez dentro del `bucle`, incrementamos la cuenta en 1 en ❶, por lo que `número_actual` es 1. A continuación, la `sentencia if` comprueba el módulo de `número_actual` y 2. Si el módulo es 0 (lo que significa que número_actual es divisible por 2), la sentencia continue le dice a Python que continúe. Si el módulo es 0 (lo que significa que `número_actual` es divisible por 2), la sentencia `continue` le dice a Python que ignore el resto del bucle y vuelva al principio. Si el número actual no es divisible por 2 , se ejecuta el resto del bucle y Python imprime el número actual:

```
1
3
5
7
9
```

Evitar bucles infinitos

Cada bucle `while` necesita una forma de parar su ejecución para que no continúe ejecutándose eternamente. Por ejemplo, este bucle de conteo debe contar de 1 a 5:

contando.py

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

Pero si accidentalmente omite la línea `x += 1` (como se muestra a continuación), el bucle se ejecutará eternamente:

```
# ¡Este bucle se ejecuta
para siempre! x = 1
mientras x <= 5:
    print(x)
```

Ahora el valor de `x` empezará en `1` pero nunca cambiará. Como resultado, la prueba con- dicional `x <= 5` siempre se evaluará como `True` y el bucle `while` se ejecutará para siempre, imprimiendo una serie de 1s, así:

```
1
1
1
1
--snip--
```

Todo programador escribe accidentalmente un bucle `while` infinito de vez en cuando, especialmente cuando los bucles de un programa tienen sutiles condiciones de salida. Si su programa se queda atascado en un bucle infinito, pulse CTRL-C o simplemente cierre la ventana del terminal que muestra la salida de su programa.

Para evitar escribir bucles infinitos, prueba cada bucle `while` y asegúrate de que el bucle se detiene cuando esperas que lo haga. Si quieres que tu programa termine cuando el usuario introduzca un determinado valor de entrada, ejecuta el programa e introduce ese valor. Si el programa no termina, escudriñe la forma en que su programa maneja el valor que debería causar la salida del bucle. Asegúrese de que al menos una parte del programa puede hacer que la condición del bucle sea `False` o hacer que llegue a una sentencia `break`.

NOTA *Algunos editores, como Sublime Text, tienen una ventana de salida incrustada. Esto puede dificultar la detención de un bucle infinito, y puede que tengas que cerrar el editor para terminar el bucle.*

TRY IT yOURSELF

7-4. Pizza Toppings: Escribe un bucle que pida al usuario que introduzca una serie de ingredientes para la pizza hasta que introduzca un valor de `'quit'`. A medida que ingresan cada ingrediente, imprime un mensaje diciendo que agregarás ese ingrediente a su pizza.

7-5. Entradas de cine: Los cines cobran precios diferentes según la edad de la persona. Si una persona tiene menos de 3 años, la entrada es gratuita; si tiene entre 3 y 12 años, la entrada cuesta 10 dólares; y si tiene más de 12 años, la entrada cuesta 1 euro.
$15. Escribe un bucle en el que preguntes a los usuarios su edad y luego les digas el precio de su entrada de cine.

(continuación)

7-6. Tres salidas: Escribe diferentes versiones del Ejercicio 7-4 o del Ejercicio 7-5 que hagan cada una de las siguientes cosas al menos una vez:

- Utilice una prueba condicional en la sentencia `while` para detener el bucle.
- Utilice una variable `activa` para controlar la duración del bucle.
- Utilice una sentencia `break` para salir del bucle cuando el usuario introduzca un valor `'quit'`.

7-7. Infinito: Escribe un bucle que no termine nunca y ejecútalo. (Para terminar el bucle, pulse CTRL-C o cierre la ventana que muestra la salida)

Uso de un bucle while con listas y diccionarios

Hasta ahora, hemos trabajado con una sola pieza de información del usuario a la vez. Recibimos la entrada del `usuario` y luego imprimimos la entrada o una respuesta a ella. La próxima vez a través del bucle `while`, recibiríamos otro valor de entrada y responderíamos a eso. Pero para realizar un seguimiento de muchos usuarios y piezas de información, tendremos que utilizar listas y diccionarios con nuestros bucles `while`.

Un bucle `for` es efectivo para recorrer una lista, pero no deberías modificar una lista dentro de un bucle for porque Python tendrá problemas para seguir la pista de los elementos de la lista. Para modificar una lista a medida que la recorres, utiliza un bucle `while`. Usar bucles `while` con listas y diccionarios te permite recoger, almacenar y organizar mucha información para examinarla y elaborar informes más tarde.

Mover elementos de una lista a otra

Consideremos una lista de usuarios recién registrados pero no verificados de un sitio web. Después de verificar estos usuarios, ¿cómo podemos moverlos a una lista separada de usuarios confirmados? Una forma sería utilizar un bucle `while` para extraer usuarios de la lista de usuarios no confirmados a medida que los verificamos y luego los añadimos a una lista separada de usuarios confirmados. Este es el aspecto que podría tener ese código:

confirmed_usuarios.py

```
# Empezar con los usuarios que necesitan ser verificados,
# y una lista vacía para contener los usuarios confirmados.
❶ unconfirmed_users = ['alice', 'brian', 'candace']
confirmed_users = []

# Verifica cada usuario hasta que no haya más usuarios sin
confirmar. # Mueve cada usuario verificado a la lista de
usuarios confirmados.
❷ mientras que unconfirmed_users:
❸     usuario_actual = usuarios_no_confirmados.pop()
```

```
        print("Verificando usuario: " + usuario_actual.title())
❹       confirmed_users.append(usuario_actual)
```

```
# Mostrar todos los usuarios confirmados.
print("Los siguientes usuarios han sido
confirmados:") for usuario_confirmado in
usuarios_confirmados:
    print(usuario_confirmado.titulo())
```

Comenzamos con una lista de usuarios no confirmados en ❶ (Alice, Brian y Candace) y una lista vacía para contener los usuarios confirmados. El bucle `while` en ❷ se ejecuta mientras la lista `unconfirmed_users` no esté vacía. Dentro de este bucle, la función `pop()` en ❸ elimina los usuarios no verificados de uno en uno desde el final de `unconfirmed_users`. Aquí, como Candace es la última en la lista de `unconfirmed_users`, su nombre será el primero en ser eliminado, almacenado en `current_user`, y añadido a la lista de `confirmed_users` en ❹. El siguiente es Brian, y después Alice.

Simulamos la confirmación de cada usuario imprimiendo un mensaje de verificación y añadiéndolo después a la lista de usuarios confirmados. A medida que se reduce la lista de usuarios no confirmados, aumenta la lista de usuarios confirmados. Cuando la lista de usuarios no confirmados está vacía, el bucle se detiene y se imprime la lista de usuarios confirmados:

```
Verificando usuario:
Candace Usuario
verificador: Brian
Usuario verificador:
Alice

Los siguientes usuarios han sido confirmados:
Candace
Brian
Alice
```

Eliminar todas las instancias 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 sólo aparecía una vez en la lista. Pero, ¿y si queremos eliminar todas las instancias de un valor de una lista?

Digamos que tienes una lista de mascotas con el valor `'gato'` repetido varias veces. Para eliminar todas las instancias de ese valor, puede ejecutar un bucle `while` hasta que `'gato'` ya no esté en la lista, como se muestra aquí:

mascotas.py

```
    pets = ['perro', 'gato', 'perro', 'pez', 'gato',
'conejo', 'gato'] print(pets)

while 'gato' in mascotas:
    mascotas.eliminar('gato')

print(mascotas)
```

Empezamos con una lista que contiene múltiples instancias de `'cat'`.

Después de imprimir la lista, Python entra en el bucle `while` porque encuentra el valor `'cat'` en la lista

al menos una vez. Una vez dentro del bucle, Python elimina la primera instancia de 'cat', vuelve a la línea `while`, y vuelve a entrar en el bucle cuando encuentra que `'cat'` todavía está en la lista. Elimina cada instancia de 'cat' hasta que el valor deja de estar en la lista, momento en el que Python sale del bucle y vuelve a imprimir la lista:

```
["perro", "gato", "perro", "carpa dorada", "gato",
"conejo", "gato"] ["perro", "perro", "carpa dorada",
"conejo"]
```

Rellenar un diccionario con datos introducidos por el usuario

Puedes pedir tanta información como necesites en cada paso de un bucle `while`. Hagamos un programa de sondeo en el que en cada paso por el bucle se pida el nombre y la respuesta del participante. Almacenaremos los datos que obtengamos en un diccionario, porque queremos relacionar cada respuesta con un usuario concreto:

montaña_encuesta.py

```
respuestas = {}
# Establecer una bandera para indicar que el sondeo está
activo. polling_active = True

mientras sondeo_activo:
    # Pregunte el nombre de la persona y responda.
❶   nombre = input("\n¿Cuál es tu nombre? ")
    response = input("¿Qué montaña te gustaría escalar algún día?")

    # Almacena la respuesta en el diccionario:
❷   responses[name] = respuesta

    # Averigua si alguien más va a hacer la encuesta.
❸   repeat = input("¿Desea que responda otra persona? (sí/no) ") if repeat ==
    'no':
        polling_active = False

# El sondeo ha terminado. Mostrar los
resultados. print("\n--- Poll Results --
-")
❹ para nombre, respuesta en respuestas.items():
    print(nombre + " quisiera subir " + respuesta + ".")
```

El programa define primero un diccionario vacío (`responses`) y establece una bandera (`polling_active`) para indicar que el sondeo está activo. Mientras `polling_active` sea `True`, Python ejecutará el código en el bucle `while`.

Dentro del bucle, se pide al usuario que introduzca su nombre de usuario y una montaña que le gustaría escalar ❶. Esa información se almacena en el diccionario de `respuestas` ❷, y se pregunta al usuario si desea o no seguir con el sondeo ❸. En caso `afirmativo, el` programa vuelve a entrar en el bucle

`while`. Si la respuesta es `negativa`, el indicador `polling_active se` establece en `False`, el bucle `while` deja de ejecutarse y el bloque de código final en ❹ muestra los resultados de la encuesta.

Si ejecuta este programa e introduce respuestas de ejemplo, debería ver una salida como ésta:

```
¿Cómo te llamas? Eric
¿Qué montaña le gustaría escalar algún día? Denali
¿Desea que responda otra persona? (sí/no) sí

¿Cómo te llamas? Lynn
¿Qué montaña te gustaría escalar algún día? Pulgar del Diablo
¿Desea que responda otra persona? (sí/no) no

--- Resultados de la encuesta ---
A Lynn le gustaría escalar el Pulgar del
Diablo. A Eric le gustaría escalar el
Denali.
```

TRY IT yOURSELF

7-8. Deli: Crea una lista llamada `pedidos_sandwiches` y rellénala con los nombres de varios sándwiches. Luego haz una lista vacía llamada `bocadillos_terminados`. Recorre la lista de pedidos de sándwiches e imprime un mensaje para cada pedido, por ejemplo: `He hecho tu sándwich de atún.` A medida que se hace cada sándwich, muévelo a la lista de sándwiches terminados. Después de que todos los sándwiches hayan sido hechos, imprime un mensaje listando cada sándwich que fue hecho.

7-9. No Pastrami: Usando la lista `sandwich_orders` del Ejercicio 7-8, asegúrese de que el sandwich `'pastrami'` aparece en la lista al menos tres veces. Añade código cerca del principio de tu programa para imprimir un mensaje diciendo que la charcutería se ha quedado sin pastrami, y luego usa un bucle `while` para eliminar todas las apariciones de `'pastrami'` de `sandwich_pedidos`. Asegúrate de que ningún bocadillo de pastrami termina en `bocadillos_terminados`.

7-10. Vacaciones de ensueño: Escribe un programa que sondee a los usuarios sobre sus vacaciones soñadas. Escribe una pregunta similar a *Si pudieras visitar un lugar en el mundo, ¿a dónde irías?* Incluye un bloque de código que imprima los resultados de la encuesta.

resumen

En este capítulo ha aprendido a utilizar `input()` para permitir a los usuarios proporcionar su propia información en sus programas. Aprendió a trabajar tanto con texto como con entradas numéricas y cómo usar bucles `while` para hacer que sus programas se ejecuten tanto tiempo como sus usuarios quieran. Viste varias formas de controlar el flujo de un bucle `while` estableciendo una bandera `activa`, usando la sentencia `break`, y

utilizando la sentencia `continue`. Aprendiste cómo usar un bucle `while` para mover elementos de una lista a otra y cómo eliminar todas las instancias de un valor de una lista. También ha aprendido cómo utilizar los bucles `while` con diccionarios.

En el Capítulo 8 aprenderás sobre las *funciones*. Las funciones te permiten dividir tus programas en pequeñas partes, cada una de las cuales hace un trabajo específico. Puedes llamar a una función tantas veces como quieras, y puedes almacenar tus funciones en archivos separados. Usando funciones, serás capaz de escribir código más eficiente que es más fácil de solucionar y mantener y que puede ser reutilizado en muchos programas diferentes.

8

FUNCIONES



En este capítulo aprenderás a escribir *funciones*, que son bloques de código con nombre que están diseñados para hacer un trabajo específico.

Cuando desee realizar una tarea concreta que haya definido en una función, *llame* al nombre de la función responsable de la misma. Si necesita realizar esa tarea varias veces a lo largo de tu programa, no necesitas escribir todo el código para la misma tarea una y otra vez; simplemente llamas a la función dedicada a manejar esa tarea, y la llamada le dice a Python que ejecute el código dentro de la función. Verás que el uso de funciones hace que tus programas sean más fáciles de escribir, leer, probar y corregir.

En este capítulo también aprenderás formas de pasar información a las funciones. Aprenderás a escribir ciertas funciones cuyo trabajo principal es mostrar información y otras funciones diseñadas para procesar datos y devolver un valor o conjunto de valores. Finalmente, aprenderás a almacenar funciones en archivos separados llamados *módulos* para ayudar a organizar los archivos principales de tu programa.

Definición de una función

Aquí hay una función simple llamada `greet_user()` que imprime un saludo:

greeter.py
```
❶ def greet_user():
❷     """Mostrar un saludo sencillo."""
❸     print("¡Hola!")

❹ greet_user()
```

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

Las líneas sangradas que siguen a `def greet_user():` constituyen el *cuerpo* de la función. El texto en ❷ es un comentario llamado *docstring*, que describe lo que hace la función. Los docstrings van entre comillas triples, que Python busca cuando genera documentación para las funciones de tus programas.

La línea `print("¡Hola!")` ❸ es la única línea de código real en el cuerpo de esta función, por lo que `greet_user()` sólo tiene un trabajo: `imprimir("¡Hola!")`.

Cuando quieras usar esta función, llámala. Una llamada *a una función* le dice a Python que ejecute el código de la función. Para *llamar a* una función, escribes el nombre de la función, seguido de cualquier información necesaria en paréntesis, como se muestra en ❹. Como aquí no se necesita ninguna información, llamar a nuestra función es tan sencillo como escribir `greet_user()`. Como era de esperar, imprime `¡Hola!`

```
¡Hola!
```

Pasar información a una función

Modificada ligeramente, la función `greet_user()` no sólo puede decir al usuario `¡Hola!` sino también saludarle por su nombre. Para que la función haga esto, introduce `nombre_usuario` en el paréntesis de la definición de la función en `def greet_user()`. Añadiendo `nombre_usuario` aquí permites que la función acepte cualquier valor de `nombre_usuario` que especifiques. La función ahora espera que proporciones un valor para el nombre de `usuario` cada vez que la llames. Cuando llames a `greet_user()`, puedes pasarle un nombre, como `'jesse'`, dentro de los paréntesis:

```
def saludar_usuario(nombre_usuario):
    """Mostrar un simple saludo.""" print("¡Hola,
```

```
    " + username.title() + "!")

greet_user('jesse')
```

Introduciendo `greet_user('jesse')` llama a `greet_user()` y da a la función la información que necesita para ejecutar la sentencia `print`. La función acepta el nombre que le pasaste y muestra el saludo para ese nombre:

```
¡Hola, Jesse!
```

Del mismo modo, introducir `greet_user('sarah')` llama a `greet_user()`, le pasa `'sarah'`, e imprime `¡Hola, Sarah!` Puedes llamar a `greet_user()` tantas veces como quieras y pasarle cualquier nombre que desees para producir una salida predecible cada vez.

Argumentos y parámetros

En la función anterior `greet_user()`, definimos `greet_user()` para requerir un valor para la variable `nombre_usuario`. Una vez que llamamos a la función y le dimos la información (el nombre de una persona), imprimió el saludo correcto.

La variable `nombre_usuario` en la definición de greet_user() es un ejemplo de *parámetro*, una pieza de información que la función necesita para hacer su trabajo. El valor `'jesse'` en greet_user(`'jesse'`) es un ejemplo de *argumento*. Un argumento
es una pieza de información que se pasa de una llamada a una función.
Cuando llamamos a la función, colocamos entre paréntesis el valor con el que queremos que trabaje la función. En este caso el argumento `'jesse'` se pasó a la función `greet_user()`, y el valor se almacenó en el parámetro `username`.

NOTE *A veces se habla indistintamente de argumentos y parámetros. No se sorprenda si ve que las variables de una definición de función se denominan argumentos o que las variables de una llamada a función se denominan parámetros.*

TRY IT yOURSELF

8-1. Mensaje: Escriba una función llamada `display_message()` que imprima una sen- cencia diciendo a todos lo que está aprendiendo en este capítulo. Llama a la función y asegúrate de que el mensaje se muestra correctamente.

8-2. Libro favorito: Escribe una función llamada `libro_favorito()` que acepte un parámetro, `título`. La función debe imprimir un mensaje, tal como `Uno de mis libros favoritos es Alicia en el País de las Maravillas`. Llama a la función, asegurándote de incluir un título de libro como argumento en la llamada a la función.

Pasar argumentos

Dado que una definición de función puede tener varios parámetros, una llamada a una función puede necesitar varios argumentos. Puede pasar argumentos a sus funciones

de varias maneras. Puede utilizar *argumentos posicionales*, que deben estar en

en el mismo orden en que se escribieron los parámetros; *argumentos de palabra clave*, en los que cada argumento consta de un nombre de variable y un valor; y listas y diccionarios de valores. Veamos cada uno de ellos por separado.

Argumentos posicionales

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

Para ver cómo funciona, considere una función que muestra información sobre mascotas. La función nos dice qué tipo de animal es cada mascota y el nombre de la mascota, como se muestra aquí:

pets.py

```
❶ def describir_mascota(tipo_animal,
      nombre_mascota): """Mostrar información
      sobre una mascota.""" print("\nTengo un " +
      tipo_animal + ".")
      print("El nombre de mi " + tipo_animal + " es " + nombre_mascota.title() +
      ".")

❷ describir_mascota('hamster', 'harry')
```

La definición muestra que esta función necesita un tipo de animal y el nombre del animal ❶. Cuando llamamos a `describe_pet()`, necesitamos proporcionar un tipo de ani- mal y un `nombre`, en ese orden. Por ejemplo, en la llamada a la función, el argumento `"hámster"` se almacena en el parámetro `tipo_animal` y el argumento `"harry"` se almacena en el parámetro `nombre_mascota` ❷. En el cuerpo de la función, estos dos parámetros se utilizan para mostrar información sobre la mascota que se está describiendo.

La salida describe a un hámster llamado Harry:

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

Múltiples llamadas a funciones

Puede llamar a una función tantas veces como sea necesario. La descripción de una segunda mascota diferente requiere sólo una llamada más a `describe_pet()`:

```
def describir_mascota(tipo_animal,
    nombre_mascota): """Mostrar información sobre
    una mascota.""" print("\nTengo un " +
    tipo_animal + ".")
    print("El nombre de mi " + tipo_animal + " es " + nombre_mascota.title() +
    ".")
```

```
describe_mascota('hamster', 'harry')
describe_mascota('perro', 'willie')
```

En esta segunda llamada a la función, pasamos a `describe_pet()` los argumentos `'perro'` y `'willie'`. Al igual que con el conjunto anterior de argumentos que utilizamos, Python hace coincidir `'perro'` con el parámetro `tipo_animal` y `'willie'` con el parámetro `nombre_mascota`.

Como antes, la función hace su trabajo, pero esta vez imprime valores para un perro llamado `Willie`. Ahora tenemos un hámster llamado Harry y un perro llamado `Willie`:

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

Tengo un perro.
Mi perro se llama Willie.
```

Llamar a una función varias veces es una forma muy eficiente de trabajar. El código que describe una mascota se escribe una vez en la función. Luego, cada vez que quiera describir una nueva mascota, llame a la función con la información de la nueva mascota. Incluso si el código para describir una mascota se extendiera a diez líneas, podría describir una nueva mascota en una sola línea llamando de nuevo a la función.

Puedes utilizar tantos argumentos posicionales como necesites en tus funciones. Python revisa los argumentos que proporcionas al llamar a la función y compara cada uno de ellos con el parámetro correspondiente en la definición de la función.

El orden es importante en los argumentos posicionales

Puede obtener resultados inesperados si confunde el orden de los argumentos en una llamada a función cuando utiliza argumentos posicionales:

```
def describir_mascota(tipo_animal,
    nombre_mascota): """Mostrar información sobre
    una mascota.""" print("\nTengo un " +
    tipo_animal + ".")
    print("El nombre de mi " + tipo_animal + " es " + nombre_mascota.title() +
    ".")

describe_mascota('harry', 'hamster')
```

En esta llamada a la función listamos primero el nombre y después el tipo de animal. Como el argumento `'harry'` aparece primero esta vez, ese valor se almacena en el parámetro `tipo_animal`. Del mismo modo, `"hamster"` se almacena en `pet_name`. Ahora tenemos un "harry" llamado "Hamster":

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

Si obtiene resultados extraños como éste, compruebe que el orden de los argumentos en la llamada a la función coincide con el orden de los parámetros en la definición de la función.

Argumentos de las palabras clave

Un *argumento de palabra clave* es un par nombre-valor que se pasa a una función. Se asocian directamente el nombre y el valor dentro del argumento,

de modo que cuando se pasa el argumento a la función, no hay confusión (no acabará

con un Harry llamado Hamster). Los argumentos de palabra clave le liberan de tener que preocuparse de ordenar correctamente sus argumentos en la llamada a la función, y aclaran el papel de cada valor en la llamada a la función.

Reescribamos *pets.py* usando argumentos de palabra clave para llamar a `describe_pet()`:

```
def describir_mascota(tipo_animal,
    nombre_mascota): """Mostrar información sobre
    una mascota.""" print("\nTengo un " +
    tipo_animal + ".")
    print("El nombre de mi " + tipo_animal + " es " + nombre_mascota.title() +
    ".")

describe_mascota(tipo_animal='hamster', nombre_mascota='harry')
```

La función `describe_pet()` no ha cambiado. Pero cuando llamamos a la función, le decimos explícitamente a Python con qué parámetro debe emparejarse cada argumento. Cuando Python lee la llamada a la función, sabe que debe almacenar el argumento `"hámster"` en el parámetro `tipo_animal` y el argumento `"harry"` en `nombre_mascota`. La salida muestra correctamente que tenemos un hámster llamado Harry.

El orden de los argumentos de las palabras clave no importa porque Python sabe dónde debe ir cada valor. Las siguientes dos llamadas a funciones son equivalentes:

```
describe_pet(tipo_animal='hamster', nombre_mascota='harry')
describe_pet(nombre_mascota='harry', tipo_animal='hamster')
```

NOTE *Cuando utilice argumentos de palabra clave, asegúrese de utilizar los nombres exactos de los parámetros en la definición de la función.*

Valores por defecto

Cuando se escribe una función, se puede definir un *valor por defecto* para cada parámetro. Si se proporciona un argumento para un parámetro en la llamada a la función, Python utiliza el valor del argumento. Si no, utiliza el valor por defecto del parámetro. Así que cuando defines un valor por defecto para un parámetro, puedes excluir el argumento correspondiente que normalmente escribirías en la llamada a la función. El uso de valores por defecto puede simplificar las llamadas a funciones y aclarar la forma en que éstas se utilizan normalmente.

Por ejemplo, si observas que la mayoría de las llamadas a `describe_pet()` se utilizan para describir perros, puedes establecer el valor por defecto de `animal_type` en `'perro'`. Ahora cualquiera que llame a `describe_pet()` para un perro puede omitir esa información:

```
def describir_mascota(nombre_mascota,
    tipo_animal='perro'): """Mostrar información
    sobre una mascota.""" print("\nTengo un " +
```

```
    tipo_animal + ".")
    print("El nombre de mi " + tipo_animal + " es " + nombre_mascota.title() +
    ".")

describe_mascota(nombre_mascota='willie')
```

Hemos cambiado la definición de `describe_pet()` para incluir un valor por defecto, `'perro'`, para `animal_type`. Ahora, cuando se llama a la función sin especificar animal_type, Python sabe que debe usar el valor `'perro'` para este parámetro:

```
Tengo un perro.
Mi perro se llama Willie.
```

Observe que ha sido necesario cambiar el orden de los parámetros en la definición de la función. Como el valor por defecto 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 todavía interpreta esto como un argumento posicional, así que si la función se llama sólo con el nombre de la mascota, ese argumento coincidirá con el primer parámetro listado en la definición de la función. Esta es la razón por la que el primer parámetro debe ser `nombre_mascota`.

La forma más sencilla de utilizar esta función ahora es proporcionar sólo el nombre del perro en la llamada a la función:

```
describe_mascota('willie')
```

Esta llamada a la función tendría el mismo resultado que el ejemplo anterior.

El único argumento proporcionado es `'willie'`, por lo que se empareja con el primer parámetro de la definición, `nombre_mascota`. Como no se proporciona ningún argumento para `tipo_animal`, Python utiliza el valor por defecto `"perro"`.

Para describir un animal que no sea un perro, puedes utilizar una llamada a una función como ésta:

```
describe_mascota(nombre_mascota='harry', tipo_animal='hamster')
```

Dado que se proporciona un argumento explícito para `animal_type`, Python ignorará el valor por defecto del parámetro.

NOTE *Cuando usas valores por defecto, cualquier parámetro con un valor por defecto necesita ser listado después de todos los parámetros que no tienen valores por defecto. Esto permite a Python seguir interpretando correctamente los argumentos posicionales.*

Llamadas a funciones equivalentes

Dado que los argumentos posicionales, los argumentos de palabra clave y los valores por defecto pueden utilizarse conjuntamente, a menudo dispondrá de varias formas equivalentes de llamar a una función. Considere la siguiente definición de `describe_pets()` con un valor por defecto:

```
def describir_mascota(nombre_mascota, tipo_animal='perro'):
```

Con esta definición, siempre es necesario proporcionar un argumento para
`nombre_mascota`, y este valor puede proporcionarse utilizando el posicional o la palabra clave

formato. Si el animal que se describe no es un perro, debe incluirse un argumento para `tipo_animal` en la llamada, y este argumento también puede especificarse utilizando el formato posicional o de palabra clave.

Todas las llamadas siguientes servirían para esta función:

```
# Un perro llamado Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# Un hámster llamado Harry.
describe_pet('harry', 'hamster')
describe_pet(nombre_mascota='harry', tipo_animal='hamster')
describe_pet(tipo_animal='hamster', nombre_mascota='harry')
```

Cada una de estas llamadas a funciones tendría la misma salida que en los ejemplos anteriores.

NOTE *En realidad, no importa qué estilo de llamada utilice. Mientras las llamadas a las funciones produzcan el resultado deseado, utilice el estilo que le resulte más fácil de entender.*

Evitar errores de argumentación

Cuando empiece a utilizar funciones, no se sorprenda si encuentra errores sobre argumentos no coincidentes. Los argumentos no coincidentes ocurren cuando proporcionas menos o más argumentos de los que una función necesita para hacer su trabajo. Por ejemplo, esto es lo que ocurre si intentamos llamar a `describe_pet()` sin argumentos:

```
def describir_mascota(tipo_animal,
    nombre_mascota): """Mostrar información sobre
    una mascota.""" print("\nTengo un " +
    tipo_animal + ".")
    print("El nombre de mi " + tipo_animal + " es " + nombre_mascota.title() +
    ".")

describir_mascota()
```

Python reconoce que falta alguna información en la llamada a la función, y el traceback nos lo indica:

```
Traceback (última llamada más reciente):
❶  File "pets.py", line 6, in <module>.
❷    describir_mascota()
❸ TypeError: describe_pet() missing 2 required positional arguments:
'tipo_animal' y 'nombre_mascota'
```

En ❶ el rastreo nos indica la ubicación del problema, lo que nos permite mirar hacia atrás y ver que algo salió mal en nuestra llamada a la función. En ❷ se escribe la llamada a la función infractora para que la veamos. En ❸ el traceback

nos dice que a la llamada le faltan dos argumentos e informa de los nombres de los argumentos que faltan. Si esta función estuviera en un archivo separado, probablemente podríamos reescribir la llamada correctamente sin tener que abrir ese archivo y leer el código de la función.

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

Si proporciona demasiados argumentos, debería obtener un rastreo similar que puede ayudarle a hacer coincidir correctamente su llamada a la función con la definición de la función.

TRY IT yOURSELF

8-3. Camiseta: Escribe una función llamada `hacer_camiseta()` que acepte una talla y el texto de un mensaje que debe imprimirse en la camiseta. La función debe imprimir una sentencia que resuma el tamaño de la camiseta y el mensaje impreso en ella.

Llame a la función una vez utilizando argumentos posicionales para hacer una camisa. Llama a la función una segunda vez utilizando argumentos de palabra clave.

8-4. Camisas grandes: Modifique la función `make_shirt()` para que las camisas sean grandes por defecto con un mensaje que diga *I love Python*. Haga una camisa grande y una camiseta mediana con el mensaje por defecto, y una camiseta de cualquier talla con un mensaje diferente.

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

Valores de retorno

Una función no siempre tiene que mostrar su salida directamente. En su lugar, puede procesar algunos datos y luego devolver un valor o conjunto de valores. El valor que devuelve la función se denomina *valor de retorno*. La sentencia `return` toma un valor del interior de una función y lo devuelve a la línea que llamó a la función. Los valores de retorno le permiten mover gran parte del trabajo pesado de su programa a las funciones, lo que puede simplificar el cuerpo de su programa.

Devolver un valor simple

Veamos una función que toma un nombre y un apellido y devuelve un nombre completo con un formato claro:

formatted_nombre.py

```
❶ def get_formatted_name(first_name, last_name):
    """Devuelve un nombre completo, ordenadamente
                                    formateado."""
❷   nombre_completo = nombre + ' ' + apellido
❸   return nombre_completo.titulo()

❹ músico = get_formatted_name('jimi', 'hendrix')
print(músico)
```

La definición de `get_formatted_name()` toma como parámetros un nombre y un apellido ❶. La función combina estos dos nombres, añade un espacio entre ellos y almacena el resultado en `nombre_completo` ❷. El valor de `nombre_completo` se convierte a mayúsculas y se devuelve a la línea de llamada en ❸.

Cuando se llama a una función que devuelve un valor, es necesario proporcionar una variable donde se pueda almacenar el valor devuelto. En este caso, el valor devuelto se almacena en la variable `músico` en ❹. La salida muestra un nombre prolijamente for- mado compuesto por las partes del nombre de una persona:

```
Jimi Hendrix
```

Esto puede parecer mucho trabajo para obtener un nombre bien formateado cuando podríamos haber escrito simplemente:

```
print("Jimi Hendrix")
```

Pero cuando se piensa en trabajar con un programa grande que necesita almacenar muchos nombres y apellidos por separado, funciones como `get_formatted_name()` resultan muy útiles. Se almacenan los nombres y apellidos por separado y luego se llama a esta función cada vez que se desea mostrar un nombre completo.

Convertir un argumento en opcional

A veces tiene sentido hacer que un argumento sea opcional para que las personas que utilicen la función puedan elegir proporcionar información adicional sólo si lo desean. Puede utilizar valores por defecto para hacer que un argumento sea opcional.

Por ejemplo, supongamos que queremos ampliar `get_formatted_name()` para que también incluya los segundos nombres. Un

primer intento de incluir los segundos nombres podría ser el siguiente:

```
def obtener_nombre_formateado(nombre, segundo nombre,
    apellido): """Devuelve un nombre completo, bien
    formateado""".
    nombre_completo = nombre_completo + ' ' + segundo_nombre + ' ' +
    apellido return nombre_completo.title()

músico = get_formatted_name('john', 'lee', 'hooker')
print(músico)
```

Esta función funciona cuando se le da un nombre, un segundo nombre y un apellido. La función toma las tres partes del nombre y construye una cadena a partir de ellas. La función añade espacios donde corresponda y convierte el nombre completo a mayúsculas y minúsculas:

```
John Lee Hooker
```

Pero los segundos nombres no siempre son necesarios, y esta función, tal y como está escrita, no funcionaría si se intentara invocarla sólo con un nombre y un apellido. Para que el segundo nombre sea opcional, podemos dar al argumento `middle_name` un valor vacío por defecto e ignorar el argumento a menos que el usuario proporcione un valor. Para hacer que `get_formatted_name()` funcione sin un segundo nombre, establecemos el valor por defecto de `middle_name` a una cadena vacía y la movemos al final de la lista de parámetros:

```
❶ def obtener_nombre_formateado(nombre, apellido, segundo_nombre=''):
      """Devuelve un nombre completo, bien formateado""".
❷     si segundo_nombre:
          full_name = first_name + ' ' + middle_name + ' + last_name
❸     si no:
          nombre_completo = nombre_completo + ' ' +
      apellido return nombre_completo.title()

músico = get_formatted_name('jimi', 'hendrix')
print(músico)

❹ músico = get_formatted_name('john', 'hooker', 'lee') print(músico)
```

En este ejemplo, el nombre se construye a partir de tres partes posibles. Como siempre hay un nombre y un apellido, estos parámetros aparecen primero en la definición de la función. El segundo nombre es opcional, por lo que aparece al final de la definición, y su valor por defecto es una cadena vacía ❶.

En el cuerpo de la función, comprobamos si se ha proporcionado un segundo nombre. Python interpreta las cadenas no vacías como `True`, por lo que `si middle_name` evalúa a `True` si hay un argumento de segundo nombre en la llamada a la función ❷. Si se proporciona un segundo nombre, el nombre, el segundo nombre y los apellidos se combinan para formar un nombre completo. Este nombre se cambia a mayúsculas y se devuelve a la línea de llamada a la función, donde se almacena en la variable `musician` y se imprime. Si no se proporciona el segundo nombre, la cadena vacía falla la prueba `if` y se ejecuta el bloque `else` ❸. El nombre completo se hace sólo con nombre y apellido, y el nombre formateado se devuelve a la línea de llamada donde se almacena en `musician y se imprime`.

Llamar a esta función con un nombre y un apellido es sencillo. Sin embargo, si usamos un segundo nombre, tenemos que asegurarnos de que el segundo nombre es el último argumento pasado para que Python

empareje los argumentos posicionales correctamente ❹.

Esta versión modificada de nuestra función funciona para las personas que sólo tienen nombre y apellidos, y también para las que tienen segundo nombre:

```
Jimi Hendrix
John Lee Hooker
```

Los valores opcionales permiten que las funciones manejen una amplia gama de casos de uso, al tiempo que permiten que las llamadas a las funciones sigan siendo lo más sencillas posible.

Devolver un diccionario

Una función puede devolver cualquier tipo de valor, incluidas estructuras de datos más complejas, como listas y diccionarios. Por ejemplo, la siguiente función toma partes de un nombre y devuelve un diccionario que representa a una persona:

persona.py

```
def construir_persona(nombre, apellido):
    """Devuelve un diccionario de información sobre una persona."""
❶   persona = {'nombre': first_name, 'apellido': last_name}
❷   persona de vuelta

musician = build_person('jimi', 'hendrix')
❸ print(músico)
```

La función `construir_persona()` toma un nombre y un apellido, y empaqueta estos valores en un diccionario en ❶. El valor de `first_name` se almacena con la clave `'first'`, y el valor de `last_name` se almacena con la clave `'last'`. El diccionario completo que representa a la persona se devuelve en ❷. El valor devuelto se imprime en ❸ con las dos piezas originales de información textual ahora almacenadas en un diccionario:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Esta función toma información textual simple y la coloca en una estructura de datos más significativa que le permite trabajar con la información más allá de sólo imprimirla. Las cadenas `'jimi'` y `'hendrix'` se etiquetan ahora como nombre y apellidos. Puedes ampliar fácilmente esta función para que acepte valores opcionales como un segundo nombre, una edad, una ocupación o cualquier otra información que quieras almacenar sobre una persona. Por ejemplo, el siguiente cambio permite almacenar también la edad de una persona:

```
def construir_persona(nombre, apellido, edad=''):
    """Devuelve un diccionario de información sobre una
    persona."""  persona = {'first': nombre, 'last': apellido}
    si la edad:
```

```
        persona['edad'] = edad
    return persona

música = construir_persona('jimi', 'hendrix', edad=27)
print(música)
```

Añadimos un nuevo parámetro opcional `age` a la definición de la función y asignamos al parámetro un valor por defecto vacío. Si la llamada a la función incluye un valor para este parámetro, el valor se almacena en el diccionario. Esta función siempre almacena el nombre de una persona, pero también puede modificarse para almacenar cualquier otra información que desee sobre una persona.

Uso de una función con un bucle while

Puedes utilizar funciones con todas las estructuras de Python que has aprendido hasta ahora. Por ejemplo, usemos la función `get_formatted_name()` con un bucle `while` para saludar a los usuarios de manera más formal. Aquí tienes un primer intento de saludar a la gente usando su nombre y apellidos:

greeter.py

```
def obtener_nombre_formateado(nombre_apellido):
    """Devuelve un nombre completo, bien
    formateado."""  full_name = first_name + ' '
    + last_name return full_name.title()

# ¡Esto es un bucle infinito!
while True:
❶    print("\nPor favor, dígame su
    nombre:") f_name = input("Nombre:
    ") l_name = input("Apellido: ")

    nombre_formateado = get_nombre_formateado(nombre_f, nombre_l)
    print("\n¡Hola, " + nombre_formateado + "!")
```

Para este ejemplo, usamos una versión simple de `get_formatted_name()` que no involucra segundos nombres. El bucle `while` pide al usuario que introduzca su nombre, y le preguntamos su nombre y apellidos por separado ❶.

Pero hay un problema con este bucle `while`: No hemos definido una condición de abandono. ¿Dónde se pone una condición de abandono cuando se pide una serie de entradas? Queremos que el usuario sea capaz de salir tan fácilmente como sea posible, por lo que cada pregunta debe ofrecer una forma de salir. La sentencia `break` ofrece una forma directa de salir del bucle en cualquiera de los prompt:

```
def obtener_nombre_formateado(nombre_apellido):
    """Devuelve un nombre completo, bien
    formateado."""  full_name = first_name + ' ' +
    last_name return full_name.title()

mientras sea verdad:
    print("\nPor favor, dígame su nombre:")
    print("(introduzca 'q' en cualquier momento
    para salir)")

    f_name = input("Nombre: ") if
    f_name == 'q':
```

```
    romper

l_name = input("Apellido: ") if
l_name == 'q':
    romper
```

```
    nombre_formateado = get_nombre_formateado(nombre_f, nombre_l)
    print("\n¡Hola, " + nombre_formateado + "!")
```

Añadimos un mensaje que informa al usuario de cómo salir, y luego salimos del bucle si el usuario introduce el valor quit en cualquiera de los dos prompt. Ahora el programa continuará saludando a la gente hasta que alguien introduzca `'q'` para cualquiera de los nombres:

```
Por favor, dígame su nombre:
(introduzca "q" en cualquier
momento para salir) Nombre:
eric
Apellidos: matthes

Hola, Eric Matthes Por

favor, dime tu nombre:
(introduzca "q" en cualquier
momento para salir) Nombre: q
```

TRY IT yOURSELF

8-6. Nombres de ciudades: Escribe una función llamada `ciudad_país()` que reciba el nombre de una ciudad y su país. La función debe devolver una cadena con el siguiente formato:

```
"Santiago, Chile"
```

Llama a tu función con al menos tres pares ciudad-país e imprime el valor devuelto.

8-7. Album: Escribe una función llamada `make_album()` que construya un diccionario describiendo un álbum de música. La función debe tomar un nombre de artista y un título de álbum, y debe devolver un diccionario que contenga estas dos piezas de información. Utiliza la función para crear tres diccionarios que representen diferentes álbumes. Imprime cada valor devuelto para mostrar que los diccionarios almacenan la información del álbum correctamente.

Añade un parámetro opcional a `make_album()` que permite almacenar el número de pistas de un álbum. Si la línea de llamada incluye un valor para el número de pistas, añada ese valor al diccionario del álbum. Realiza al menos una nueva llamada a una función que incluya el número de pistas de un álbum.

8-8. Álbumes de Usuario: Comience con su 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 tenga esa información, llame a `make_album()` con la entrada del usuario e imprima el diccionario creado. Asegúrese de incluir un valor quit en el bucle `while`.

Pasar una lista

A menudo te resultará útil pasar una lista a una función, ya sea una lista de nombres, números u objetos más complejos, como diccionarios. Cuando pasas una lista a una función, ésta obtiene acceso directo al contenido de la lista. Utilicemos las funciones para trabajar con listas de forma más eficaz.

Digamos que tenemos una lista de usuarios y queremos imprimir un saludo a cada uno. El siguiente ejemplo envía una lista de nombres a una función llamada `greet_users()`, que saluda individualmente a cada persona de la lista:

salud_usuarios.py

```
def salud_usuarios(nombres):
    """Imprime un simple saludo a cada usuario de la
    lista.""" for nombre in nombres:
        msg = "¡Hola, " + nombre.título() + "!"
        print(msg)

❶ nombres_usuario = ['hannah', 'ty',
'margot'] greet_users(nombres_usuario)
```

Definimos `greet_users() de` modo que espera una lista de nombres, que almacena en los `nombres de los` parámetros. La función recorre la lista que recibe e imprime un saludo a cada usuario. En ❶ definimos una lista de usuarios y luego pasamos los nombres `de usuario de` la lista a `greet_users()`en nuestra llamada a la función:

```
¡Hola, Hannah!
¡Hola, Ty!
¡Hola, Margot!
```

Este es el resultado que queríamos. Cada usuario ve un saludo personalizado, y puedes llamar a la función cada vez que quieras saludar a un grupo específico de usuarios.

Modificación de una lista en una función

Cuando pasas una lista a una función, ésta puede modificarla. Cualquier cambio realizado en la lista dentro del cuerpo de la función es permanente, lo que te permite trabajar con eficacia incluso cuando manejas grandes cantidades de datos.

Pensemos en una empresa que crea modelos impresos en 3D a partir de diseños que envían los usuarios. Los diseños que necesitan ser impresos se almacenan en una lista, y después de ser impresos se mueven a otra lista. El siguiente código hace esto sin usar funciones:

printing_ modelos.py

```
# Comience con algunos diseños que necesitan ser impresos.
diseños_no_impresos = ['funda_iphone', 'colgante_robot',
'dodecaedro'] modelos_completados = []

# Simular la impresión de cada diseño, hasta que no quede ninguno.
# Mueve cada diseño a completed_models después de
imprimirlo. while unprinted_designs:
```

```
    diseño_actual = diseños_no_impresos.pop()
```

```
    # Simula la creación de una impresión 3D a
    partir del diseño. print("Imprimiendo modelo: "
    + diseño_actual)
    completed_models.append(diseño_actual)

# Mostrar todos los modelos completados.
print("Se han impreso los siguientes modelos:") for
modelo_completado in modelos_completados:
    print(modelo_completado)
```

Este programa comienza con una lista de diseños que deben imprimirse y una lista vacía llamada `modelos_completados` a la que se moverá cada diseño una vez impreso. Mientras los diseños permanezcan en `diseños_no_impresos`, el bucle `while` simula la impresión de cada diseño eliminando un diseño del final de la lista, almacenándolo en `diseño_actual` y mostrando un mensaje que indica que el diseño actual se está imprimiendo. A continuación, añade el diseño a la lista de modelos completados. Cuando el bucle termina de ejecutarse, se muestra una lista de los diseños que se han impreso:

```
Modelo de impresión: dodecaedro
Modelo de impresión: colgante
robot Modelo de impresión: funda
iphone

Se han impreso los siguientes modelos:
dodecaedro robot
colgante funda
iphone
```

Podemos reorganizar este código escribiendo dos funciones, cada una de las cuales hace un trabajo específico. La mayor parte del código no cambiará; sólo lo estamos haciendo más eficiente. La primera función se encargará de imprimir los diseños, y la segunda resumirá las impresiones que se han realizado:

```
❶ def imprimir_modelos(diseños_no_impresos,
    modelos_completados): """
    Simular la impresión de cada diseño, hasta que no
    quede ninguno. Mueva cada diseño a completed_models
    después de la impresión. """
    mientras que unprinted_designs:
        diseño_actual = diseños_no_impresos.pop()

        # Simula la creación de una impresión 3D a
        partir del diseño. print("Imprimiendo modelo: "
        + diseño_actual)
        completed_models.append(diseño_actual)
```

❷ def

```
    mostrar_modelos_completados(modelos_completados
    ): """Mostrar todos los modelos que se
    imprimieron""".
    print("Se han impreso los siguientes modelos:") for
    modelo_completado in modelos_completados:
        print(modelo_completado)

diseños_no_impresos = ['funda_iphone', 'colgante_robot', 'dodecaedro']
modelos_completados = []
```

```
print_models(diseños_no_impresos,
modelos_completados)
show_completed_models(modelos_completados)
```

En ❶ definimos la función `print_models()` con dos parámetros: una lista de diseños que necesitan ser impresos y una lista de modelos completados. Dadas estas dos listas, la función simula la impresión de cada diseño vaciando la lista
de diseños sin imprimir y rellenando la lista de modelos completados. En ❷ definimos la función `show_completed_models()` con un parámetro: la lista de modelos completados. Dada esta lista, `show_completed_models()` muestra el nombre de cada modelo que se imprimió.

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 movido a dos funciones separadas, lo que hace que la parte principal del programa sea más fácil de entender. Mira el cuerpo del programa para ver cuánto más fácil es entender lo que este programa está haciendo:

```
diseños_no_impresos = ['funda_iphone', 'colgante_robot', 'dodecaedro']
modelos_completados = []

print_models(diseños_no_impresos,
modelos_completados)
show_completed_models(modelos_completados)
```

Establecemos una lista de diseños sin imprimir y una lista vacía que contendrá los modelos terminados. Después, como ya hemos definido nuestras dos funciones, todo lo que tenemos que hacer es llamarlas y pasarles los argumentos adecuados. Llamamos a `print_models()` y le pasamos las dos listas que necesita; como era de esperar, `print_models()` simula la impresión de los diseños. Luego llamamos a `show_completed_models()` y le pasamos la lista de modelos completados para que pueda informar de los modelos que se han impreso. Los nombres descriptivos de las funciones permiten a otros leer este código y entenderlo, incluso sin comentarios.

Este programa es más fácil de ampliar y mantener que la versión sin funciones. Si necesitamos imprimir más diseños más adelante, podemos simplemente llamar a `print_models()` de nuevo. Si nos damos cuenta de que el código de impresión necesita ser modificado, podemos cambiar el código una vez, y nuestros cambios tendrán lugar en todas partes donde se llame a la función. Esta técnica es más eficiente que tener que actualizar el código por separado en varios lugares del programa.

Este ejemplo también demuestra la idea de que cada función debe tener un trabajo específico. La primera función imprime cada diseño, y la segunda muestra los modelos completados. Esto es más beneficioso que usar una función para hacer ambos trabajos. Si está escribiendo una función y se da cuenta de que la función está haciendo demasiadas tareas diferentes, intente dividir el código en dos funciones. Recuerde que siempre puede llamar a una función desde otra función, lo que puede ser

útil al dividir una tarea compleja en una serie de pasos.

Impedir que una función modifique una lista

A veces querrá evitar que una función modifique una lista. Por ejemplo, digamos que empiezas con una lista de diseños sin imprimir y escribes una función

para moverlos a una lista de modelos completados, como en el ejemplo anterior. Puede que decida que, aunque haya impreso todos los diseños, desea conservar la lista original de diseños sin imprimir para sus archivos. Pero como ha movido todos los nombres de los diseños fuera de `unprinted_designs`, la lista está ahora vacía, y la lista vacía es la única versión que tiene; la original ha desaparecido. En este caso, puede solucionar este problema pasando a la función una copia de la lista, no el original. Cualquier cambio que la función haga en la lista afectará sólo a la copia, dejando intacta la lista original.

Puede enviar una copia de una lista a una función como ésta:

```
nombre_funcion(nombre_lista[:])
```

La notación slice `[:]` hace una copia de la lista para enviarla a la función. Si no quisiéramos vaciar la lista de diseños no impresos en *print_models.py*, podríamos llamar a `print_models()` así:

```
print_models(diseños_no_impresos[:], modelos_completados)
```

La función `imprimir_modelos()` puede hacer su trabajo porque sigue recibiendo los nombres de todos los diseños no impresos. Pero esta vez utiliza una copia de la lista original de diseños no impresos, no la lista actual de `diseños no impresos`. La lista `modelos_completados` se llenará con los nombres de los modelos impresos como antes, pero la lista original de diseños no impresos no se verá afectada por la función.

Aunque puedes preservar el contenido de una lista pasando una copia de ella a tus funciones, deberías pasar la lista original a las funciones a menos que tengas una razón específica para pasar una copia. Es más eficiente que una función trabaje con una lista existente para evitar emplear el tiempo y la memoria necesarios para hacer una copia por separado, especialmente cuando se trabaja con listas grandes.

TRY IT yOURSELF

8-9. Magos: Haz una lista con los nombres de los magos. Pasa la lista a una función llamada `show_magicians()`, que imprime el nombre de cada mago de la lista.

8-10. Grandes Magos: Empiece con una copia de su programa del Ejercicio 8-9. Escriba una función llamada `make_great()` que modifique la lista de magos añadiendo la frase *the Great* al nombre de cada mago. Llame a `show_magicians()` para ver que la lista ha sido realmente modificada.

8-11. Magos sin cambios: Empiece con su trabajo del Ejercicio 8-10. Llame a la función `hacer_grande()` con una copia de la lista de nombres de magos. Como la lista original no se modificará, devuelva la nueva lista y guárdela en una lista separada. Llame a `show_magicians()` con cada lista para mostrar que tiene una lista con los nombres originales y otra con *el Grande* añadido al nombre de cada mago.

Pasar un número arbitrario de argumentos

A veces no sabrás de antemano cuántos argumentos necesita aceptar una función. Afortunadamente, Python permite que una función recoja un número arbitrario de argumentos de la sentencia que la llama.

Por ejemplo, considere una función que construye una pizza. Necesita aceptar un número de ingredientes, pero no se puede saber de antemano cuántos ingredientes querrá una persona. La función en el siguiente ejemplo tiene un parámetro, `*toppings`, pero este parámetro recoge tantos argumentos como la línea de llamada proporcione:

pizza.py

```
def hacer_pizza(*toppings):
    """Imprime la lista de coberturas que se han solicitado."""
    print(coberturas)

make_pizza('pepperoni')
make_pizza('champiñones', 'pimientos verdes', 'queso extra')
```

El asterisco en el nombre del parámetro `*toppings` le dice a Python que cree una tupla vacía llamada `toppings` y empaquete cualquier valor que reciba en esta tupla. La sentencia `print` en el cuerpo de la función produce una salida que muestra que Python puede manejar una llamada a la función con un valor y una llamada con tres valores. Trata las diferentes llamadas de forma similar. Observa que Python empaqueta los argumentos en una tupla, incluso si la función recibe sólo un valor:

```
('pepperoni',)
('champiñones', 'pimientos verdes', 'queso extra')
```

Ahora podemos sustituir la sentencia `print` por un bucle que recorra la lista de ingredientes y describa la pizza pedida:

```
def hacer_pizza(*toppings):
    """Resumir la pizza que vamos a hacer."""
    print("\nHacer una pizza con los siguientes
    ingredientes:") for topping in toppings:
        print("- " + topping)

make_pizza('pepperoni')
make_pizza('champiñones', 'pimientos verdes', 'queso extra')
```

La función responde adecuadamente, tanto si recibe un valor como tres:

```
Hacer una pizza con los siguientes ingredientes:
- pepperoni

Hacer una pizza con los siguientes ingredientes:
- setas
- pimientos verdes
- queso extra
```

Esta sintaxis funciona independientemente del número de argumentos que reciba la función.

Mezclar argumentos posicionales y arbitrarios

Si desea que una función acepte varios tipos de argumentos diferentes, el parámetro que acepta un número arbitrario de argumentos debe colocarse en último lugar en la definición de la función. Python empareja primero los argumentos posicionales y de palabra clave y luego recoge cualquier argumento restante en el parámetro final.

Por ejemplo, si la función necesita tomar un tamaño para la pizza, ese parámetro debe ir antes del parámetro `*toppings`:

```
def hacer_pizza(tamaño, *toppings):
    """Resume la pizza que vamos a hacer."""
    print("\nHaciendo una " + str(tamaño) +
          "-inch pizza with the following toppings:")
    para rellenar en toppings:
        print("- " + topping)

make_pizza(16, 'pepperoni')
make_pizza(12, 'champiñones', 'pimientos verdes', 'queso extra')
```

En la definición de la función, Python almacena el primer valor que recibe en el `tamaño` del parámetro. Todos los demás valores que vienen después se almacenan en la tupla `toppings`. Las llamadas a la función incluyen primero un argumento para el tamaño, seguido de tantos toppings 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:

```
Hacer una pizza de 16 pulgadas con los siguientes ingredientes:
- pepperoni

Hacer una pizza de 12 pulgadas con los siguientes ingredientes:
- setas
- pimientos verdes
- queso extra
```

Uso de argumentos arbitrarios para palabras clave

A veces querrá aceptar un número arbitrario de argumentos, p e r o no sabrá de antemano qué tipo de información se pasará a la función. En este caso, puede escribir funciones que acepten tantos pares clave-valor como proporcione la sentencia de llamada. Un ejemplo es la creación de perfiles de usuario: sabe que obtendrá información sobre un usuario, pero no está seguro de qué tipo de información recibirá. La función `build_profile()` del módulo

El siguiente ejemplo siempre toma un nombre y un apellido, pero también acepta un número arbitrario de argumentos de palabra clave:

perfil_usuario.py

```
def construir_perfil(primero, ultimo, **info_usuario):
    """Construir un diccionario que contenga todo lo que sabemos sobre un
    usuario."""
    perfil = {}
    profile['first_name'] = first
    profile['last_name'] = last
    para clave, valor en
        user_info.items():
        perfil[clave] = valor
    perfil de retorno

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='física')
print(perfil_usuario)
```

La definición de `build_profile()` espera un nombre y un apellido, y luego permite al usuario pasar tantos pares nombre-valor como quiera. Los asteriscos dobles antes del parámetro `**user_info` hacen que Python cree un diccionario vacío llamado `user_info` y empaquete cualquier par nombre-valor que reciba en este diccionario. Dentro de la función, puedes acceder a los pares nombre-valor en `user_info` como lo harías con cualquier diccionario.

En el cuerpo de `build_profile()`, creamos un diccionario vacío llamado `profile` para guardar el perfil del usuario. En ❶ añadimos el nombre y los apellidos a este diccionario porque siempre recibiremos estos dos datos del usuario. En ❷ recorremos los pares clave-valor adicionales del diccionario `user_info` y añadimos cada par al diccionario de `perfiles`. Por último, devolvemos el diccionario de `perfiles` 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'`. Almacenamos el `perfil` devuelto en `user_profile` e imprimimos `user_profile`:

```
{'first_name': 'albert', 'last_name': 'einstein',
'location': 'princeton', 'field': 'physics'}
```

El diccionario devuelto contiene el nombre y los apellidos del usuario y, en este caso, también la localidad y el campo de estudio. La función funcionaría sin importar cuántos pares clave-valor adicionales se proporcionen en la llamada a la función.

Puede mezclar valores posicionales, de palabra clave y arbitrarios de muchas maneras diferentes cuando escriba sus propias funciones. Es útil saber que existen todos estos tipos de argumentos porque los verás a menudo cuando empieces a leer el código de otras personas. Se necesita práctica para aprender a usar los diferentes tipos correctamente y para saber cuándo usar cada tipo. Por ahora, recuerda utilizar el enfoque más simple que haga el trabajo. A medida que progreses aprenderás a utilizar el enfoque más eficiente cada vez.

TRY IT yOURSELF

8-12. Bocadillos: Escribe una función que acepte una lista de elementos que una persona quiere en un sándwich. La función debe tener un parámetro que recoja tantos elementos como proporcione la llamada a la función, y debe imprimir un resumen del sándwich que se está pidiendo. Llama a la función tres veces, utilizando un número diferente de argumentos cada vez.

8-13. Perfil de usuario: Comience con una copia de *user_profile.py* de la página 153. Construir un perfil de ti mismo llamando a `build_profile()`, utilizando tu nombre y apellidos y otros tres pares clave-valor que te describan.

8-14. Coches: Escribe una función que almacene información sobre un coche en un diccionario. La función debe recibir siempre un fabricante y un nombre de modelo. A continuación, debe aceptar un número arbitrario de argumentos de palabras clave. Llame a la función con la información requerida y otros dos pares nombre-valor, como un color o una característica opcional. Su función debería funcionar para una llamada como ésta:

```
car = make_car('subaru', 'outback', color='azul', tow_package=True)
```

Imprima el diccionario devuelto para asegurarse de que toda la información se ha almacenado correctamente.

Almacenamiento de funciones en módulos

Una ventaja de las funciones es que separan bloques de código del programa principal. Utilizando nombres descriptivos para sus funciones, su programa principal será mucho más fácil de seguir. Puedes ir un paso más allá almacenando tus funciones en un archivo separado llamado *módulo* e *importando* ese módulo a tu programa principal. Una sentencia `import` le dice a Python que haga que el código de un módulo esté disponible en el archivo del programa que se está ejecutando en ese momento.

Almacenar las funciones en un archivo independiente le permite ocultar los detalles del código de su programa y centrarse en su lógica de alto nivel. También le permite reutilizar funciones en muchos programas diferentes. Cuando almacena sus funciones en archivos separados, puede compartir esos archivos con otros programadores sin tener que compartir todo su programa. Saber cómo importar funciones también le permite utilizar bibliotecas de funciones que otros programadores han escrito.

Hay varias formas de importar un módulo, y te mostraré cada una de ellas brevemente.

Importar un módulo completo

Para empezar a importar funciones, primero tenemos que crear un módulo. Un *módulo*

es un archivo que termina en *.py* y que contiene el código que desea importar a su archivo

programa. Hagamos un módulo que contenga la función make_pizza(). Para hacer este m ó d u l o , eliminaremos todo del archivo *pizza.py* excepto la función `make_pizza()`:

pizza.py

```
def hacer_pizza(tamaño, *toppings):
    """Resume la pizza que vamos a hacer."""
    print("\nHaciendo una " + str(tamaño) +
          "-inch pizza with the following toppings:")
    para rellenar en toppings:
        print("- " + topping)
```

Ahora crearemos un archivo separado llamado *making_pizzas.py* en el mismo directorio que *pizza.py*. Este archivo importa el módulo que acabamos de crear y luego hace dos llamadas a `make_pizza()`:

haciendo_pizzas.py

```
importar pizza

❶ pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'champiñones', 'pimientos verdes', 'queso extra')
```

Cuando Python lee este archivo, la línea `import pizza` le dice a Python que abra el archivo *pizza.py* y copie todas las funciones del mismo en este programa. En realidad no ves el código copiándose entre archivos porque Python copia el código entre bastidores mientras se ejecuta el programa. Todo lo que necesitas que hay que saber es que cualquier función definida en *pizza.*py estará ahora disponible en *haciendo_pizzas.py*.

Para llamar a una función desde un módulo importado, introduce el nombre del módulo que has importado, `pizza`, seguido del nombre de la función, `make_pizza()`, separados por un punto ❶. Este código produce la misma salida que el programa original que no importó ningún m ó d u l o :

```
Hacer una pizza de 16 pulgadas con los siguientes ingredientes:
- pepperoni

Hacer una pizza de 12 pulgadas con los siguientes ingredientes:
- setas
- pimientos verdes
- queso extra
```

Este primer enfoque de la importación, en el que simplemente escribes `import` seguido del nombre del módulo, hace que todas las funciones del módulo estén disponibles en tu programa. Si utilizas este tipo de sentencia `import` para importar un módulo completo llamado *nombre_del_módulo.py*, cada función del módulo estará disponible mediante la siguiente sintaxis:

```
module_name.function_name()
```

Importación de funciones específicas

También puede importar una función específica de un módulo. Aquí está la sintaxis general para este enfoque:

```
from nombre_del_módulo import nombre_de_la_función
```

Puede importar tantas funciones como desee de un módulo separando el nombre de cada función con una coma:

```
from nombre_del_módulo import función_0, función_1, función_2
```

El ejemplo *making_pizzas.py* quedaría así si queremos importar sólo la función que vamos a utilizar:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')
make_pizza(12, 'champiñones', 'pimientos verdes', 'queso extra')
```

Con esta sintaxis, no es necesario utilizar la notación de punto cuando se llama a una función. Como hemos importado explícitamente la función `make_pizza()` en la sentencia `import`, podemos llamarla por su nombre cuando utilicemos la función.

Uso de as para asignar un alias a una función

Si el nombre de una función que está importando puede entrar en conflicto con un nombre existente en su programa o si el nombre de la función es largo, puede utilizar un alias corto y único, *un* nombre alternativo similar a un apodo para la función. Este apodo especial se asigna a la función al importarla.

Aquí le damos a la función `make_pizza()` un alias, `mp()`, importando `make_pizza como mp`. La palabra clave `as` renombra una función usando el alias que le proporcionas:

```
from pizza import make_pizza as mp

mp(16, 'pepperoni')
mp(12, "champiñones", "pimientos verdes", "queso extra")
```

La sentencia `import` mostrada aquí renombra la función make_pizza() a `mp()` en este programa. Cada vez que queramos llamar a make_pizza() podemos simplemente escribir `mp()` en su lugar, y Python ejecutará el código en `make_pizza()` evitando cualquier confusión con otra función `make_pizza()` que puedas haber escrito en este archivo de programa.

La sintaxis general para proporcionar un alias es:

```
from nombre_del_módulo import nombre_de_la_función as fn
```

Uso de as para asignar un alias a un módulo

También puede proporcionar un alias para el nombre de un módulo. Dar a un módulo un alias corto, como `p` para `pizza`, permite llamar a las funciones del módulo más rápidamente. Llamar a `p.hacer_pizza()` es más conciso que llamar a `pizza.hacer_pizza()`:

```
import pizza as p

p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'champiñones', 'pimientos verdes', 'queso extra')
```

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 sólo es más conciso que escribir `pizza.make_pizza()`, sino que también desvía su atención del nombre del módulo y le permite
para centrarse en los nombres descriptivos de sus funciones. Estos nombres de función, que indican claramente lo que hace cada función, son más importantes para la legibilidad del código que utilizar el nombre completo del módulo.

La sintaxis general de este enfoque es:

```
importar nombre_módulo como mn
```

Importación de todas las funciones de un módulo

Puedes decirle a Python que importe todas las funciones de un módulo utilizando el operador asterisco (`*`):

```
from pizza import *

make_pizza(16, 'pepperoni')
make_pizza(12, 'champiñones', 'pimientos verdes', 'queso extra')
```

El asterisco en la sentencia `import` le dice a Python que copie cada función del módulo `pizza` en este archivo de programa. Como cada función es importada, puedes llamar a cada función por su nombre sin usar la notación punto. Sin embargo, es mejor no utilizar este método cuando se trabaja con módulos más grandes que no se han escrito: si el módulo tiene un nombre de función que coincide con un nombre existente en el proyecto, se pueden obtener algunos resultados inesperados. Python puede ver varias funciones o variables con el mismo nombre, y en lugar de importar todas las funciones por separado, sobreescribirá las funciones.

Lo mejor es importar la función o funciones que quieras, o importar todo el módulo y utilizar la notación por puntos. Esto conduce a un código claro que es fácil de leer y entender. Incluyo esta sección para que reconozcas declaraciones `import` como las siguientes cuando las veas en el código de otras personas:

```
from nombre_del_módulo import *
```

Funciones de estilismo

Debe tener en cuenta algunos detalles a la hora de diseñar funciones. Las funciones deben tener nombres descriptivos, y estos nombres deben usar letras minúsculas y guiones bajos. Los nombres descriptivos te ayudan a ti y a otros a entender lo que tu código está tratando de hacer. Los nombres de los módulos también deben utilizar estas convenciones.

Cada función debe tener un comentario que explique de forma concisa lo que hace la función. Este comentario debe aparecer inmediatamente después de la definición de la función y utilizar el formato docstring. En una función bien documentada, otros programadores pueden utilizar la función leyendo únicamente la descripción en el docstring. Deberían poder confiar en que el código funciona tal y como se describe, y siempre que conozcan el nombre de la función, los argumentos que necesita y el tipo de valor que devuelve, deberían poder utilizarla en sus programas.

Si especifica un valor por defecto para un parámetro, no deben utilizarse espacios a ambos lados del signo igual:

```
def nombre_funcion(parámetro_0, parámetro_1='valor por defecto')
```

La misma convención debe utilizarse para los argumentos de palabras clave en las llamadas a funciones:

```
nombre_funcion(valor_0, parametro_1='valor')
```

PEP 8 (https://www.python.org/dev/peps/pep-0008/) recomienda limitar las líneas de código a 79 caracteres para que cada línea sea visible 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 ENTER después del paréntesis de apertura en la línea de definición. En la línea siguiente, pulse TAB dos veces para separar la lista de argumentos del cuerpo de la función, que sólo se sangrará un nivel.

La mayoría de los editores alinean automáticamente las líneas adicionales de parámetros para que coincidan con la sangría establecida en la primera línea:

```
def nombre_funcion(
        parámetro_0, parámetro_1, parámetro_2,
        parámetro_3, parámetro_4, parámetro_5):
    función cuerpo...
```

Si su programa o módulo tiene más de una función, puede separar cada una de ellas con dos líneas en blanco para que sea más fácil ver dónde termina una función y empieza la siguiente.

Todas las sentencias `import` deben escribirse al principio del fichero. La única excepción es si utiliza comentarios al principio del archivo para describir el programa en general.

TRY IT yOURSELF

8-15. Impresión de modelos: Ponga las funciones para el ejemplo print_models.*py* en un archivo separado llamado *printing_functions.py*. Escriba una sentencia `import` en la parte superior de print_models.*py*, y modifique el archivo para utilizar las funciones importadas.

8-16. Importación: Usando un programa que escribiste que tiene una función en él, almacena esa función en un archivo separado. Importa la función a tu archivo de programa principal, y llama a la función usando cada uno de estos enfoques:

```
importar nombre_de_módulo
from nombre_del_módulo import nombre_de_la_función
from nombre_del_módulo import nombre_de_la_función as fn
importar nombre_módulo como mn
from nombre_del_módulo import *
```

8-17. Funciones de estilo: Elija tres programas cualesquiera que haya escrito para este capítulo y asegúrese de que siguen las directrices de estilo descritas en esta sección.

Resumen

En este capítulo aprendiste a escribir funciones y a pasar argumentos para que tus funciones tengan acceso a la información que necesitan para hacer su trabajo. Aprendiste cómo usar argumentos posicionales y de palabra clave, y cómo aceptar un número arbitrario de argumentos. Ha visto funciones que muestran la salida y funciones que devuelven valores. Ha aprendido a utilizar funciones con listas, diccionarios, sentencias `if` y bucles `while`. También vio cómo almacenar sus funciones en archivos separados llamados *módulos*, para que sus archivos de programa sean más simples y fáciles de entender. Por último, aprendió a dar estilo a sus funciones para que sus programas sigan estando bien estructurados y sean lo más fáciles de leer para usted y los demás.

Uno de tus objetivos como programador debería ser escribir código sencillo que haga lo que tú quieres, y las funciones te ayudan a conseguirlo. Te permiten escribir bloques de código y dejarlos en paz una vez que sabes que funcionan. Cuando sabes que una función hace su trabajo correctamente, puedes confiar en que seguirá funcionando y pasar a la siguiente tarea de codificación.

Las funciones permiten escribir código una sola vez y reutilizarlo tantas veces como se desee. Cuando necesites ejecutar el código de una función, todo lo que tienes que hacer es escribir una llamada de una sola línea y la función hará su trabajo. Cuando necesites modificar el comportamiento de una función, sólo tendrás que modificar un bloque de código, y tu cambio tendrá efecto en todos los lugares en los que hayas hecho una llamada a esa función.

El uso de funciones facilita la lectura de los programas, y los buenos nombres de funciones resumen lo que hace cada parte de un programa. La lectura de una serie de llamadas a funciones da una idea mucho más rápida de lo que hace un programa que la lectura de una larga serie de bloques de código.

Las funciones también facilitan la comprobación y depuración del código. Cuando la mayor parte del trabajo de tu programa lo realiza un conjunto de funciones, cada una de las cuales tiene una tarea específica, es mucho más fácil probar y mantener el código que has escrito. Puedes escribir un programa separado que llame a cada función y compruebe si cada función funciona en todas las situaciones que pueda encontrar. Al hacer esto, puedes estar seguro de que tus funciones funcionarán correctamente cada vez que las llames.

En el Capítulo 9 aprenderás a escribir clases. *Las clases* combinan funciones y datos en un paquete ordenado que puede utilizarse de forma flexible y eficiente.

9

CLASSES



La programación orientada a objetos es uno de los métodos más eficaces para escribir software. En la programación orientada a objetos

escribir *clases* que representen cosas y situaciones del mundo real, y crear *objetos* basados en estas clases.

clases. Cuando se escribe una clase, se define el comportamiento general que puede tener toda una categoría de objetos.

Cuando se crean objetos individuales a partir de la clase, cada objeto se equipa automáticamente con el comportamiento general; a continuación, puede dar a cada objeto los rasgos únicos que desee. Le sorprenderá lo bien que se pueden modelar situaciones del mundo real con la programación orientada a objetos.

Crear un objeto a partir de una clase se llama *instanciación, y se* trabaja con *instancias* de una clase. En este capítulo escribirás clases y crearás instancias de esas clases. Especificarás el tipo de información que se puede

almacenar en las instancias y definirás las acciones que se pueden realizar con estas instancias. También escribirás clases que extienden la funcionalidad de clases existentes, así que

clases similares pueden compartir código de forma eficiente. Almacenará sus clases en módulos e importará clases escritas por otros programadores en sus propios archivos de programa.

Comprender la programación orientada a objetos le ayudará a ver el mundo como lo ve un programador. Te ayudará a conocer realmente tu código, no solo lo que ocurre línea a línea, sino también los conceptos más amplios que hay detrás.

Conocer la lógica que hay detrás de las clases te capacitará para pensar de forma lógica, de modo que puedas escribir programas que aborden con eficacia casi cualquier problema que encuentres.

Las clases también te facilitan la vida a ti y a los demás programadores con los que tendrás que trabajar a medida que te enfrentes a retos cada vez más complejos. Cuando tú y otros programadores escribáis código basado en el mismo tipo de lógica, podréis entender el trabajo de los demás. Tus programas tendrán sentido para muchos colaboradores, lo que permitirá a todos lograr más.

Crear y utilizar una clase

Se puede modelar casi cualquier cosa utilizando clases. Empecemos escribiendo una clase sencilla, `Perro`, que representa a un perro, no a un perro en particular, sino a cualquier perro.

¿Qué sabemos de la mayoría de los perros? Bueno, todos tienen nombre y edad. También sabemos que la mayoría de los perros se sientan y se dan la vuelta. Esos dos datos (nombre y edad) y esos dos comportamientos (sentarse y darse la vuelta) 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 a un perro. Una vez escrita nuestra clase, la utilizaremos para crear instancias individuales, cada una de las cuales representará a un perro específico.

Creación de la clase Perro

Cada instancia creada a partir de la clase `Perro` almacenará un `nombre` y una `edad`, y daremos a cada perro la capacidad de `sentarse()` y `darse la vuelta()`:

perro.py

```
❶ clase Perro():
❷     """Un simple intento de modelar un perro""".

❸     def init (self, nombre, edad):
          """Inicializar atributos de nombre y edad."""
❹         self.nombre =
          nombre self.edad
          = edad

❺     def sit(self):
          """Simular que un perro se sienta en respuesta a una orden."""
          print(self.name.title() + " ahora está sentado.")
```

```
    def roll_over(self):
        """Simular un vuelco en respuesta a una orden."""
        print(self.name.title() + " ¡volcó!")
```

Hay mucho que notar aquí, pero no te preocupes. Verás esta estructura a lo largo de este capítulo y tendrás mucho tiempo para acostumbrarte a ella. En ❶ definimos una clase llamada `Perro`. Por convención, los nombres en mayúsculas se refieren a clases en Python. Los paréntesis en la definición de la clase están vacíos porque estamos creando esta clase desde cero. En ❷ escribimos un docstring describiendo lo que hace esta clase.

El método init ()

Una función que forma parte de una clase es un *método*. Todo lo que aprendiste sobre las funciones se aplica también a los métodos; la única diferencia práctica por ahora es la forma en que llamaremos a los métodos. El método `init ()` en ❸ es un método especial que Python ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase `Perro`. Este método tiene dos guiones bajos al principio y dos guiones bajos al final, una convención que ayuda a prevenir que los nombres de métodos por defecto de Python entren en conflicto con tus nombres de métodos.

Definimos el método `init ()` para que tenga tres parámetros: `self`, `name` y `age`. El parámetro `self` es necesario en la definición del método, y debe ir primero antes que los otros parámetros. Debe incluirse en la definición porque cuando Python llame a este método `init ()` más tarde (para crear una instancia de `Dog`), la llamada al método pasará automáticamente el argumento `self`.

Cada llamada a un método asociado a una clase pasa automáticamente `self`, que es una referencia a la instancia en sí misma; da acceso a la instancia individual a los atributos y métodos de la clase. Cuando creamos una instancia de `Dog`, Python llamará al método `init ()` de la clase `Dog`. Pasaremos Dog()
un nombre y una edad como argumentos; `self` se pasa automáticamente, por lo que no necesitamos pasarlo. Siempre que queramos crear una instancia de la clase `Dog`, proporcionaremos valores sólo para los dos últimos parámetros, `nombre` y `edad`.

Las dos variables definidas en ❹ tienen cada una el prefijo `self`. Cualquier variable prefijada con `self` está disponible para todos los métodos de la clase, y también podremos acceder a estas variables a través de cualquier instancia creada a partir de la clase. `self.nombre = nombre` toma el valor almacenado en el parámetro `nombre` y lo almacena
en el `nombre de` la variable, que luego se adjunta a la instancia que se está creando. El mismo proceso ocurre con `self.age = edad`. Las variables accesibles a través de instancias como ésta se denominan *atributos*.

La clase `Dog` tiene otros dos métodos definidos: `sit()` y `roll_over()` ❺.

Como estos métodos no necesitan información adicional como nombre o edad, sólo los definimos para que tengan un parámetro, `self`. Las instancias que creemos más tarde tendrán acceso a estos métodos. En otras palabras, podrán sentarse y darse la vuelta. Por ahora, `sit()` y `roll_over()` no hacen mucho. Simplemente imprimen un mensaje diciendo que el perro se sienta o se da la vuelta. Pero el concepto puede extenderse a situaciones realistas: si esta clase fuera parte de un juego de

ordenador real, estos métodos contendrían código para hacer que un perro animado se sentara y se diera la vuelta. Si esta clase se escribiera para controlar un robot, estos métodos dirigirían los movimientos que hacen que un perro robot se siente y se dé la vuelta.

Creación de clases en Python 2.7

Cuando creas una clase en Python 2.7, necesitas hacer un pequeño cambio. Se incluye el término `objeto` entre paréntesis cuando se crea una clase:

```
clase NombreClase(objeto):
    --snip--
```

Esto hace que las clases de Python 2.7 se comporten más como clases de Python 3, lo que facilita el trabajo en general.

La clase `Dog` se definiría así en Python 2.7:

```
clase Perro(objeto):
    --snip--
```

Crear una instancia a partir de una clase

Piensa en una clase como un conjunto de instrucciones sobre cómo crear una instancia. La clase `Perro` es un conjunto de instrucciones que le dice a Python cómo crear instancias individuales que representan perros específicos.

Hagamos una instancia que represente a un perro concreto:

```
clase Perro():
    --snip--

❶ mi_perro = Perro('willie', 6)

❷ print("El nombre de mi perro es " + mi_perro.nombre.titulo() + ".")
❸ print("Mi perro tiene " + str(mi_perro.edad) + " años.")
```

La clase `Dog` que estamos usando aquí es la que acabamos de escribir en el ejemplo anterior. En ❶ le decimos a Python que cree un perro cuyo nombre sea `'willie'` y cuya edad sea `6`. 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 establece los atributos `nombre` y `edad` utilizando los valores que proporcionamos. El método `init (_)` no tiene una sentencia `return` explícita, pero Python devuelve automáticamente una instancia que representa a este perro. Almacenamos esa instancia en la variable `mi_perro`. La convención de nombres es útil aquí: normalmente podemos asumir que un nombre en mayúsculas como `Dog` se refiere a una clase, y un nombre en minúsculas como `my_dog` se refiere a una única instancia creada a partir de una clase.

Acceso a los atributos

Para acceder a los atributos de una instancia, se utiliza la notación de puntos. En ❷ accedemos al valor del `nombre del` atributo de `mi_perro` escribiendo:

```
mi_perro.nombre
```

La notación con puntos se utiliza a menudo en Python. Esta sintaxis demuestra cómo Python encuentra el valor de un atributo. Aquí Python busca en la instancia `mi_perro`

y busca el atributo `name` asociado a `my_dog`. Este es el mismo atri- buto referido como `self.name` en la clase `Dog`. En ❸ utilizamos el mismo enfoque para trabajar con el atributo `edad`. En nuestra primera sentencia `print`, `my_dog.name.title()` hace que `'willie'`, el valor del atributo `name` de `my_dog`, empiece con mayúscula. En la segunda sentencia de impresión, `str(mi_perro.edad)` convierte `6`, el valor del atributo `edad` de `mi_perro`, en una cadena.

El resultado es un resumen de lo que sabemos sobre `mi_perro`:

```
Mi perro se llama Willie.
Mi perro tiene 6 años.
```

Métodos de llamada

Después de crear una instancia de la clase `Perro`, podemos utilizar la notación punto para llamar a cualquier método definido en `Perro`. Hagamos que nuestro perro se siente y se dé la vuelta:

```
clase Perro():
    --snip--

mi_perro = Dog('willie', 6)
mi_perro.sit()
mi_perro.roll_over()
```

Para llamar a un método, da el nombre de la instancia (en este caso, `mi_perro`) y el método que quieres llamar, separados por un punto. Cuando Python lee `mi_perro.sentarse()`, busca el método `sentarse()` en la clase `Perro` y ejecuta ese código. Python interpreta la línea `mi_perro.voltear()` de la misma manera.

Ahora Willie hace lo que le decimos:

```
Willie está ahora
sentado. ¡Willie se ha
dado la vuelta!
```

Esta sintaxis es bastante útil. Cuando a los atributos y métodos se les ha dado nombres descriptivos apropiados como `name`, `age`, `sit()`, y `roll_over()`, podemos deducir fácilmente lo que se supone que debe hacer un bloque de código, incluso uno que nunca hemos visto antes.

Creación de varias instancias

Puedes crear tantas instancias de una clase como necesites. Vamos a crear un segundo perro llamado `tu_perro`:

```
clase Perro():
    --snip--

mi_perro =
Perro('willie', 6)
tu_perro = Perro('lucy',
```

```
3)

print("Mi perro se llama " + my_dog.name.title() + ".")
print("Mi perro tiene " + str(my_dog.age) + " años.")
my_dog.sit()
```

```
print("Tu perro se llama " + tu_perro.nombre.titulo() + ".")
print("Tu perro tiene " + str(tu_perro.edad) + " años.")
your_dog.sit()
```

En este ejemplo creamos un perro llamado Willie y otro llamado Lucy. Cada perro es una instancia independiente con su propio conjunto de atributos, capaz del mismo conjunto de acciones:

```
Mi perro se llama
Willie. Mi perro tiene 6
años. Willie está ahora
sentado.

Tu perra se llama Lucy.
Tu perro tiene 3 años.
Lucy está ahora sentada.
```

Incluso si utilizáramos el mismo nombre y edad para el segundo perro, Python crearía una instancia separada de la clase `Perro`. Puedes crear tantas instancias de una clase como necesites, siempre que des a cada instancia un nombre de variable único o que ocupe un lugar único en una lista o diccionario.

TRY IT yOURSELF

9-1. Restaurante: Crea una clase llamada `Restaurante`. El método `init ()` de `Restaurant` debe almacenar dos atributos: un `nombre_restaurante` y un `tipo_de_cocina`. Crea un método llamado `describe_restaurant()` que imprima estas dos informaciones, y un método llamado `open_restaurant()` que imprima un mensaje indicando que el restaurante está abierto.

Crea una instancia llamada `restaurante` a partir de tu clase. Imprime los dos atributos individualmente y, a continuación, llama a ambos métodos.

9-2. Tres Restaurantes: Comience con su clase del Ejercicio 9-1. Cree tres instancias diferentes de la clase, y llame a `describe_restaurant()` para cada instancia.

9-3. Usuarios: Crea una clase llamada `Usuario`. Cree dos atributos llamados `first_name` y `last_name`, y luego cree varios otros atributos que son típicamente almacenados en un perfil de usuario. Crea un método llamado `describe_user()` que imprima un resumen de la información del usuario. Crea otro método llamado `greet_user()` que imprima un saludo personalizado al usuario.

Cree varias instancias que representen a diferentes usuarios y llame a ambos métodos para cada usuario.

Trabajar con clases e instancias

Puede utilizar clases para representar muchas situaciones del mundo real. Una vez que escribas una clase, pasarás la mayor parte del tiempo trabajando con instancias creadas a partir de esa clase. Una de las primeras tareas que querrás hacer es modificar los atributos asociados con una instancia en particular. Puedes modificar los atributos de una instancia directamente o escribir métodos que actualicen los atributos de formas específicas.

La clase automovilística

Escribamos una nueva clase que represente a un coche. Nuestra clase almacenará información sobre el tipo de coche con el que estamos trabajando, y tendrá un método que resuma esta información:

coche.py

```
clase Coche():
    "Un simple intento de representar un coche".

❶   def init (self, marca, modelo, año):
        """Inicializar atributos para describir un
        coche.""" self.make = make
        self.modelo = modelo
        self.año = año

❷   def obtener_nombre_descriptivo(self):
        """Devuelve un nombre descriptivo bien formateado."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

❸ mi_nuevo_coche = Coche('audi', 'a4', 2016)
  print(mi_nuevo_coche.get_descriptive_name())
```

En ❶ la clase `Coche`, definimos primero el método `init ()` con el parámetro `self`, igual que hicimos antes con nuestra clase `Perro`. También le damos otros tres parámetros: `marca`, `modelo` y `año`. El método `init ()` toma estos parámetros y los almacena en los atributos que serán asociados a las instancias creadas a partir de esta clase. Cuando creamos un nuevo `coche` necesitaremos especificar una marca, modelo y año para nuestra instancia.

En ❷ definimos un método llamado `get_descriptive_name()` que pone el `año`, la `marca` y el `modelo de` un coche en una cadena que describe claramente el coche. Esto nos evitará tener que imprimir el valor de cada atributo individualmente. Para trabajar con los valores de los atributos en este método, utilizamos `self.make`, `self.model` y `self.year`. En ❸ hacemos una instancia de la clase `Coche` y la almacenamos en la variable `mi_coche_nuevo`. Luego llamamos a `get_descriptive_name()` para mostrar qué clase de coche que tenemos:

```
Audi A4 2016
```

Para hacer la clase más interesante, vamos a añadir un atributo que cambie con el tiempo. Añadiremos un atributo que almacene el kilometraje total del coche.

Establecer un valor por defecto para un atributo

Cada atributo en una clase necesita un valor inicial, incluso si ese valor es 0 o una cadena vacía. En algunos casos, como cuando se establece un valor por defecto, tiene sentido especificar este valor inicial en el cuerpo del método `init ()`; si haces esto para un atributo, no tienes que incluir un parámetro para ese atributo.

Vamos a añadir un atributo llamado `lectura_cuentakilómetros` que siempre empieza con el valor 0. También añadiremos un método `leer_cuentakilómetros()` que nos ayude a leer el cuentakilómetros de cada coche:

```
clase Coche():

    def init (self, marca, modelo, año):
        """Inicializar atributos para describir un
        coche.""" self.marca = marca
        self.modelo = modelo
        self.año = año
❶       self.lectura_cuentakilómetros = 0

    def obtener_nombre_descriptivo(self):
        --snip--

❷   def leer_odometro(self):
        """Imprime un extracto con el kilometraje del coche."""
        print("Este coche tiene " + str(lectura_cuentakilómetros.self) + "
        kilómetros.")

mi_nuevo_coche = Coche('audi', 'a4', 2016)
print(mi_nuevo_coche.get_descriptive_name())
mi_nuevo_coche.read_odometer()
```

Esta vez cuando Python llama al método `init ()` para crear una nueva instancia, almacena los valores de marca, modelo y año como atributos como hizo en el ejemplo anterior. A continuación, Python crea un nuevo atributo llamado `lectura_cuentakilómetros` y establece su valor inicial en 0 ❶. También tenemos un atributo nuevo método llamado `read_odometer()` en ❷ que facilita la lectura de un coche. kilometraje.

Nuestro coche comienza con un kilometraje de 0:

```
Audi A4 2016
Este coche tiene 0 millas en él.
```

No se venden muchos coches con exactamente 0 millas en el cuentakilómetros, por lo que necesitamos una forma de cambiar el valor de este atributo.

Modificación de los valores de los atributos

Puedes cambiar el valor de un atributo de tres maneras: puedes cambiar el

valor directamente a través de una instancia, establecer el valor a través de un método, o incrementar el valor (añadirle una cierta cantidad) a través de un método. Veamos cada uno de estos enfoques.

Modificación directa del valor de un atributo

La forma más sencilla de modificar el valor de un atributo es acceder a él directamente a través de una instancia. Aquí fijamos directamente la lectura del cuentakilómetros en 23:

```
clase Coche():
    --snip--

mi_nuevo_coche = Coche('audi', 'a4', 2016)
print(mi_nuevo_coche.get_descriptive_name())

❶ mi_nuevo_coche.lectura_cuentakilómetros = 23
mi_nuevo_coche.leer_cuentakilómetros()
```

En ❶ usamos la notación de puntos para acceder al atributo `lectura_cuentakilómetros` del `coche` y establecer su valor directamente. Esta línea le dice a Python que tome la instancia `mi_coche_nuevo`, busque el atributo `lectura_cuentakilómetros` asociado a él y establezca el valor de ese atributo en 23:

```
Audi A4 2016
Este coche tiene 23 millas en él.
```

A veces querrás acceder a los atributos directamente de esta manera, pero otras veces querrás escribir un método que actualice el valor por ti.

Modificar el valor de un atributo mediante un método

Puede ser útil tener métodos que actualicen ciertos atributos por ti. En lugar de acceder directamente al atributo, pasas el nuevo valor a un método que se encarga internamente de la actualización.

Aquí hay un ejemplo que muestra un método llamado `update_odometer()`:

```
clase Coche():
    --snip--

❶     def actualizar_odometro(self, kilometraje):
        """Establece la lectura del cuentakilómetros en el valor dado."""
        self.odometer_reading = mileage

mi_nuevo_coche = Coche('audi', 'a4', 2016)
print(mi_nuevo_coche.get_descriptive_name())

❷ mi_coche_nuevo.actualizar_odometro(23)
mi_coche_nuevo.leer_odometro()
```

La única modificación a `Car` es la adición de `update_odometer()` en ❶. Este método toma un valor de kilometraje y lo almacena en `self.odometer_reading`. En ❷ llamamos a update_odometer() y le damos 23 como argumento (correspondiente a

al parámetro `kilometraje` en la definición del método). Establece la lectura del cuentakilómetros en 23, y `read_odometer()` imprime la lectura:

```
Audi A4 2016
Este coche tiene 23 millas en él.
```

Podemos extender el método `update_odometer()` para que realice un trabajo adicional cada vez que se modifique la lectura del cuentakilómetros. Vamos a añadir un poco de lógica para asegurarnos de que nadie intenta hacer retroceder la lectura del cuentakilómetros:

```
clase Coche():
    --snip--

    def actualizar_odometro(self,
        kilometraje): """
        Ajuste la lectura del cuentakilómetros al valor dado.
        Rechaza el cambio si intenta hacer retroceder el
        cuentakilómetros. """
❶       si kilometraje >= auto.lectura_cuentakilómetros:
            self.lectura_cuentakilómetros =
        kilometraje else:
❷           print("¡No se puede hacer retroceder un cuentakilómetros!")
```

Ahora `update_odometer()` comprueba que la nueva lectura tiene sentido antes de modificar el atributo. Si el nuevo `kilometraje`, `kilometraje`, es mayor o igual que el kilometraje existente, `self.lectura_odometro`, puedes actualizar la lectura del odómetro al nuevo kilometraje ❶. Si el nuevo kilometraje es menor que el kilometraje existente, obtendrás una advertencia de que no puedes retroceder un cuentakilómetros ❷.

Incrementar el valor de un atributo a través de un método

A veces querrá incrementar el valor de un atributo en una cantidad determinada en lugar de establecer un valor completamente nuevo. Digamos que compramos un coche usado y le ponemos 100 millas entre el momento en que lo compramos y el momento en que lo registramos. Aquí tenemos un método que nos permite pasar esta cantidad incremental y añadir ese valor a la lectura del cuentakilómetros:

```
clase Coche():
    --snip--

    def actualizar_odometro(self, kilometraje):
        --snip--

❶   def incremento_odometro(self, millas):
        """Añade la cantidad dada a la lectura del cuentakilómetros."""
        self.lectura_cuentakilómetros += millas

❷ mi_coche_usado = Coche('subaru', 'outback', 2013)
```

```
print(mi_coche_usado.get_descriptive_name())

❸ mi_coche_usado.actualizar_odometro(23500)
mi_coche_usado.leer_odometro()
```

```
❹ mi_coche_usado.increment_odometer(100)
  mi_coche_usado.read_odometer()
```

El nuevo método `increment_odometer()` en ❶ toma un número de millas, y añade este valor a `self.odometer_reading`. En ❷ creamos un coche usado, `mi_coche_usado`. Ponemos su cuentakilómetros en 23.500 llamando a `update_odometer()` y pasándole `23500` en ❸. En ❹ llamamos a `increment_odometer()` y le pasamos `100` para añadir los 160 kilómetros que recorrimos entre la compra del coche y su matriculación:

```
Subaru Outback 2013
Este coche tiene 23500 millas
en él. Este coche tiene 23600
millas en él.
```

Puede modificar fácilmente este método para rechazar incrementos negativos, de modo que nadie utilice esta función para hacer retroceder un cuentakilómetros.

NOTE *Puede utilizar métodos como éste para controlar cómo los usuarios de su programa actualizan valores como la lectura de un cuentakilómetros, pero cualquiera con acceso al programa puede establecer la lectura del cuentakilómetros en cualquier valor accediendo directamente al atributo. La seguridad efectiva requiere una atención extrema a los detalles, además de comprobaciones básicas como las que se muestran aquí.*

TRY IT yOURSELF

9-4. Número servido: Empiece con su programa del Ejercicio 9-1 (página 166). Añada un atributo llamado `número_servido` con un valor por defecto de 0. Cree una instancia llamada `restaurante` a partir de esta clase. Imprima el número de clientes que el restaurante ha servido, y luego cambie este valor e imprímalo de nuevo.

Añade un método llamado `set_number_served()` que te permita establecer el número de clientes que han sido servidos. Llama a este método con un nuevo número e imprime el valor de nuevo.

Añade un método llamado `increment_number_served()` que te permita incrementar el número de clientes que han sido atendidos. Llama a este método con cualquier número que quieras que pueda representar cuántos clientes fueron atendidos en, digamos, un día de negocio.

9-5. Intentos de Inicio de Sesión: Añada un atributo llamado `login_attempts` a su clase `User` del Ejercicio 9-3 (página 166). Escribe un método llamado `increment_ login_attempts()` que incremente el valor de `login_attempts` en 1. Escribe otro método llamado `reset_login_attempts()` que restablezca el valor de `login_attempts` a 0.

Crea una instancia de la clase `User` y llama a `increment_login_attempts()` varias veces. Imprime el valor de `login_attempts` para asegurarte de que se ha incrementado correctamente, y luego llama a `reset_login_attempts()`. Vuelve a imprimir `login_attempts para asegurarte de que se ha` reiniciado a 0.

Herencia

No siempre tienes que empezar de cero cuando escribes una clase. Si la clase que estás escribiendo es una versión especializada de otra clase que escribiste, puedes utilizar la *herencia*. Cuando una clase *hereda* de otra, adopta automáticamente todos los atributos y métodos de l a primera. La clase original s e denomina *clase padre,* y la nueva clase es la *clase hija*. La clase hija hereda todos los atributos y métodos de su clase padre, pero también es libre de definir nuevos atributos y métodos propios.

El método init () para una clase hija

La primera tarea de Python al crear una instancia a partir de una clase hija es asignar valores a todos los atributos de la clase padre. Para ello, el método `init ()` de una clase hija necesita la ayuda de su clase padre.

Por ejemplo, modelemos un coche eléctrico. Un coche eléctrico es sólo un tipo específico de coche, así que podemos basar nuestra nueva clase `ElectricCar` en la clase `Car` que escribimos anteriormente. Así sólo tendremos que escribir código para los atributos y comportamientos específicos de los coches eléctricos.

Empecemos haciendo una versión simple de la clase `CocheEléctrico`, que hace todo lo que hace la clase `Coche`:

electric_car.py

```
❶ class Coche():
    "Un simple intento de representar un coche".

    def init (self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.lectura_cuentakilóme
        tros = 0

    def obtener_nombre_descriptivo(self):
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def leer_odometro(self):
        print("Este coche tiene " + str(lectura_cuentakilómetros.self) + "
        kilómetros.")

    def actualizar_odometro(self, kilometraje):
        si cuentakilómetros >=
            cuentakilómetros_propio:
            cuentakilómetros_propio =
            cuentakilómetros
        si no:
            print("¡No se puede hacer retroceder un cuentakilómetros!")

    def incremento_odometro(self, millas):
        self.lectura_odometro += millas

❷ clase Cocheeléctrico(Coche):
```

```
    """Representar aspectos de un coche, específicos de los vehículos
    eléctricos""".
```

```
❸     def init (self, marca, modelo, año): """Inicializar
          atributos de la clase padre""".
❹         super(). init (marca, modelo, año)


❺ mi_tesla = Coche_eléctrico('tesla', 'modelo s',
2016) print(mi_tesla.get_descriptive_name())
```

En ❶ empezamos con `Car`. Cuando se crea una clase hija, la clase padre debe formar parte del fichero actual y debe aparecer antes que la clase hija en el fichero. En ❷ definimos la clase hija, `CocheEléctrico`. El nombre de la clase padre debe incluirse entre paréntesis en la definición de la clase hija. El método `init ()` de ❸ toma la información necesaria para hacer un `Coche` instancia.

La función `super()` en ❹ es una función especial que ayuda a Python a hacer conexiones entre la clase padre y la clase hija. Esta línea le dice a Python que llame al método `init ()` de la clase padre de `ElectricCar`, que le da a una instancia de `ElectricCar` todos los atributos de su clase padre. El nombre *super* viene de la convención de llamar a la clase padre *superclase* y a la clase hija *subclase*.

Probamos si la herencia funciona correctamente intentando crear un coche eléctrico con el mismo tipo de información que proporcionaríamos al hacer un coche normal. En ❺ hacemos una instancia de la clase `ElectricCar`, y la almacenamos en `my_tesla`. Esta línea llama al método `init ()` definido en `ElectricCar`, que a su vez le dice a Python que llame al método `init ()` definido en la clase particular `Car`. Proporcionamos los argumentos `'tesla'`, `'model s'`, y `2016`.

Aparte de `init ()`, todavía no hay atributos o métodos particulares de un coche eléctrico. En este momento sólo nos estamos asegurando de que el coche eléctrico tiene los comportamientos apropiados de `Car`:

```
Tesla Model S 2016
```

La instancia `ElectricCar` funciona igual que una instancia de `Car`, así que ahora podemos empezar a definir atributos y métodos específicos para los coches eléctricos.

Herencia en Python 2.7

En Python 2.7, la herencia es ligeramente diferente. La clase `ElectricCar` tendría este aspecto:

```
clase Coche(objeto):
    def init (self, marca, modelo, año):
        --snip--

clase Cocheeléctrico(Coche):
    def init (self, marca, modelo, año):
        super(Cocheeléctrico, self). init (marca, modelo,
        año)
        --snip--
```

La función `super()` necesita dos argumentos: una referencia a la clase hija y el objeto `self`. Estos argumentos son necesarios para ayudar a Python a establecer conexiones adecuadas entre las clases padre e hija. Cuando utilices la herencia en Python 2.7, asegúrate de definir la clase padre utilizando también la sintaxis de `objeto`.

Definición de atributos y métodos para la clase hija

Una vez que se tiene una clase hija que hereda de una clase padre, se pueden añadir los atributos y métodos necesarios para diferenciar la clase hija de la clase padre.

Vamos a añadir un atributo específico de los coches eléctricos (una batería, por ejemplo) y un método para informar sobre este atributo. Almacenaremos el tamaño de la batería y escribiremos un método que imprima una descripción de la batería:

```
clase Coche():
    --snip--

clase Cocheeléctrico(Coche):
    """Representar aspectos de un coche, específicos de los vehículos
    eléctricos""".

    def init (self, marca, modelo, año):
        """
        Inicializar atributos de la clase padre.
        A continuación, inicializa los atributos específicos de
        un coche eléctrico. """
        super(). init (marca, modelo, año)
❶       self.tamaño_pila = 70

❷   def describir_pila(self):
        """Imprime una declaración que describa el tamaño de la pila."""
        print("Este coche tiene una batería de " + str(self.battery_size) + "-
        kWh.")

my_tesla = Coche eléctrico('tesla', 'model s',
2016) print(my_tesla.get_descriptive_name())
my_tesla.describe_battery()
```

En ❶ añadimos un nuevo atributo `self.battery_size` y establecemos su valor inicial en, digamos, `70`. Este atributo se asociará a todas las instancias creadas a partir de la clase `ElectricCar` pero no se asociará a ninguna instancia de `Car`. También añadimos un método llamado `describir_batería()` que imprime información sobre la batería en ❷. Cuando llamamos a este método, obtenemos una descripción que es claramente específica de un coche eléctrico:

```
Tesla Model S 2016
Este coche tiene una batería de 70 kWh.
```

No hay límite a la especialización de la clase `ElectricCar`. Puedes

añadir tantos atributos y métodos como necesites para modelar un coche eléctrico con el grado de precisión que necesites. Un atributo o método que podría pertenecer a cualquier coche, en lugar de uno que es específico de un coche eléctrico, es un atributo o método que podría pertenecer a cualquier coche, en lugar de uno que es específico de un coche eléctrico.

debería añadirse a la clase Coche en lugar de a la clase CocheEléctrico. Así, cualquiera que utilice la clase Coche tendrá también disponible esa funcionalidad, y la clase CocheEléctrico sólo contendrá código para la información y el comportamiento específicos de los vehículos eléctricos.

Sobreescritura de métodos de la clase padre

Puede anular cualquier método de la clase padre que no se ajuste a lo que está intentando modelar con la clase hija. Para hacer esto, define un método en la clase hija con el mismo nombre que el método que quieres anular en la clase padre. Python ignorará el método de la clase padre y sólo prestará atención al método que definas en la clase hija.

Digamos que la clase Coche tiene un método llamado llenar_depósito(). Este método no tiene sentido para un vehículo totalmente eléctrico, por lo que es posible que desee anular este método. Esta es una forma de hacerlo:

```
def Cocheeléctrico(Coche):
    --snip--

    def llenar_tanque_gas():
        """Los coches eléctricos no tienen
        depósito de gasolina.""" print("¡Este
        coche no necesita depósito de gasolina!")
```

Ahora si alguien intenta llamar a llenar_depósito() con un coche eléctrico, Python ignorará el método llenar_depósito() en Coche y ejecutará este código en su lugar. Cuando usas herencia, puedes hacer que tus clases hijas retengan lo que necesitas y anulen cualquier cosa que no necesites de la clase padre.

Instancias como atributos

Al modelar algo del mundo real en código, puede que te des cuenta de que vas añadiendo cada vez más detalles a una clase. Encontrarás que tienes una lista creciente de atributos y métodos y que tus archivos se están volviendo largos. En estas situaciones, podrías reconocer que parte de una clase puede ser escrita como una clase separada. Puedes dividir tu clase grande en clases más pequeñas que trabajen juntas.

Por ejemplo, si seguimos añadiendo detalles a la clase ElectricCar, podríamos darnos cuenta de que estamos añadiendo muchos atributos y métodos específicos de la batería del coche. Cuando veamos que esto ocurre, podemos parar y mover esos
a una clase separada llamada Batería. Entonces podemos utilizar una clase Instancia de batería como atributo en la clase ElectricCar:

```
clase Coche():
    --snip--

❶ clase Batería():
    """Un simple intento de modelar una batería para un coche eléctrico""".
```

```
❷     def init (self, tamaño_pila=70): """Inicializa
          los atributos de la batería."""
          self.battery_size = battery_size.
```

```
❸     def describir_pila(self):
          """Imprime una declaración que describa el tamaño de la pila."""
          print("Este coche tiene una batería de " + str(self.battery_size) + "-
          kWh.")


clase Cocheeléctrico(Coche):
    """Representar aspectos de un coche, específicos de los vehículos
    eléctricos""".

    def init (self, marca, modelo, año):
        """
        Inicializar atributos de la clase padre.
        A continuación, inicializa los atributos específicos de
        un coche eléctrico. """
        super(). init (marca, modelo, año)
❹       self.battery = Batería()


mi_tesla = Coche eléctrico('tesla', 'modelo s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

En ❶ definimos una nueva clase llamada `Batería` que no hereda de ninguna otra clase. El método `init ()` en ❷ tiene un parámetro, `battery_size`, además de `self`. Este es un parámetro opcional que establece el tamaño de la batería a 70 si no se proporciona ningún valor. El método `describe_pila()` también se ha trasladado a esta clase ❸.

En la clase `ElectricCar`, ahora añadimos un atributo llamado self.`battery` ❹. Esta línea le dice a Python que cree una nueva instancia de `Batería` (con un tamaño por defecto de 70, porque no estamos especificando un valor) y almacene esa instancia en el atributo self.`battery`. Esto ocurrirá cada vez que el método `init ()`
cualquier instancia de `Coche Eléctrico` tendrá una instancia de `Batería` creada automáticamente.

Creamos un coche eléctrico y lo almacenamos en la variable `mi_tesla`. Cuando queramos describir la batería, tendremos que trabajar a través del atributo `battery` del coche:

```
my_tesla.battery.describe_battery()
```

Esta línea le dice a Python que busque la instancia `my_tesla`, encuentre su atributo `battery`, y llame al método `describe_battery()` que está asociado con la instancia `Battery` almacenada en el atributo.

El resultado es idéntico al que vimos anteriormente:

```
Tesla Model S 2016
Este coche tiene una batería de 70 kWh.
```

Esto parece mucho trabajo extra, pero ahora podemos describir la batería con tanto detalle como queramos sin saturar la clase `ElectricCar`. Añadamos otro método a `Battery` que informe de la autonomía del coche en función del tamaño de la batería:

```
clase Coche():
    --snip--

clase Batería():
    --snip--

❶   def get_range(self):
        """Imprime una declaración sobre la autonomía que proporciona
       esta batería.""" if self.battery_size == 70:
            alcance = 240
        elif self.battery_size == 85:
            range = 270

        message = "Este coche puede recorrer aproximadamente " +
        str(range) message += " millas con una carga completa."
        print(mensaje)

clase Cocheeléctrico(Coche):
    --snip--

my_tesla = Coche eléctrico('tesla', 'model s',
2016) print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

El nuevo método `get_range()` en ❶ realiza algunos análisis sencillos. Si la capacidad de la batería es de 70 kWh, `get_range()` establece la autonomía en 240 millas, y si la capacidad es de 85 kWh, establece la autonomía en 270 millas. A continuación, informa de este valor. Cuando queramos usar este método, de nuevo tenemos que llamarlo a través del atributo de la `batería del coche` en ❷.

La salida nos indica la autonomía del coche en función del tamaño de su batería:

```
Tesla Model S 2016
Este coche tiene una batería de 70 kWh.
Este coche puede recorrer aproximadamente 240 millas con una carga completa.
```

Modelado de objetos del mundo real

Cuando empieces a modelar elementos más complicados, como los coches eléctricos, te plantearás cuestiones interesantes. ¿La autonomía de un coche eléctrico es una propiedad de la batería o del coche? Si sólo estamos describiendo un coche, probablemente esté bien mantener la asociación del método `get_range()` con la clase `Battery`. Pero si estamos describiendo toda la línea de coches de un fabricante, proba-

Tal vez quieras mover `get_range()` a la clase `ElectricCar`. El método `get_range()`

seguiría comprobando el tamaño de la batería antes de determinar la autonomía, pero informaría de una autonomía específica para el tipo de coche con el que está asociado. Alternativamente, podríamos mantener la asociación del método `get_range()` con la batería pero pasándole un parámetro como `modelo_coche`. El método `get_range()` informaría entonces de una autonomía basada en el tamaño de la batería y el modelo de coche.

Esto te lleva a un punto interesante en tu crecimiento como programador. Cuando te enfrentas a preguntas como éstas, estás pensando en un nivel lógico superior en lugar de en un nivel centrado en la sintaxis. No estás pensando en Python, sino en cómo representar el mundo real en código. Cuando llegues a este punto, te darás cuenta de que a menudo no hay enfoques correctos o incorrectos para modelar situaciones del mundo real. Algunos enfoques son más eficientes que otros, pero se necesita práctica para encontrar las representaciones más eficientes. Si tu código funciona como quieres, lo estás haciendo bien. No te desanimes si descubres que estás desmontando tus clases y reescribiéndolas varias veces utilizando diferentes enfoques. En la búsqueda de un código preciso y eficiente, todo el mundo pasa por este proceso.

TRY IT yOURSELF

9-6. Puesto de helados: Un puesto de helados es una clase específica de restaurante. Escriba una clase llamada `PuestoDeHelados` que herede de la clase `Restaurante` que escribió en el Ejercicio 9-1 (página 166) o en el Ejercicio 9-4 (página 171). Cualquiera de las dos versiones de la clase funcionará; simplemente elige la que más te guste. Añade un atributo llamado `sabores` que almacene una lista de sabores de helados. Escribe un método que muestre estos sabores. Crea una instancia de `IceCreamStand`, y llama a este método.

9-7. Administrador: Un administrador es una clase especial de usuario. Escribe una clase llamada `Admin` que herede de la clase `Usuario` que escribiste en el Ejercicio 9-3 (página 166) o en el Ejercicio 9-5 (página 171). Añade un atributo, `privilegios`, que almacene una lista de cadenas como `"puede añadir post"`, `"puede borrar post"`, `"puede banear usuario"`, etc. Escribe un método llamado `show_privileges()` que liste el conjunto de privilegios del administrador. Crea una instancia de `Admin`, y llama a tu método.

9-8. Privilegios: Escriba una clase `Privilegios` separada. La clase debería tener un atributo, `privilegios`, que almacene una lista de cadenas como se describe en el Ejercicio 9-7. Mueva el método show_privileges() a esta clase. Mueva el método `show_privileges()` a esta clase. Haga una instancia de `Privilegios` como un atributo en la clase `Admin`. Cree una nueva instancia de `Admin` y use su método para mostrar sus privilegios.

9-9. Actualización de la batería: Utilice la versión final de *electric_car.py* de esta sección. Añade un método a la clase `Battery` llamado `upgrade_battery()`. Este método debe comprobar el tamaño de la batería y establecer la capacidad a 85 si no lo es ya.
Crea un coche eléctrico con un tamaño de batería por defecto, llama a `get_range()` una vez, y luego llama a `get_range()` una segunda vez después de actualizar la batería. Deberías ver un incremento en la autonomía del coche.

Importación de clases

A medida que añades más funcionalidad a tus clases, tus archivos pueden hacerse largos, incluso cuando usas la herencia correctamente. De acuerdo con la filosofía general de Python, querrás mantener tus archivos lo más despejados posible. Para ayudarte, Python te permite almacenar clases en módulos y luego importar las clases que necesites en tu programa principal.

Importar una sola clase

Vamos a crear un módulo que contenga sólo la clase `Car`. Esto plantea un sutil problema de nomenclatura: ya tenemos un archivo llamado car.*py* en este capítulo, pero este módulo debería llamarse *car.py* porque contiene código que representa un coche. Resolveremos este problema de nomenclatura almacenando la clase `Car` en un módulo llamado car.*py*, reemplazando el archivo *car.py* que estábamos usando anteriormente. A partir de ahora, cualquier programa que utilice este módulo necesitará un nombre de archivo más específico, como *mi_coche.py*. Aquí está car.*py* con sólo el código de la clase `Car`:

coche.py

```
❶ """Una clase que se puede utilizar para

representar un coche."""
class Coche():
    "Un simple intento de representar un coche".

    def init (self, marca, modelo, año):
        """Inicializar atributos para describir un
        coche."""
        self.marca = marca
        self.modelo = modelo
        self.año = año
        self.lectura_cuentakilómet
        ros = 0

    def obtener_nombre_descriptivo(self):
        """Devuelve un nombre descriptivo bien formateado."""
        long_name = str(self.year) + ' ' + self.make + ' ' + self.model
        return long_name.title()

    def leer_odometro(self):
        """Imprime un extracto con el kilometraje del coche."""
        print("Este coche tiene " + str(lectura_cuentakilómetros.self) + "
        kilómetros.")

    def actualizar_odometro(self,
        kilometraje): """
        Ajuste la lectura del cuentakilómetros al valor dado.
        Rechaza el cambio si intenta hacer retroceder el
        cuentakilómetros. """
        si cuentakilómetros >=
            cuentakilómetros_propio:
            cuentakilómetros_propio =
            cuentakilómetros
        si no:
            print("¡No se puede hacer retroceder un cuentakilómetros!")
```

```
    def incremento_odometro(self, millas):
        """Añade la cantidad dada a la lectura del cuentakilómetros."""
        self.lectura_cuentakilómetros += millas
```

En ❶ incluimos un docstring a nivel de módulo que describe brevemente el contenido de este módulo. Debes escribir un docstring para cada módulo que crees.

Ahora creamos un archivo separado llamado *mi_coche.py*. Este archivo importará el archivo
`Car` y luego crear una instancia de esa clase:

mi_coche.py

```
❶ from coche import Coche.

mi_nuevo_coche = Coche('audi', 'a4', 2016)
print(mi_nuevo_coche.get_descriptive_name())

mi_nuevo_coche.lectura_cuentakilómetros = 23
mi_nuevo_coche.leer_cuentakilómetros()
```

La sentencia `import` en ❶ le dice a Python que abra el módulo `car` e importe la clase `Car`. Ahora podemos usar la clase `Car` como si estuviera definida en este fichero. La salida es la misma que vimos antes:

```
Audi A4 2016
Este coche tiene 23 millas en él.
```

Importar clases es una forma eficaz de programar. Imagina lo largo que sería este archivo de programa si se incluyera toda la clase `Coche`. En cambio, si trasladas la clase a un módulo e importas el módulo, seguirás obteniendo todos los elementos de la clase.
la misma funcionalidad, pero manteniendo el archivo principal del programa limpio y fácil de leer. También almacena la mayor parte de la lógica en archivos separados; una vez que sus clases funcionan como desea, puede dejar esos archivos y centrarse en la lógica de alto nivel de su programa principal.

Almacenamiento de varias clases en un módulo

Puedes almacenar tantas clases como necesites en un solo módulo, aunque cada clase en un módulo debe estar relacionada de alguna manera. Las clases `Battery` y `ElectricCar` ayudan a representar coches, así que vamos a añadirlas al módulo *car.py*:

coche.py

```
"""Un conjunto de clases utilizadas para representar coches de

gasolina y eléctricos.""" class Coche():
    --snip--

clase Batería():
    """Un simple intento de modelar una batería para un coche eléctrico""".

    def init (self, battery_size=60):
        """Inicializar los atributos de la
        batería.""" self.battery_size = battery_size

    def describir_pila(self):
        """Imprime una declaración que describa el tamaño de la pila."""
```

```
        print("Este coche tiene una batería de " + str(self.battery_size) + "-
        kWh.")

    def get_range(self):
```

```
        """Imprime una declaración sobre la autonomía que proporciona
        esta batería.""" if self.battery_size == 70:
            alcance = 240
        elif self.battery_size == 85:
            range = 270

        message = "Este coche puede recorrer aproximadamente " +
        str(range) message += " millas con una carga completa."
        print(mensaje)

clase Cocheeléctrico(Coche):
    """Modela aspectos de un coche, específicos de los vehículos eléctricos""".

    def init (self, marca, modelo, año):
        """
        Inicializar atributos de la clase padre.
        A continuación, inicializa los atributos específicos de
        un coche eléctrico. """
        super(). init (marca, modelo, año)
        self.battery = Battery()
```

Ahora podemos crear un nuevo archivo llamado *mi_coche_eléctrico.py*, importar el archivo
Clase `ElectricCar`, y fabrica un coche eléctrico:

mi_eléctrico coche.py

```
from coche import Coche eléctrico

mi_tesla = Coche eléctrico('tesla', 'modelo s', 2016)

print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

Esto tiene la misma salida que vimos antes, a pesar de que la mayor parte de la lógica está oculta en un módulo:

```
Tesla Model S 2016
Este coche tiene una batería de 70 kWh.
Este coche puede recorrer aproximadamente 240 millas con una carga completa.
```

Importar varias clases de un módulo

Puedes importar tantas clases como necesites en un fichero de programa. Si queremos hacer un coche normal y un coche eléctrico en el mismo archivo, necesitamos importar ambas clases, `Coche` y `CocheEléctrico`:

mis_coches.py

```
❶ from coche import Coche, Cocheeléctrico.

❷ mi_escarabajo = Coche('volkswagen', 'escarabajo', 2016)
print(mi_escarabajo.get_descriptive_name())

❸ mi_tesla = Coche eléctrico('tesla', 'roadster', 2016)
```

```
print(mi_tesla.get_descriptive_name())
```

Puedes importar múltiples clases de un módulo separando cada clase con una coma ❶. Una vez importadas las clases necesarias, puedes crear tantas instancias de cada clase como necesites.

En este ejemplo, fabricamos un Volkswagen Escarabajo normal a ❷ y un Tesla Roadster eléctrico a ❸:

```
Volkswagen Escarabajo 2016
Tesla Roadster 2016
```

Importar un módulo completo

También puedes importar un módulo entero y acceder a las clases que necesites utilizando la notación de puntos. Este enfoque es sencillo y da como resultado un código fácil de leer. Debido a que cada llamada que crea una instancia de una clase incluye el nombre del módulo, no tendrá conflictos de nombres con cualquier nombre utilizado en el archivo actual.

Este es el aspecto que tiene importar todo el módulo de `coches` y luego crear un coche normal y un coche eléctrico:

mis_coches.py

```
❶ import coche

❷ mi_escarabajo = coche.Coche('volkswagen', 'escarabajo', 2016)
  print(mi_escarabajo.get_descriptive_name())

❸ mi_tesla = coche.Coche_eléctrico('tesla', 'roadster', 2016)
  print(mi_tesla.get_descriptive_name())
```

En ❶ importamos todo el módulo `del coche`. A continuación, accedemos a las clases que necesitamos a través de la sintaxis *module_name.class_name*. En ❷ creamos de nuevo un Volkswagen Beetle, y en ❸ creamos un Tesla Roadster.

Importar todas las clases de un módulo

Puede importar todas las clases de un módulo utilizando la siguiente sintaxis:

```
from nombre_del_módulo import *
```

Este método no es recomendable por dos razones. En primer lugar, es útil poder leer las declaraciones de `importación` en la parte superior de un archivo y obtener una clara
sentido de qué clases utiliza un programa. Con este enfoque no queda claro qué clases del módulo se están utilizando. Este enfoque también puede llevar a confusión con los nombres en el archivo. Si accidentalmente importa una clase con el mismo nombre que algo más en su archivo de programa, puede crear errores que son difíciles de diagnosticar. Muestro esto aquí porque aunque no es un enfoque recomendado, es probable que lo veas en el código de otras personas.

Si necesitas importar muchas clases de un módulo, es mejor que importes todo el módulo y utilices la sintaxis *nombre_módulo*.*nombre_clase*.

No verá todas las clases utilizadas en la parte superior del archivo, pero verá claramente dónde se utiliza el módulo en el programa. También evitarás los posibles conflictos de nombres que pueden surgir cuando importas todas las clases de un módulo.

Importación de un módulo a otro módulo

A veces querrá distribuir sus clases en varios módulos para evitar que un archivo crezca demasiado y almacenar archivos no relacionados.
clases en el mismo módulo. Cuando almacena sus clases en varios módulos, puede ocurrir que una clase de un módulo dependa de una clase de otro módulo. En este caso, puede importar la clase necesaria en el primer módulo.

Por ejemplo, almacenemos la clase `Coche` en un módulo y las clases `Coche Eléctrico` y `Batería` en otro módulo. Crearemos un nuevo módulo llamado *electric_car.py -reemplazando* el archivo *electric_car.py* que creamos anteriormente- y copiaremos sólo las clases `Batería` y Coche `Eléctrico` en este archivo:

coche_eléctrico.py

```
"""Un conjunto de clases que se pueden utilizar para representar coches
eléctricos."""

❶ from coche import

Coche class

Batería():
    --snip--

clase Cocheeléctrico(Coche):
    --snip--
```

La clase `CocheEléctrico` necesita acceso a su clase padre `Coche`, así que importamos `Coche` directamente al módulo en ❶. Si olvidamos esta línea, Python dará un error cuando intentemos crear una instancia de Coche `Eléctrico`. También necesitamos actualizar el `módulo Car` para que sólo contenga la `clase Car`:

coche.py

```
"""Una clase que se puede utilizar para

representar un coche.""" class Coche():
    --snip--
```

Ahora podemos importar de cada módulo por separado y crear cualquier tipo de coche que necesitemos:

mis_coches.py

```
❶ from coche import Coche.
from coche_eléctrico import Coche_eléctrico

mi_escarabajo = Coche('volkswagen', 'escarabajo', 2016)
print(mi_escarabajo.get_descriptive_name())
```

```
mi_tesla = Coche_eléctrico('tesla', 'roadster', 2016)
print(mi_tesla.get_descriptive_name())
```

En ❶ importamos `Car` de su módulo, y `ElectricCar` de su módulo. A continuación creamos un `coche` normal y un coche eléctrico. Ambos tipos de coches se crean correctamente:

```
Volkswagen Escarabajo 2016
Tesla Roadster 2016
```

Encontrar su propio flujo de trabajo

Como puedes ver, Python te da muchas opciones sobre cómo estructurar el código en un proyecto grande. Es importante conocer todas estas posibilidades para que puedas determinar las mejores formas de organizar tus proyectos, así como entender los proyectos de otras personas.

Cuando estés empezando, mantén una estructura de código sencilla. Intenta hacer todo en un archivo y mover tus clases a módulos separados una vez que todo esté funcionando. Si te gusta cómo interactúan los módulos y los archivos, intenta almacenar tus clases en módulos cuando empieces un proyecto. Encuentra un enfoque que te permita escribir código que funcione, y sigue a partir de ahí.

TRY IT yOURSELF

9-10. Restaurante importado: Usando su última clase `Restaurant`, guárdela en un mod- ule. Haga un archivo separado que importe `Restaurant`. Cree una instancia de `Restaurant` y llame a uno de sus métodos para mostrar que la sentencia `import` funciona correctamente.

9-11. Importar Admin: Comience con su trabajo del Ejercicio 9-8 (página 178). Guarde las clases `User`, `Privileges` y `Admin` en un módulo. Cree un archivo separado, cree una instancia de `Admin` y llame a `show_privileges()` para mostrar que todo funciona correctamente.

9-12. Módulos Múltiples: Almacene la clase `Usuario` en un módulo, y almacene las clases `Privilegios` y `Admin` en un módulo separado. En un archivo separado, cree una instancia de `Admin` y llame a `show_privileges()` para mostrar que todo sigue funcionando correctamente.

La biblioteca estándar de Python

La *biblioteca estándar de Python* es un conjunto de módulos incluidos con cada instalación de Python. Ahora que tienes una comprensión básica de cómo funcionan las clases, puedes empezar a utilizar módulos como estos que otros programadores han escrito. Puedes utilizar cualquier función o clase de la biblioteca estándar incluyendo una simple sentencia `import` al principio de tu archivo. Veamos una clase, `OrderedDict`, del módulo `collections`.

Los diccionarios te permiten conectar piezas de información, pero no llevan la cuenta del orden en que añades pares clave-valor. Si vas a crear un diccionario y quieres controlar el orden en que se añaden los pares clave-valor, puedes utilizar la clase `OrderedDict` del módulo `collections`. Las instancias de la clase `OrderedDict` se comportan casi exactamente igual que los diccionarios, con la diferencia de que no pierden de vista el orden en que se añaden los pares clave-valor.

Volvamos al ejemplo *favorite_languages.py* del capítulo 6. Esta vez llevaremos la cuenta del orden en que la gente responde a la encuesta:

favorite_idiomas.py

```
❶ from colecciones import OrderedDict

❷ idiomas_favoritos = OrderedDict()

❸ favorite_languages['jen'] =
  'python' favorite_languages['sarah']
  = 'c' favorite_languages['edward'] =
  'ruby' favorite_languages['phil'] =
  'python'

❹ para nombre, idioma en idiomas_favoritos.elementos():
      print(nombre.titulo() + " el idioma favorito del
      usuario es " +
          language.title() + ".")
```

Comenzamos importando la clase `OrderedDict` del módulo `collections` en ❶. En ❷ creamos una instancia de la clase `OrderedDict` y almacenamos esta instancia en `favorite_languages`. Observa que no hay llaves; la llamada a `OrderedDict()` crea un diccionario ordenado vacío y lo almacena en `favorite_languages`. A continuación, añadimos cada nombre y cada idioma a `favorite_languages` de uno en uno ❸. Ahora, cuando hagamos un bucle a través de `favorite_languages` en ❹, sabremos que siempre obtendremos las respuestas en el orden en que fueron añadidas:

```
El lenguaje favorito de Jen es
Python. El lenguaje favorito de
Sarah es C. El lenguaje favorito de
Edward es Ruby. El lenguaje
favorito de Phil es Python.
```

Esta es una gran clase a tener en cuenta porque combina la principal ventaja de las listas (conservar el orden original) con la principal característica de los diccionarios (conectar piezas de información). Cuando empieces a modelar situaciones del mundo real que te interesen, probablemente te encontrarás con una situación en la que un diccionario ordenado es exactamente lo que necesitas. A medida que aprendas más sobre la biblioteca estándar, te familiarizarás con una serie de módulos como éste que te ayudarán a manejar situaciones comunes.

NOTE *También puedes descargar módulos de fuentes externas. Verás varios de estos ejemplos en la Parte II, donde necesitaremos módulos externos para completar cada proyecto.*

TRY IT yOURSELF

9-13. Reescritura de OrderedDict: Comience con el Ejercicio 6-4 (página 108), donde utilizó un diccionario estándar para representar un glosario. Reescriba el programa usando la clase `OrderedDict` y asegúrese de que el orden de la salida coincide con el orden en que los pares clave-valor fueron añadidos al diccionario.

9-14. Dados: El módulo `random` contiene funciones que generan números aleatorios de varias maneras. La función `randint()` devuelve un número entero en el rango que le proporcione. El siguiente código devuelve un número entre 1 y 6:

```
from random import randint
x = randint(1, 6)
```

Crea una clase `Dado` con un atributo llamado `caras`, que tiene un valor por defecto de 6. Escribe un método llamado `lanzar_un_dado()` que imprima un número aleatorio entre 1 y el número de caras que tiene el dado. Haz un dado de 6 caras y lánzalo 10 veces.

Haz un dado de 10 caras y otro de 20 caras. Lanza cada dado 10 veces.

9-15. Módulo Python de la semana: Un excelente recurso para explorar la biblioteca estándar de Python es un sitio llamado *Python Module of the Week*. Entra en *http://pymotw.com/* y mira la tabla de contenidos. Busca un módulo que te parezca interesante y lee sobre él, o explora la documentación de los módulos `collections` y `random`.

Clases de estilismo

Merece la pena aclarar algunas cuestiones de estilo relacionadas con las clases, sobre todo a medida que tus programas se complican.

Los nombres de las clases deben escribirse en *CamelCaps*. Para ello, escriba en mayúsculas la primera letra de cada palabra del nombre y no utilice guiones bajos. Los nombres de instancias y módulos deben escribirse en minúsculas con guiones bajos entre las palabras.

Cada clase debe tener un docstring inmediatamente después de la defi- nición de la clase. El docstring debe ser una breve descripción de lo que hace la clase, y debe seguir las mismas convenciones de formato que utilizó para escribir docstrings en funciones. Cada módulo también debe tener una docstring que describa para qué se pueden utilizar las clases de un módulo.

Puede utilizar líneas en blanco para organizar el código, pero no las utilice excesivamente. 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 un módulo escrito por usted, coloque la sentencia import para el módulo de la

biblioteca estándar

primero. A continuación, añada una línea en blanco y la sentencia import del módulo que ha escrito. En programas con múltiples sentencias import, esta convención facilita ver de dónde proceden los diferentes módulos utilizados en el programa.

Resumen

En este capítulo aprendiste a escribir tus propias clases. Aprendiste cómo almacenar información en una clase usando atributos y cómo escribir métodos que den a tus clases el comportamiento que necesitan. Aprendiste a escribir métodos `__init__()` que crean instancias de tus clases con exactamente los atributos que quieres. Viste cómo modificar los atributos de una instancia directamente y a través de métodos. Aprendiste que la herencia puede simplificar la creación de clases que están relacionadas entre sí, y aprendiste a usar instancias de una clase como atributos en otra clase para mantener cada clase simple.

Viste cómo almacenar clases en módulos e importar clases que necesitas en los archivos donde serán usadas puede mantener tus proyectos organizados. Empezaste a aprender sobre la biblioteca estándar de Python, y viste un ejemplo basado en la clase `OrderedDict` del módulo `collections`. Finalmente, aprendiste a dar estilo a tus clases usando las convenciones de Python.

En el Capítulo 10 aprenderás a trabajar con archivos para que puedas guardar el trabajo que has hecho en un programa y el trabajo que has permitido hacer a los usuarios. También aprenderás sobre *excepciones*, una clase especial de Python diseñada para ayudarte a responder a los errores cuando surgen.

10

FILES Y EXCEPTIONS



Ahora que ya dominas los conocimientos básicos necesarios para escribir programas organizados que sean fáciles de usar, es hora de pensar en hacer que sus programas sean aún más relevantes y útil. En este capítulo aprenderás a trabajar con archivos para que tus programas puedan analizar rápidamente muchos datos.

Aprenderás a manejar errores para que tus programas no se bloqueen cuando se encuentren con situaciones inesperadas. Aprenderás sobre *excepciones*, que son objetos especiales que Python crea para manejar errores que surgen mientras un programa se está ejecutando. También aprenderás sobre el módulo `json`, que te permite guardar datos de usuario para que no se pierdan cuando tu programa deje de ejecutarse.

Aprender a trabajar con archivos y a guardar datos facilitará el uso de sus programas. Los usuarios podrán elegir qué datos introducir y cuándo hacerlo. La gente puede ejecutar su programa, hacer algo de trabajo, y luego cerrar el programa y continuar donde lo dejaron más tarde. Aprender

a manejar excepciones le ayudará a lidiar con situaciones en las que los archivos no existen y lidiar con otros problemas que pueden causar que sus programas se bloqueen. Esto hará que sus programas sean más robustos cuando se encuentren con datos erróneos, ya sean

proviene de errores inocentes o de intentos maliciosos de romper tus programas. Con los conocimientos que aprenderá en este capítulo, conseguirá que sus programas sean más aplicables, utilizables y estables.

Lectura de un archivo

Los archivos de texto contienen una cantidad increíble de datos. Los archivos de texto pueden contener datos meteorológicos, datos de tráfico, datos socioeconómicos, obras literarias y mucho más. La lectura de un archivo es particularmente útil en aplicaciones de análisis de datos, pero también es aplicable a cualquier situación en la que desee analizar o modificar la información almacenada en un archivo. Por ejemplo, puede escribir un programa que lea el contenido de un archivo de texto y reescriba el archivo con un formato que permita visualizarlo en un navegador.

Cuando quieras trabajar con la información de un fichero de texto, el primer paso es leer el fichero en la memoria. Puedes leer todo el contenido de un fichero o trabajar con él línea a línea.

Lectura de un archivo completo

Para empezar, necesitamos un archivo con algunas líneas de texto. Vamos a empezar con un archivo que contiene *pi* a 30 decimales con 10 decimales por línea:

pi_digits.txt

```
3.1415926535
  8979323846
  2643383279
```

Para probar los siguientes ejemplos usted mismo, puede introducir estas líneas en un editor y guardar el archivo como *pi_digits.txt*, o puede descargar el archivo de los recursos del libro a través de https://www.nostarch.com/pythoncrashcourse/. Guarda el archivo en el mismo directorio donde guardarás los programas de este capítulo.

He aquí un programa que abre este fichero, lo lee e imprime su contenido en la pantalla:

lector_archivos.py

```
con open('pi_digits.txt') como archivo_objeto:
    contenido =
    objeto_archivo.read()
    print(contenido)
```

La primera línea de este programa tiene mucho que ver. Empecemos por la función `open()`. Para hacer cualquier trabajo con un archivo, aunque sólo sea imprimir sus contenidos, primero necesita *abrir* el archivo para acceder a él. La función `open()` necesita un argumento: el nombre del archivo que quieres abrir. Python busca este archivo en el directorio donde está almacenado el programa que se está ejecutando en ese momento. En este ejemplo, se está ejecutando *file_reader.py*, por lo que Python busca *pi_digits.txt* en el directorio donde está almacenado *file_reader.py*. La función `open()` devuelve un objeto que representa el archivo. Aquí,

`open('pi_digits.txt')` devuelve un objeto que representa a *pi_digits.txt*. Python almacena este objeto en `file_object`, con el que trabajaremos más adelante en el programa.

La palabra clave `con` cierra el archivo una vez que ya no se necesita acceder a él. Observe que en este programa llamamos a `open()` pero no a `close()`. Podrías abrir

y cerrar el archivo llamando a `open()` y `close()`, pero si un error en su programa impide que se ejecute la sentencia `close()`, es posible que el archivo nunca se cierre. Esto puede parecer trivial, pero los archivos mal cerrados pueden causar la pérdida o corrupción de datos. Y si llama a `close()` demasiado pronto en su programa, se encontrará tratando de trabajar con un archivo *cerrado* (un archivo al que no puede acceder), lo que conduce a más errores. No siempre es fácil saber exactamente cuándo debe cerrar un archivo, pero con la estructura mostrada aquí, Python lo hará por ti. Todo lo que tienes que hacer es abrir el archivo y trabajar con él como desees, confiando en que Python lo cerrará automáticamente cuando sea el momento adecuado.

Una vez que tenemos un objeto archivo que representa *pi_digits.txt*, utilizamos el método `read()` en la segunda línea de nuestro programa para leer todo el contenido del archivo y almacenarlo como una cadena larga en `contents`. Cuando imprimimos el valor de `contents`, obtenemos de vuelta el archivo de texto completo:

```
3.1415926535
  8979323846
  2643383279
```

La única diferencia entre esta salida y el fichero original es la línea en blanco adicional al final de la salida. La línea en blanco aparece porque `read()` devuelve una cadena vacía cuando llega al final del fichero; esta cadena vacía aparece como una línea en blanco. Si desea eliminar la línea en blanco adicional, puede utilizar `rstrip()` en la sentencia `print`:

```
with open('pi_digits.txt') as
    objeto_archivo: contenido =
    objeto_archivo.read()
    print(contenido.rstrip())
```

Recuerde que el método `rstrip()` de Python elimina cualquier carácter de espacio en blanco de la parte derecha de una cadena. Ahora la salida coincide exactamente con los contenidos del archivo original:

```
3.1415926535
  8979323846
  2643383279
```

Rutas de archivos

Cuando pasas un nombre de archivo simple como *pi_digits.txt* a la función `open()`, Python busca en el directorio donde está almacenado el archivo que se está ejecutando en ese momento (es decir, tu archivo de programa *.py*).

A veces, dependiendo de cómo organices tu trabajo, el archivo que quieres abrir no estará en el mismo directorio que tu archivo de programa. Por ejemplo, puedes almacenar tus archivos de programa en una carpeta llamada *python_work*; dentro de *python_work*, puedes tener otra carpeta

llamada *text_files* para distinguir tus archivos de programa de los archivos de texto que están manipulando. Aunque *text_files* esté en *python_work*, pasarle a `open()` el nombre de un fichero en *text_files* no funcionará, porque Python sólo buscará en *python_work* y se detendrá ahí; no seguirá con

y buscar en *archivos_de_texto*. Para hacer que Python abra archivos de un directorio distinto de aquel en el que está almacenado el archivo del programa, es necesario proporcionar una *ruta de archivo*, que le indica a Python que busque en una ubicación específica del sistema.

Como *text_files* está dentro de *python_work*, puedes usar una ruta de archivo relativa para abrir un archivo desde *text_files*. Una ruta *de archivo relativa* le dice a Python que busque una ubicación dada relativa al directorio donde está almacenado el archivo del programa que se está ejecutando actualmente. En Linux y OS X, escribirías:

```
con open('archivos_de_texto/nombre_de_archivo.txt') como objeto_archivo:
```

Esta línea le dice a Python que busque el archivo *.txt* deseado en la carpeta *text_files* y asume que *text_files* se encuentra dentro de *python_work* (que lo está). En los sistemas Windows, se utiliza una barra invertida (\) en lugar de una barra inclinada (/) en la ruta del archivo:

```
con open('nombre_archivo.txt') como objeto_archivo:
```

También puedes decirle a Python exactamente dónde está el archivo en tu ordenador, independientemente de dónde esté almacenado el programa que se está ejecutando. Esto se llama una ruta *absoluta de archivo*. Utiliza una ruta absoluta si una ruta relativa no funciona. Por ejemplo, si has puesto *archivos_de_texto* en alguna carpeta que no sea *python_work-por ejemplo*, una carpeta llamada otros_archivos-entonces pasarle a `open()` la ruta `'archivos_de_texto/nombre_archivo.txt'` no funcionará porque Python sólo buscará esa ubicación dentro de *python_work*. Tendrás que escribir una ruta completa para aclarar dónde quieres que Python busque.

Las rutas absolutas suelen ser más largas que las relativas, por lo que es útil almacenarlas en una variable y luego pasar esa variable a `open()`. En Linux y OS X, las rutas absolutas tienen este aspecto:

```
ruta_archivo =
'/home/ehmatthes/otros_archivos/archivos_de_texto/nombre_archivo.txt'
with open(ruta_archivo) as objeto_archivo:
```

y en Windows tienen este aspecto:

```
file_path = 'C:\sers\ehmatthes\other_files\text_files\ename.txt' with
open(file_path) as file_object:
```

Utilizando rutas absolutas, puede leer archivos desde cualquier ubicación de su sistema. Por ahora, lo más fácil es almacenar los archivos en el mismo directorio que los archivos de programa o en una carpeta como *archivos_de_texto* dentro del directorio que almacena los archivos de programa.

NOTE *Los sistemas Windows a veces interpretan correctamente las barras diagonales en las rutas de los archivos. Si utiliza Windows y no obtiene los resultados esperados,*

pruebe a utilizar barras invertidas.

Leer línea por línea

Al leer un archivo, a menudo querrá examinar cada línea del mismo. Puede que estés buscando cierta información en el fichero, o puede que quieras modificar el texto del fichero de alguna manera. Por ejemplo, es posible que desee leer un archivo de datos meteorológicos y trabajar con cualquier línea que incluya la palabra *soleado* en la descripción del tiempo de ese día. En un informe de noticias, puede buscar cualquier línea con la etiqueta `<headline>` y reescribir esa línea con un tipo específico de formato.

Puede utilizar un bucle `for` en el objeto fichero para examinar cada línea de un fichero de una en una:

file_reader.py

```
❶ filename = 'pi_digits.txt'

❷ con open(nombre_archivo) como archivo_objeto:
❸     for línea in
          objeto_archivo:
          print(línea)
```

En ❶ almacenamos el nombre del archivo del que estamos leyendo en la variable `filename`. Esta es una convención común cuando se trabaja con archivos. Dado que la variable `filename` no representa el archivo real -es sólo una cadena que le dice a Python dónde encontrar el archivo- puedes cambiar fácilmente `'pi_digits.txt'` por el nombre de otro archivo con el que quieras trabajar. Después de llamar a `open()`, un objeto que representa el archivo y su contenido se almacena en la variable `archivo_objeto` ❷. Volvemos a utilizar la sintaxis `with` para que Python abra y cierre el archivo correctamente. Para examinar el contenido del archivo, trabajamos a través de cada línea de l a r c h i v o haciendo un bucle sobre el objeto archivo ❸.

Al imprimir cada línea, encontramos aún más líneas en blanco:

```
3.1415926535

  8979323846

  2643383279
```

Estas líneas en blanco aparecen porque hay un carácter de nueva línea invisible al final de cada línea del archivo de texto. La sentencia `print` añade su propia nueva línea cada vez que la invocamos, por lo que terminamos con dos caracteres de nueva línea en al final de cada línea: una del fichero y otra de la *sentencia* `print`. El uso de `rstrip()` en cada línea de la *sentencia* `print` elimina estas líneas en blanco adicionales:

```
nombre_archivo = 'pi_digits.txt'

con open(nombre_archivo) como
    objeto_archivo: for línea en
```

```
objeto_archivo:
    print(line.rstrip())
```

Ahora la salida vuelve a coincidir con el contenido del archivo:

```
3.1415926535
  8979323846
  2643383279
```

Hacer una lista de líneas de un fichero

Cuando se utiliza `with`, el objeto fichero devuelto por `open()` sólo está disponible dentro del bloque `with` que lo contiene. Si desea mantener el acceso a los contenidos de un archivo fuera del bloque `with`, puede almacenar las líneas del archivo en una lista dentro del bloque y luego trabajar con esa lista. Puede procesar partes del fichero inmediatamente y posponer parte del procesamiento para más adelante en el programa.

El siguiente ejemplo almacena las líneas de *pi_digits.txt* en una lista dentro del archivo
y luego imprime las líneas fuera del bloque `with`:

```
nombre_archivo = 'pi_digits.txt'

con open(nombre_archivo) como objeto_archivo:
❶    líneas = objeto_archivo.readlíneas()

❷ para línea en líneas:
    print(line.rstrip())
```

En ❶ el método `readlines()` toma cada línea del archivo y la almacena en una lista. Esta lista se almacena entonces en `lines`, con la que podemos seguir trabajando después de que termine el bloque `with`. En ❷ utilizamos un simple bucle `for` para imprimir cada línea de `lines`. Como cada elemento de `lines` corresponde a cada línea del fichero, la salida coincide exactamente con el contenido del fichero.

Trabajar con el contenido de un fichero

Después de leer un archivo en memoria, puedes hacer lo que quieras con esos datos, así que vamos a explorar brevemente los dígitos de *pi*. Primero, intentaremos construir una sola cadena que contenga todos los dígitos del `archivo` sin espacios en blanco:

pi_cadena.py

```
nombre_archivo = 'pi_digits.txt'

con open(nombre_archivo) como
    objeto_archivo: líneas =
    objeto_archivo.readlíneas()

❶ pi_string = ''
❷ para línea en líneas:
    pi_string += line.rstrip()

❸ print(pi_cadena)
```

```
print(len(pi_cadena))
```

Empezamos abriendo el fichero y almacenando cada línea de dígitos en una lista, igual que hicimos en el ejemplo anterior. En ❶ creamos una variable, `pi_string`, para

contienen los dígitos de *pi*. A continuación, creamos un bucle que añade cada línea de dígitos a `pi_string` y elimina el carácter de nueva línea de cada línea ❷. En ❸ imprimimos esta cadena y también mostramos su longitud:

```
3.1415926535 8979323846 2643383279
36
```

La variable `pi_string` contiene los espacios en blanco que había a la izquierda de los dígitos en cada línea, pero podemos deshacernos de ellos utilizando `strip()` en lugar de `rstrip()`:

```
nombre_archivo = 'pi_30_digitos.txt'

con open(nombre_archivo) como
    objeto_archivo: líneas =
    objeto_archivo.readlíneas()

pi_string = ''
para línea en líneas:
    pi_string += line.strip()

print(pi_cadena)
print(len(pi_cadena))
```

Ahora tenemos una cadena que contiene *pi* con 30 decimales. La cadena tiene 32 caracteres porque también incluye el 3 inicial y un punto decimal:

```
3.141592653589793238462643383279
32
```

NOTE *Cuando Python lee de un archivo de texto, interpreta todo el texto del archivo como una cadena. Si lees un número y quieres trabajar con ese valor en un contexto numérico, tendrás que convertirlo a un entero usando la función* `int()` *o convertirlo a un flotador usando la función* `float()`.

Archivos grandes: Un millón de dígitos

Hasta ahora nos hemos centrado en el análisis de un archivo de texto que contiene sólo tres líneas, pero el código de estos ejemplos funcionaría igual de bien en archivos mucho más grandes. Si empezamos con un archivo de texto que contiene *pi* con 1.000.000 decimales en lugar de sólo 30, podemos crear una única cadena que contenga todos estos dígitos. No necesitamos cambiar nuestro programa en absoluto excepto para pasarle un archivo diferente. También vamos a imprimir sólo los primeros 50 decimales, por lo que no tenemos que ver un millón de dígitos desplazarse en el terminal:

pi_cadena.py

```
nombre_archivo =
```

```
'pi_millones_digitos.txt' with

open(nombre_archivo) as

objeto_archivo:
    líneas = objeto_archivo.readlíneas()
```

```
pi_string = ''
para línea en líneas:
    pi_string += line.strip()

print(pi_string[:52] + "...")
print(len(pi_string))
```

La salida muestra que efectivamente tenemos una cadena que contiene *pi* con 1.000.000 decimales:

```
3.14159265358979323846264338327950288419716939937510...
1000002
```

Python no tiene un límite inherente a la cantidad de datos con los que puedes trabajar; puedes trabajar con tantos datos como la memoria de tu sistema pueda manejar.

NOTE *Para ejecutar este programa (y muchos de los ejemplos que siguen), necesitará descargar los recursos disponibles en https://www.nostarch.com/pythoncrashcourse/.*

¿Su cumpleaños está contenido en Pi?

Siempre he tenido curiosidad por saber si mi cumpleaños aparece en algún lugar de los dígitos de *pi*. Utilicemos el programa que acabamos de escribir para averiguar si el cumpleaños de alguien aparece en algún lugar del primer millón de dígitos de *pi*. Podemos h a c e r l o expresando cada cumpleaños como una cadena de dígitos y viendo si esa cadena aparece en algún lugar de `pi_cadena`:

```
nombre_archivo =

'pi_million_digits.txt' with

open(nombre_archivo) as

objeto_archivo:
    líneas = objeto_archivo.readlíneas()

pi_string = ''
para línea en líneas:
    pi_string += line.rstrip()

❶ cumpleaños = input("Introduzca su fecha de cumpleaños, en el formulario mmddyy:
")
❷ si cumpleaños en pi_string:
    print("¡Tu cumpleaños aparece en el primer millón de dígitos de
pi!") else:
    print("Tu fecha de nacimiento no aparece en el primer millón de dígitos de
    pi.")
```

En ❶ pedimos el cumpleaños del usuario, y luego en ❷ comprobamos si esa cadena está en `pi_string`. Intentémoslo:

```
Introduzca su fecha de nacimiento, en formato mmddyy: 120372
¡Tu cumpleaños aparece en el primer millón de dígitos de pi!
```

¡Mi cumpleaños sí aparece en los dígitos de *pi*! Una vez que has leído un archivo, puedes analizar su contenido de cualquier forma que puedas imaginar.

TRY IT yOURSELF

10-1. Aprendiendo Python: Abre un archivo en blanco en tu editor de texto y escribe unas líneas resumiendo lo que has aprendido sobre Python hasta ahora. Comienza cada línea con la frase *En Python puedes* Guardar el archivo como *aprendiendo_python.txt* en la carpeta mismo directorio que tus ejercicios de este capítulo. Escribe un programa que lea el fichero e imprima lo que has escrito tres veces. Imprime el contenido una vez leyendo el archivo completo, una vez haciendo un bucle sobre el objeto archivo y una vez almacenando las líneas en una lista y trabajando con ellas fuera del bloque `with`.

10-2. Aprendizaje de C: Puede utilizar el método `replace()` para sustituir cualquier palabra de una cadena por otra diferente. He aquí un ejemplo rápido que muestra cómo sustituir `'perro'` por `'gato'` en una frase:

```
>>> mensaje = "Me gustan mucho los perros".
>>> message.replace('perro', 'gato')
Me gustan mucho los gatos.
```

Lee cada línea del archivo que acabas de crear, *learning_python.txt*, y sustituye la palabra *Python* por el nombre de otro lenguaje, como *C*. Imprime en la pantalla cada línea modificada.

Escribir en un archivo

Una de las formas más sencillas de guardar datos es escribirlos en un archivo. Cuando escribes texto en un archivo, la salida seguirá estando disponible después de que cierres el terminal que contiene la salida de tu programa. Puede examinar la salida después de que un programa termine de ejecutarse, y también puede compartir los archivos de salida con otros. También puedes escribir programas que vuelvan a leer el texto en la memoria y trabajar con él más tarde.

Escribir en un archivo vacío

Para escribir texto en un fichero, necesitas llamar a `open()` con un segundo argumento que indique a Python que quieres escribir en el fichero. Para ver cómo funciona, vamos a escribir un mensaje simple y almacenarlo en un archivo en lugar de imprimirlo en la pantalla:

escribir_mensaje.py

```
nombre_archivo = 'programacion.txt'

❶ con open(nombre_archivo, 'w') como archivo_objeto:
❷     file_object.write("Me encanta programar.")
```

La llamada a `open()` en este ejemplo tiene dos argumentos ❶. El primer argumento sigue siendo el nombre del archivo que queremos abrir. El

segundo argumento, `'w'`, le dice a Python que queremos abrir el archivo en *modo de escritura*. Puedes abrir un archivo

en *modo lectura* (`'r'`), *modo escritura* (`'w'`), *modo append* (`'a'`), o un modo que te permite leer y escribir en el archivo (`'r+'`). Si omites el argumento modo, Python abre el archivo en modo sólo lectura por defecto.

La función `open()` crea automáticamente el archivo en el que estás escribiendo si aún no existe. Sin embargo, ten cuidado al abrir un archivo en modo escritura (`'w'`) porque si el archivo existe, Python lo borrará antes de devolver el objeto archivo.

En ❷ usamos el método `write()` en el objeto file para escribir una cadena en el archivo. Este programa no tiene salida de terminal, pero si abres el archivo *programming.txt*, verás una línea:

programación.txt

```
Me encanta programar.
```

Este archivo se comporta como cualquier otro archivo de tu ordenador. Puedes abrirlo, escribir texto en él, copiarlo, pegarlo, etc.

NOTE *Python sólo puede escribir cadenas en un archivo de texto. Si quieres almacenar datos numéricos en un fichero de texto, tendrás que convertirlos primero a formato cadena utilizando la función* ***str()****.*

Escribir varias líneas

La función `write()` no añade ninguna nueva línea al texto que escribes. Por tanto, si escribe más de una línea sin incluir caracteres de nueva línea, es posible que su fichero no tenga el aspecto deseado:

```
nombre de archivo = 'programacion.txt'

with open(nombre_archivo, 'w') as objeto_archivo:
    objeto_archivo.write("Me encanta programar.")
    objeto_archivo.write("Me encanta crear juegos
    nuevos.")
```

Si abres *programming.txt*, verás las dos líneas aplastadas:

```
Me encanta programar. Me encanta crear juegos nuevos.
```

Incluir nuevas líneas en las sentencias `write()` hace que cada cadena aparezca en su propia línea:

```
nombre de archivo = 'programacion.txt'

with open(nombre_archivo, 'w') as objeto_archivo:
    objeto_archivo.write("Me encanta programar.\n")
    objeto_archivo.write("Me encanta crear nuevos
    juegos.\n")
```

La salida aparece ahora en líneas separadas:

```
Me encanta programar.
Me encanta crear juegos nuevos.
```

También puede utilizar espacios, caracteres de tabulación y líneas en blanco para dar formato a la salida, al igual que ha estado haciendo con la salida basada en terminal.

Añadir a un fichero

Si quieres añadir contenido a un archivo en lugar de escribir sobre el contenido existente, puedes abrir el archivo en *modo append*. Cuando abres un archivo en modo append, Python no borra el archivo antes de devolver el objeto archivo. Cualquier línea que escribas en el archivo se añadirá al final del archivo. Si el archivo aún no existe, Python creará un archivo vacío.

Modifiquemos *write_message.py* añadiendo algunas nuevas razones por las que nos encanta pro- gramar al archivo existente *programming.txt*:

escribir mensaje.py

```
nombre_archivo = 'programacion.txt'

❶ con open(nombre_archivo, 'a') como archivo_objeto:
❷     file_object.write("También me encanta encontrar significado en grandes
      conjuntos de datos.\n") file_object.write("Me encanta crear
      aplicaciones que puedan ejecutarse en un navegador.\n")
```

En ❶ usamos el argumento `'a'` para abrir el fichero a añadir en lugar de escribir sobre el fichero existente. En ❷ escribimos dos nuevas líneas, que se añaden a *programming.txt*:

programación.txt

```
Me encanta programar.
Me encanta crear juegos nuevos.
También me encanta encontrar sentido a grandes
conjuntos de datos. Me encanta crear
aplicaciones que puedan ejecutarse en un
navegador.
```

Terminamos con el contenido original del archivo, seguido del nuevo contenido que acabamos de añadir.

TRY IT yOURSELF

10-3. Invitado: Escribe un programa que pregunte al usuario su nombre. Cuando responda, escribe su nombre en un fichero llamado *guest.txt*.

10-4. Libro de visitas: Escribe un bucle `while` que pregunte a los usuarios por su nombre. Cuando ingresen su nombre, imprima un saludo en la pantalla y agregue una línea registrando su visita en un archivo llamado *guest_book.txt*. Asegúrate de que cada entrada aparece en una nueva línea del fichero.

10-5. Encuesta sobre programación: Escribe un bucle `while` que pregunte a la gente por qué les gusta programar. Cada vez que alguien introduzca una razón, añádela a un archivo que almacene todas las respuestas.

Excepciones

Python utiliza objetos especiales llamados *excepciones* para gestionar los errores que surgen durante la ejecución de un programa. Cada vez que se produce un error que hace que Python no sepa qué hacer a continuación, crea un objeto de excepción. Si escribes código que maneje la excepción, el programa continuará ejecutándose. Si no manejas la excepción, el programa se detendrá y mostrará un *rastreo*, que incluye un informe de la excepción que se produjo.

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 se produce una excepción. Cuando usas bloques `try-except`, tus programas continuarán ejecutándose incluso si las cosas empiezan a ir mal. En lugar de trazas, que pueden ser confusas de leer para los usuarios, los usuarios verán mensajes de error amigables que tú escribes.

Tratamiento de la excepción ZeroDivisionError

Veamos un error sencillo que hace que Python lance una excepción. Probablemente sabes que es imposible dividir un número por cero, pero vamos a pedirle a Python que lo haga de todos modos:

división.py

```
print(5/0)
```

Por supuesto Python no puede hacer esto, así que obtenemos un traceback:

```
Traceback (última llamada más reciente):
  File "division.py", line 1, in <module>
    print(5/0)
❶ ZeroDivisionError: división por cero.
```

El error reportado en ❶ en el traceback, `ZeroDivisionError`, es un objeto de excepción. Python crea este tipo de objeto en respuesta a una situación en la que no puede hacer lo que le pedimos. Cuando esto ocurre, Python detiene el programa y nos indica el tipo de excepción que se ha producido. Podemos utilizar esta información para modificar nuestro programa. Le diremos a Python qué hacer cuando ocurra este tipo de excepción; así, si vuelve a ocurrir, estaremos preparados.

Uso de bloques try-except

Cuando pienses que puede ocurrir un error, puedes escribir un bloque `try-except` para manejar la excepción que pueda surgir. Le dices a Python que intente ejecutar algún código, y le dices qué hacer si el código resulta en un tipo particular de excepción.

Este es el aspecto de un bloque `try-except` para tratar la excepción `ZeroDivisionError`:

```
Inténtalo:
    print(5/0)
excepto ZeroDivisionError:
```

```
    print("¡No se puede dividir por cero!")
```

Pusimos `print(5/0)`, la línea que causó el error, dentro de un bloque `try`. Si el código del bloque `try` funciona, Python se salta el bloque `except`. Si el código en el bloque `try` causa un error, Python busca un bloque `except` cuyo error coincida con el que se produjo y ejecuta el código en ese bloque.

En este ejemplo, el código en el bloque `try` produce un `ZeroDivisionError`, por lo que Python busca un bloque `except` que le indique cómo responder. Python entonces ejecuta el código en ese bloque, y el usuario ve un mensaje de error amigable en lugar de un traceback:

```
¡No se puede dividir por cero!
```

Si más código siguiera al bloque `try-except`, el programa continuaría ejecutándose porque le dijimos a Python cómo manejar el error. Veamos un ejemplo en el que la captura de un error puede permitir que un programa continúe ejecutándose.

Uso de excepciones para evitar colisiones

Manejar los errores correctamente es especialmente importante cuando el programa tiene más trabajo que hacer después de que se produzca el error. Esto ocurre a menudo en programas que solicitan a los usuarios que introduzcan datos. Si el programa responde adecuadamente a las entradas no válidas, puede pedir más entradas válidas en lugar de bloquearse.

Vamos a crear una calculadora sencilla que sólo haga divisiones:

división.py

```
print("Dame dos números, y los dividiré.")
print("Introduce 'q' para salir.")

mientras sea verdad:
❶    primer_número = input("\nPrimer número:
     ") if primer_número == 'q':
         romper
❷    segundo_número = input("Segundo número:
     ") if segundo_número == 'q':
         romper
❸    respuesta = int(primer_número) /
     int(segundo_número) print(respuesta)
```

Este programa pide al usuario que introduzca un `primer_número` ❶ y, si no introduce *q* para salir, un `segundo_número` ❷. A continuación, dividimos estos dos números para obtener una `respuesta` ❸. Este programa no hace nada para manejar errores, por lo que pedirle que divida por cero hace que se bloquee:

```
Dame dos números y los dividiré. Introduce
'q' para salir.

Primer número: 5
Segundo número: 0
Traceback (última llamada más reciente):
```

```
  File "division.py", line 9, in <module>.
    answer = int(primer_número) /
int(segundo_número) ZeroDivisionError: división por
cero
```

Es malo que el programa se haya bloqueado, pero tampoco es buena idea dejar que los usuarios vean los rastreos. Los usuarios no técnicos se sentirán confundidos por ellos, y en un entorno malicioso, los atacantes aprenderán más de lo que quieres que sepan de un rastreo. Por ejemplo, sabrán el nombre de tu archivo de programa y verán una parte de tu código que no funciona correctamente. Un atacante hábil puede a veces utilizar esta información para determinar qué tipo de ataques utilizar contra tu código.

El otro bloque

Podemos hacer este programa más resistente a errores envolviendo la línea que podría producir errores en un bloque `try-except`. El error se produce en la línea que realiza la división, así que ahí es donde pondremos el `bloque try-except`. Este ejemplo también incluye un `bloque else`. Cualquier código que dependa de que el bloque `try` se ejecute correctamente va en el `bloque else`:

```
print("Dame dos números y los dividiré")
print("Introduce 'q' para salir")

mientras sea verdad:
    primer_número = input("\nPrimer número:
    ") if primer_número == 'q':
        romper
    segundo_número = input("Segundo número: ")
❶   probar:
        respuesta = int(primer_número) / int(segundo_número)
❷   excepto ZeroDivisionError:
        print("¡No se puede dividir por 0!")
❸   else:
        print(respues
        ta)
```

Pedimos a Python que intente completar la operación de división en un bloque `try` ❶, que incluye sólo el código que podría causar un error. Cualquier código que dependa del éxito del bloque `try` se añade al `bloque else`. En este caso, si la operación de división tiene éxito, utilizamos el bloque `else` para imprimir el resultado ❸.

El bloque `except` le dice a Python cómo responder cuando se produce un `ZeroDivisionError` ❷. Si la sentencia `try` no tiene éxito debido a un error de división por cero, imprimimos un mensaje amigable indicando al usuario cómo evitar este tipo de error. El programa continúa ejecutándose, y el usuario nunca ve un traceback:

```
Dame dos números y los dividiré. Introduce
'q' para salir.

Primer número: 5
Segundo número: 0
```

```
¡No se puede dividir por 0!
```

```
Primer número: 5
Segundo número: 2
2.5

Primer número: q
```

El bloque `try-except-else` funciona así: Python intenta ejecutar el código de la sentencia `try`. El único código que debe ir en una sentencia `try` es el código que pueda causar una excepción. A veces tendrás código adicional que debe ejecutarse sólo si el bloque `try` tuvo éxito; este código va en el bloque `else`. El bloque `except` indica a Python qué hacer en caso de que se produzca una excepción al intentar ejecutar el código de la sentencia `try`.

Al anticiparse a las posibles fuentes de error, puede escribir programas robustos que continúen ejecutándose incluso cuando se encuentren con datos no válidos y recursos faltantes. Su código será resistente a los errores inocentes de los usuarios y a los ataques maliciosos.

Manejo de la excepción FileNotFoundError

Un problema habitual al trabajar con archivos es el de los archivos perdidos. El archivo que busca puede estar en una ubicación diferente, el nombre del archivo puede estar mal escrito, o el archivo puede no existir en absoluto. Puede manejar todas estas situaciones de forma sencilla con un bloque `try-except`.

Intentemos leer un fichero que no existe. El siguiente programa intenta leer el contenido de *Alicia en el País de las Maravillas*, pero no he guardado el archivo *alice.txt en el* mismo directorio que *alice.py*:

alice.py

```
nombre de archivo = 'alice.txt'

con open(nombre_archivo) como
    f_obj: contenido =
    f_obj.read()
```

Python no puede leer de un archivo que falta, por lo que lanza una excepción:

```
Traceback (última llamada más reciente):
  File "alice.py", line 3, in <module>
    with open(nombre_archivo) as f_obj:
FileNotFoundError: [Errno 2] No existe tal archivo o directorio: 'alice.txt'
```

La última línea del rastreo reporta un `FileNotFoundError`: esta es la excepción que Python crea cuando no puede encontrar el archivo que está tratando de abrir. En este ejemplo, la función `open()` produce el error, así que para manejarlo, el bloque `try` comenzará justo antes de la línea que contiene `open()`:

```
filename = 'alice.txt'
```

```
try:
    con open(nombre_archivo) como
        f_obj: contenido =
        f_obj.read()
```

```
excepto FileNotFoundError:
    msg = "Lo sentimos, el archivo " + nombre de archivo + " no
    existe." print(msg)
```

En este ejemplo, el código en el bloque `try` produce un `FileNotFoundError`, por lo que Python busca un bloque `except` que coincida con ese error. Python entonces ejecuta el código en ese bloque, y el resultado es un mensaje de error amigable en lugar de un traceback:

```
Lo sentimos, el archivo alice.txt no existe.
```

El programa no tiene nada más que hacer si el archivo no existe, por lo que el código de manejo de errores no añade mucho a este programa. Construyamos sobre este ejemplo y veamos cómo el manejo de excepciones puede ayudar cuando se trabaja con más de un archivo.

Análisis de textos

Puede analizar archivos de texto que contengan libros enteros. Muchas obras clásicas de la literatura están disponibles como simples archivos de texto porque son de dominio público. Los textos utilizados en esta sección proceden del Proyecto Gutenberg *(*http://gutenberg.org/). El Proyecto Gutenberg mantiene una colección de obras literarias.
ary obras que están disponibles en el dominio público, y es un gran recurso si está interesado en trabajar con textos literarios en sus proyectos de programación.

Saquemos el texto de *Alicia en el País de las Maravillas* e intentemos contar el número de palabras del texto. Utilizaremos el método `split()`, que puede construir una lista de palabras a partir de una cadena. Esto es lo que hace `split()` con una cadena que sólo contiene el título `"Alicia en el País de las Maravillas"`:

```
>>> title = "Alicia en el País de las Maravillas"
>>> title.split()
['Alicia', 'en', 'País de las Maravillas']
```

El método `split()` separa una cadena en partes siempre que encuentra un espacio y almacena todas las partes de la cadena en una lista. El resultado es una lista de palabras de la cadena, aunque también pueden aparecer algunos signos de puntuación con algunas de las palabras. Para contar el número de palabras de *Alicia en el País de las Maravillas*, utilizaremos `split()` en todo el texto. A continuación, contaremos los elementos de la lista para hacernos una idea aproximada del número de palabras del texto:

```
filename = 'alice.txt'

try:
    con open(nombre_archivo) como
        f_obj: contenido =
        f_obj.read()
```

```
excepto FileNotFoundError:
    msg = "Lo sentimos, el archivo " + nombre de archivo + " no
    existe." print(msg)
```

```
    si no:
        # Cuenta el número aproximado de palabras del fichero.
❶       palabras = contenido.split()
❷       número_palabras = len(palabras)
❸       print("El archivo " + nombre_archivo + " tiene unas " + str(num_palabras) +
" palabras.")
```

He movido el fichero *alice.txt* al directorio correcto, así que el bloque `try` funcionará esta vez. En ❶ tomamos el `contenido de la` cadena, que ahora contiene todo el texto de *Alicia en el País de las Maravillas* como una cadena larga, y usamos el método `split()` para producir una lista de todas las palabras del libro. Cuando utilizamos `len()` en esta lista para examinar su longitud, obtenemos una buena aproximación del número de palabras de la cadena original ❷. En ❸ imprimimos una sentencia que informa de cuántas palabras se han encontrado en el fichero. Este código se coloca en el bloque `else` porque sólo funcionará si el código del bloque `try` se ha ejecutado correctamente. La salida nos dice cuántas palabras hay en *alice.txt*:

```
El archivo alice.txt tiene alrededor de 29461 palabras.
```

El recuento es un poco alto porque el editor proporciona información extra en el archivo de texto utilizado aquí, pero es una buena aproximación a la longitud de *Alicia en el País de las Maravillas*.

Trabajar con varios archivos

Añadamos más libros para analizar. Pero antes de hacerlo, traslademos el grueso de este programa a una función llamada `count_words()`. Al hacerlo, será más fácil ejecutar el análisis para múltiples libros:

recuento_palabras.py

```
    def contar_palabras(nombre_archivo):
❶   """Contar el número aproximado de palabras de un
    archivo.""" inténtalo:
        con open(nombre_archivo) como
            f_obj: contenido =
            f_obj.read()
    excepto FileNotFoundError:
        msg = "Lo sentimos, el archivo " + nombre de archivo + " no
        existe." print(msg)
    si no:
        # Contar el número aproximado de palabras en el
        archivo. words = contents.split()
        número_palabras = len(palabras)
        print("El archivo " + nombre_archivo + " tiene unas " +
            str(num_palabras) + " palabras.")

nombre_archivo = 'alice.txt'
contar_palabras(nombre_archivo)
```

La mayor parte de este código no ha cambiado. Simplemente lo hemos sangrado y movido al cuerpo de `count_words()`. Es un buen hábito mantener los comentarios actualizados cuando modificas un programa, así que

cambiamos el comentario a una cadena doc- y lo reformulamos ligeramente ❶.

Ahora podemos escribir un bucle sencillo para contar las palabras de cualquier texto que queramos analizar. Para ello, almacenamos los nombres de los archivos que queremos analizar en una lista y, a continuación, llamamos a `count_words()` para cada archivo de la lista. Intentaremos contar las palabras de *Alicia en el País de las Maravillas*, *Siddhartha*, *Moby Dick* y *Mujercitas*, todos ellos de dominio público. He dejado intencionadamente *siddhartha.txt* fuera del directorio que contiene *word_count.py*, para que podamos ver lo bien que nuestro programa maneja un archivo que falta:

```
def contar_palabras(nombre_archivo):
    --snip--

nombres_de_archivo = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
'mujer_pequeña.txt'] for nombre_de_archivo in nombres_de_archivo:
    contar_palabras(nombre_archivo)
```

La falta del archivo *siddhartha.*txt no afecta al resto de la ejecución del programa:

```
El archivo alice.txt tiene alrededor de 29461
palabras. Lo sentimos, el archivo
siddhartha.txt no existe. El archivo
moby_dick.txt tiene alrededor de 215136
palabras.
El archivo mujer_pequeña.txt tiene alrededor de 189079 palabras.
```

El uso del bloque `try-except` en este ejemplo proporciona dos ventajas significativas. Evitamos que nuestros usuarios vean un rastreo, y permitimos que el programa continúe analizando los textos que es capaz de encontrar. Si no detectamos el `error FileNotFoundError` que *siddhartha.txt* generó, el usuario vería un rastreo completo, y el programa dejaría de ejecutarse después de intentar analizar *Siddhartha*. Nunca analizaría *Moby Dick* o *Mujercitas*.

Fracasar en silencio

En el ejemplo anterior, informamos a nuestros usuarios de que uno de los archivos
no estaba disponible. Pero no es necesario informar de todas las excepciones que se detecten. A veces querrás que el programa falle silenciosamente cuando ocurra una excepción y continúe como si nada hubiera pasado. Para hacer que un programa falle silenciosamente, escribe un bloque `try` como de costumbre, pero explícitamente dile a Python que no haga nada en el bloque `except`. Python tiene una sentencia `pass` que le dice que no haga nada en un bloque:

```
def contar_palabras(nombre_archivo):
    """Contar el número aproximado de palabras de un
    archivo."""  inténtalo:
        --snip--
    excepto FileNotFoundError:
❶       pase
    si no:
```

```
        --snip--

nombres_de_archivo = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
'mujer_pequeña.txt'] for nombre_de_archivo in nombres_de_archivo:
    contar_palabras(nombre_archivo)
```

La única diferencia entre este listado y el anterior es la sentencia `pass` en ❶. Ahora cuando se produce un `FileNotFoundError`, el código en el bloque `except` se ejecuta, pero no pasa nada. No se produce ningún rastreo, y no hay salida en respuesta al error que se ha producido. Los usuarios ven los recuentos de palabras para cada archivo que existe, pero no ven ninguna indicación de que no se encontró un archivo:

```
El archivo alice.txt tiene alrededor de 29461 palabras.
El archivo moby_dick.txt tiene alrededor de
215136 palabras. El archivo little_women.txt
tiene alrededor de 189079 palabras.
```

La sentencia `pass` también actúa como un marcador de posición. Es un recordatorio de que está eligiendo no hacer nada en un punto específico de la ejecución de su programa y que podría querer hacer algo allí más tarde. Por ejemplo, en este programa podríamos decidir escribir cualquier nombre de archivo que falte en un archivo llamado *missing_files.txt*. Nuestros usuarios no verían este archivo, pero nosotros podríamos leer el archivo y ocuparnos de cualquier texto que falte.

Decidir qué errores notificar

¿Cómo saber cuándo informar de un error a los usuarios y cuándo fallar en silencio? Si los usuarios saben qué textos deben analizarse, es posible que aprecien un mensaje que les informe de por qué no se analizaron algunos textos. Si los usuarios esperan ver algunos resultados pero no saben qué libros se supone que deben ser analizados, puede que no necesiten saber que algunos textos no estaban disponibles. Dar a los usuarios información que no están buscando puede disminuir la usabilidad de su programa. Las estructuras de manejo de errores de Python le dan un control fino sobre cuánto compartir con los usuarios cuando las cosas van mal; depende de usted decidir cuánta información compartir.

El código bien escrito y probado no es muy propenso a errores internos, como los de sintaxis o los lógicos. Pero cada vez que su programa depende de algo externo, como la entrada de datos del usuario, la existencia de un archivo o la disponibilidad de una conexión de red, existe la posibilidad de que se produzca una excepción. Un poco de experiencia le ayudará a saber dónde incluir bloques de manejo de excepciones en su programa y cuánto informar a los usuarios sobre los errores que surjan.

TRY IT yOURSELF

10-6. Suma: Un problema común cuando se pide una entrada numérica ocurre cuando la gente proporciona texto en lugar de números. Cuando intente convertir la entrada a un `int`, obtendrá un `TypeError`. Escriba un programa que pida dos números. Súmelos e imprima el resultado. Capture el `TypeError` si cualquiera de los valores de entrada no es un número, e imprima un mensaje de error amigable. Pruebe su programa introduciendo dos números y luego introduciendo algún texto en lugar de un número.

(continuación)

10-7. Calculadora de Sumas: Envuelva su código del Ejercicio 10-6 en un bucle `while` para que el usuario pueda continuar introduciendo números incluso si comete un error e introduce texto en lugar de un número.

10-8. Gatos y perros: Crea dos ficheros, *gatos.txt* y *perros.txt*. Guarda al menos tres nombres de gatos en el primer fichero y tres nombres de perros en el segundo. Escribe un programa que intente leer estos ficheros e imprima su contenido en la pantalla. Envuelva su código en un bloque `try-except` para capturar el error `FileNotFound`, e imprima un mensaje amigable si falta un archivo. Mueva uno de los archivos a una ubicación diferente en su sistema, y asegúrese de que el código en el bloque `except` se ejecuta correctamente.

10-9. Perros y gatos silenciosos: Modifique su bloque `except` en el Ejercicio 10-8 para que falle silenciosamente si falta alguno de los ficheros.

10-10. Palabras comunes: Visita el Proyecto Gutenberg *(http://gutenberg.org/*) y busca algunos textos que te gustaría analizar. Descarga los archivos de texto de estas obras, o copia el texto en bruto desde tu navegador a un archivo de texto en tu ordenador.

Puede utilizar el método `count()` para averiguar cuántas veces aparece una palabra o frase en una cadena. Por ejemplo, el siguiente código cuenta el número de veces que `'fila'` aparece en una cadena:

```
>>> line = "Rema, rema, rema tu barco"
>>> line.count('row')
2
>>> line.lower().count('fila')
3
```

Observe que al convertir la cadena a minúsculas mediante `lower()` se capturan todas las apariciones de la palabra buscada, independientemente de su formato.

Escribe un programa que lea los archivos que has encontrado en el Proyecto Gutenberg y determine cuántas veces aparece la palabra `'el'` en cada texto.

Almacenamiento de datos

Muchos de sus programas pedirán a los usuarios que introduzcan cierto tipo de información. Puede permitir a los usuarios almacenar preferencias en un juego o proporcionar datos para una visualización. Sea cual sea el objetivo de su programa, almacenará la información que los usuarios proporcionen en estructuras de datos como listas y diccionarios. Cuando los usuarios cierran un programa, casi siempre querrás guardar la información que han introducido. Una forma sencilla de hacerlo consiste en almacenar los datos utilizando el módulo `json`.

El módulo `json` permite volcar estructuras de datos sencillas de Python en un archivo y cargar los datos desde ese archivo la próxima vez que se ejecute el programa. También puedes utilizar `json` para compartir datos entre diferentes programas Python. Aún mejor, el formato de datos JSON no es específico de Python, por lo que puedes compartir los datos que almacenas en el formato JSON con personas que trabajan en muchos otros lenguajes de programación. Es un formato útil y portátil, y es fácil de aprender.

NOTE *El formato JSON (JavaScript Object Notation) se desarrolló originalmente para JavaScript. Sin embargo, desde entonces se ha convertido en un formato común utilizado por muchos lenguajes, incluido Python.*

Uso de `json.dump()` y `json.load()`

Escribamos un programa corto que almacene un conjunto de números y otro programa que lea estos números en memoria. El primer programa utilizará `json.dump()` para almacenar el conjunto de números, y el segundo programa utilizará `json.load()`.

La función `json.dump()` toma dos argumentos: un fragmento de datos para almacenar y un objeto de archivo que puede utilizar para almacenar los datos. A continuación se muestra cómo se puede utilizar `json.dump()` para almacenar una lista de números:

número_writer.py

```
importar json

números = [2, 3, 5, 7, 11, 13]

❶ filename = 'numbers.json'
❷ with open(nombre_archivo, 'w') as f_obj:
❸     json.dump(numbers, f_obj)
```

Primero importamos el módulo `json` y luego creamos una lista de números con la que trabajar. En ❶ elegimos un nombre de archivo en el que almacenar la lista de números. Es habitual utilizar la extensión de archivo *.json* para indicar que los datos del archivo se almacenan en formato JSON. A continuación, abrimos el archivo en modo de escritura, lo que permite a `json` escribir los datos en el archivo ❷. En ❸ utilizamos la función `json.dump()` para almacenar los `números de` la lista en el archivo *numbers.json*.

Este programa no tiene salida, pero vamos a abrir el archivo *numbers.json* y mirarlo. Los datos se almacenan en un formato que se parece a Python:

```
[2, 3, 5, 7, 11, 13]
```

Ahora vamos a escribir un programa que utiliza `json.load()` para leer la lista de nuevo en la memoria:

número_lector.py

```
importar json
```

```
❶ filename = 'numbers.json'
❷ con open(nombre_archivo) como f_obj:
❸     numbers = json.load(f_obj)

  print(números)
```

En ❶ nos aseguramos de leer del mismo archivo en el que escribimos. Esta vez cuando abrimos el archivo, lo abrimos en modo lectura porque Python sólo necesita leer del archivo ❷. En ❸ utilizamos la función `json.load()` para cargar la información almacenada en *números.json*, y la almacenamos en la variable `números`. Finalmente imprimimos la lista de números recuperada y vemos que es la misma lista creada en *number_writer.py*:

```
[2, 3, 5, 7, 11, 13]
```

Se trata de una forma sencilla de compartir datos entre dos programas.

Guardar y leer datos generados por el usuario

Guardar datos con `json` es útil cuando trabajas con datos generados por el usuario, porque si no almacenas la información de tu usuario de alguna manera, la perderás cuando el programa deje de ejecutarse. Veamos un ejemplo en el que pedimos al usuario su nombre la primera vez que ejecuta un programa y luego recordamos su nombre cuando vuelve a ejecutar el programa.

Empecemos por almacenar el nombre del usuario:

recordar_me.py

```
importar json

❶ username = input("¿Cuál es tu nombre? ")

nombre_archivo = 'nombre_usuario.json'
con open(nombre_archivo, 'w') como f_obj:
❷     json.dump(nombre_usuario, f_obj)
❸     print("¡Nos acordaremos de ti cuando vuelvas, " + nombre de usuario + "!")
```

En ❶ pedimos un nombre de usuario para almacenar. A continuación, usamos `json.dump()`, pasándole un nombre de usuario y un objeto archivo, para almacenar el nombre de usuario en un archivo ❷. Luego imprimimos un mensaje informando al usuario que hemos almacenado su información ❸:

```
¿Cómo te llamas? Eric
¡Nos acordaremos de ti cuando vuelvas, Eric!
```

Ahora escribamos un nuevo programa que salude a un usuario cuyo nombre ya ha sido almacenado:

greet_user.py

```
importar json

nombre_archivo =

'nombre_usuario.json' with
```

```
open(nombre_archivo) as

f_obj:
    nombre_usuario = json.load(f_obj)
    print("¡Bienvenido de nuevo, " + nombre de usuario + "!")
```

En ❶ usamos `json.load()` para leer la información almacenada en *nombredeusuario.json* en la variable `nombredeusuario`. Ahora que hemos recuperado el nombre de usuario, podemos darles la bienvenida de nuevo ❷:

```
¡Bienvenido, Eric!
```

Necesitamos combinar estos dos programas en un solo archivo. Cuando alguien ejecuta *remember_me.py*, queremos recuperar su nombre de usuario de la memoria si es posible; por lo tanto, comenzaremos con un bloque `try` que intenta recuperar el nombre de usuario. Si el archivo *nombre_usuario.json no* existe, haremos que el bloque `except` pida un nombre de usuario y lo almacene en *nombre_usuario.json* para la próxima vez:

recordar_me.py

```
importar json

# Cargar el nombre de usuario, si se ha almacenado
previamente. # En caso contrario, pide el nombre de
usuario y lo almacena. filename =
'nombredeusuario.json'
Inténtalo:
❶     con open(nombre_archivo) como f_obj:
❷         nombre_usuario = json.load(f_obj)
❸ except FileNotFoundError:
❹     username = input("¿Cómo te llamas? ")
❺     with open(nombre_archivo, 'w') as
          f_obj:
          json.dump(nombre_usuario,
          f_obj)
          print("¡Nos acordaremos de ti cuando vuelvas, " + nombre de usuario +
          "!")
si no:
    print("¡Bienvenido de nuevo, " + nombre de usuario + "!")
```

Aquí no hay código nuevo; simplemente se combinan bloques de código de los dos últimos ejemplos en un solo archivo. En ❶ intentamos abrir el fichero *nombredeusuario.json*. Si este fichero existe, volvemos a leer el nombre de usuario en memoria ❷ e imprimimos un mensaje de bienvenida al usuario en el bloque `else`. Si es la primera vez que el usuario ejecuta el programa, *nombredeusuario.json* no existirá y se producirá un `FileNotFoundError` ❸. Python pasará al bloque `except` donde pediremos al usuario que introduzca su nombre de usuario ❹. Luego usamos `json.dump()` para almacenar el nombre de usuario e imprimir un saludo ❺.

Cualquiera que 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, ésta es la salida:

```
¿Cómo te llamas? Eric
¡Nos acordaremos de ti cuando vuelvas, Eric!
```

Por lo demás:

```
¡Bienvenido, Eric!
```

Esta es la salida que se ve si el programa ya se ha ejecutado al menos una vez.

Refactorización

A menudo, llegarás a un punto en el que tu código funcionará, pero te darás cuenta de que podrías mejorarlo dividiéndolo en una serie de funciones con tareas específicas. Este proceso se llama *refactorización*. La refactorización hace que tu código sea más limpio, más fácil de entender y más fácil de extender.

Podemos refactorizar remember_me.*py* moviendo la mayor parte de su lógica a una o más funciones. El objetivo de remember_me.*py* es saludar al usuario, así que vamos a mover todo nuestro código existente a una función llamada `greet_user()`:

recordar_me.py

```
importar json

def saludar_usuario():
    """Saluda al usuario por su
    nombre.""" filename =
    'nombre_usuario.json' try:
        with open(nombre_archivo) as f_obj:
            nombre_usuario =
            json.load(f_obj)
    excepto FileNotFoundError:
        nombre_usuario = input("¿Cuál es tu
        nombre? ") with open(nombre_archivo, 'w')
        as f_obj:
            json.dump(nombre_usuario, f_obj)
            print("¡Nos acordaremos de ti cuando vuelvas, " + nombre de usuario
            + "!")
    si no:
        print("¡Bienvenido de nuevo, " + nombre de usuario + "!")

greet_user()
```

Como ahora estamos usando una función, actualizamos los comentarios con un docstring que refleje cómo funciona actualmente el programa ❶.

Este archivo es un poco más limpio, pero la función `greet_user()` está haciendo algo más que saludar al usuario, también está recuperando un nombre de usuario almacenado si existe y solicitando un nuevo nombre de usuario si no existe.

Vamos a refactorizar `greet_user()` para que no haga tantas tareas diferentes. Empezaremos moviendo el código para recuperar un nombre de usuario almacenado a una función separada:

```
importar json

def get_stored_username():
    """Obtener el nombre de usuario
    almacenado si está disponible."""
    filename = 'nombre_usuario.json'
    Inténtalo:
        with open(nombre_archivo) as f_obj:
            nombre_usuario =
```

```
            json.load(f_obj)
    excepto FileNotFoundError:
❷       return None
    else:
        devolver nombre de usuario
```

```
def saludar_usuario():
    """Saluda al usuario por su
    nombre""" username =
    get_stored_username()
❸   si nombre de usuario:
        print("¡Bienvenido de nuevo, " + nombre
    de usuario + "!") else:
        username = input("¿Cuál es tu nombre? ")
        filename = 'username.json'
        with open(nombre_archivo, 'w') as f_obj:
            json.dump(nombre_usuario, f_obj)
            print("¡Nos acordaremos de ti cuando vuelvas, " + nombre de usuario
            + "!")

greet_user()
```

La nueva función `get_stored_username()` tiene un propósito claro, como se indica en el docstring en ❶. Esta función recupera un nombre de usuario almacenado y devuelve el nombre de usuario si lo encuentra. Si el archivo *username.json no* existe, la función devuelve `None` ❷. Esta es una buena práctica: una función debería devolver el valor que esperas o debería devolver `Ninguno`. Esto nos permite `realizar` una prueba sencilla con el valor de retorno de la función. En ❸ imprimimos un mensaje de bienvenida al usuario si el intento de recuperar un nombre de usuario tuvo éxito, y si no lo tiene, le pedimos un nuevo nombre de usuario.

Deberíamos eliminar un bloque más de código de `greet_user()`. Si el nombre de usuario no existe, deberíamos mover el código que solicita un nuevo nombre de usuario a una función dedicada a ese propósito:

```
importar json

def get_stored_username():
    """Obtener el nombre de usuario almacenado si está disponible."""
    --snip--

def obtener_nuevo_nombre_usuario():
    """Pide un nuevo nombre de usuario."""
    username = input("¿Cuál es tu nombre?
    "") filename = 'username.json'
    with open(nombre_archivo, 'w') as f_obj:
        json.dump(nombre_usuario, f_obj)
    devolver nombre de usuario

def saludar_usuario():
    """Saluda al usuario por su
    nombre.""" nombre_usuario =
    get_stored_username() if
    nombre_usuario:
        print("¡Bienvenido de nuevo, " + nombre
    de usuario + "!") else:
        nombre_usuario = get_new_username()
```

```
        print("¡Nos acordaremos de ti cuando vuelvas, " + nombre de usuario +
        "!")

greet_user()
```

Cada función en esta versión final de *remember_me.py* tiene un único y claro propósito. Llamamos a `greet_user()`, y esa función imprime un mensaje apropiado: da la bienvenida a un usuario existente o saluda a un nuevo usuario. Lo hace llamando a `get_stored_username()`, que sólo es responsable de recuperar un nombre de usuario almacenado si existe. Finalmente, `greet_user()` llama a `get_new_username()` si es necesario, que sólo es responsable de obtener un nuevo nombre de usuario y almacenarlo. Esta compartimentación del trabajo es una parte esencial para escribir un código claro que sea fácil de mantener y extender.

TRY IT yOURSELF

10-11. Número favorito: Escribe un programa que pregunte por el número favorito del usuario. Utilice `json.dump()` para almacenar este número en un archivo. Escriba un programa separado que lea este valor e imprima el mensaje "¡Conozco tu número favorito! Es _____."

10-12. Número Favorito Recordado: Combine los dos programas del Ejercicio 10-11 en un solo archivo. Si el número ya está almacenado, informe el número favorito al usuario. Si no, pregunte por el número favorito del usuario y guárdelo en un fichero. Ejecute el programa dos veces para comprobar que funciona.

10-13. Verificar usuario: El listado final de *remember_me.py* asume que el usuario ya ha introducido su nombre de usuario o que el programa se está ejecutando por primera vez. Deberíamos modificarlo en caso de que el usuario actual no sea la persona que utilizó el programa por última vez.

Antes de imprimir un mensaje de bienvenida en `greet_user()`, pregunta al usuario si este es el nombre de usuario correcto. Si no lo es, llame a `get_new_username()` para obtener el nombre de usuario correcto.

Resumen

En este capítulo has aprendido a trabajar con ficheros. Aprendiste a leer un archivo completo de una sola vez y a leer el contenido de un archivo línea por línea. Aprendiste a escribir en un archivo y a agregar texto al final de un archivo. Leíste sobre excepciones y cómo manejar las excepciones que probablemente veas en tus programas. Finalmente, aprendiste cómo almacenar estructuras de datos Python para que puedas guardar la información que tus usuarios te proporcionen, evitando que tengan que empezar de nuevo cada vez que ejecuten un programa.

En el capítulo 11 aprenderás formas eficientes de probar tu código. Esto te ayudará a confiar en que el código que desarrollas es correcto, y te ayudará a identificar errores que se introducen a medida que continúas construyendo sobre los programas que has escrito.

11

TESTING YOUR CODE



Cuando escribes una función o una clase, también puedes escribir pruebas para ese código. Las pruebas demuestran que el código funciona como es debido.

en respuesta a todos los tipos de entrada que está diseñado para recibir. Cuando escribes pruebas, puedes estar seguro de que tu código funcionará correctamente a medida que más gente empiece a utilizar tus programas. También podrá probar nuevo código a medida que lo añades para asegurarte de que tus cambios no rompen el comportamiento existente de tu programa. Todos los programadores cometen errores, por lo que deben probar su código a menudo para detectar problemas antes de que los usuarios los encuentren.

En este capítulo aprenderás a probar tu código utilizando las herramientas del módulo `unittest` de Python. Aprenderás a construir un caso de prueba y comprobar que un conjunto de entradas da como resultado la salida que deseas. Verás cómo se ve una prueba que pasa y cómo se ve una prueba que falla, y aprenderás cómo una prueba que falla puede ayudarte a mejorar tu código. Aprenderás a probar funciones y

clases, y empezarás a entender cuántas pruebas escribir para un proyecto.

Probar una función

Para aprender a probar, necesitamos código que probar. He aquí una función sencilla que recibe un nombre y un apellido y devuelve un nombre completo con un formato claro:

nombre_function.py

```
def obtener_nombre_formateado(nombre, apellido):
    """Generar un nombre completo bien
    formateado.""" full_name = first + ' ' +
    last
    return nombre_completo.titulo()
```

La función `get_formatted_name()` combina el nombre y el apellido con un espacio en medio para completar un nombre completo, y luego escribe en mayúsculas y devuelve el nombre completo. Para comprobar que get_formatted_name() funciona, vamos a crear un programa que utilice esta función. El programa *nombres.py* permite a los usuarios introducir un nombre y un apellido, y ver un nombre completo formateado ordenadamente:

nombres.py

```
from nombre_funcion import get_nombre_formateado

print("Introduce 'q' en cualquier momento
para salir") while True:
    first = input("\nPor favor, dame un nombre: ") if
    first == 'q':
        romper
    last = input("Por favor, indíqueme un
    apellido: ") if last == 'q':
        romper

    nombre_formateado = nombre_formateado(nombre, apellido)
    print("Nombre formateado: " + nombre_formateado + '.')
```

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 formateados que se generan:

```
Introduzca 'q' en cualquier

momento para salir. Por favor,

dame un nombre: janis
Por favor, dame un apellido: joplin
        Nombre bien formateado: Janis Joplin.

Por favor, dame un nombre de pila: bob
Por favor, dame un apellido: dylan
        Nombre bien formateado: Bob Dylan.

Por favor, dígame su nombre: q
```

Podemos ver que los nombres generados aquí son correctos. Pero digamos

que queremos modificar `get_formatted_name()` para que también pueda manejar segundos nombres.
Al hacerlo, queremos asegurarnos de que no rompemos la forma en que la función maneja los nombres que sólo tienen nombre y apellidos. Podríamos probar nuestro código

ejecutando *names.py* e introduciendo un nombre como `Janis Joplin` cada vez que modificamos `get_formatted_name()`, pero eso se volvería tedioso. Afortunadamente, Python proporciona una forma eficiente de automatizar la comprobación de la salida de una función. Si automatizamos las pruebas de get_formatted_name(), siempre podemos estar seguros de que la función funcionará cuando se le den los tipos de nombres para los que hemos escrito las pruebas.

Pruebas unitarias y casos de prueba

El módulo `unittest de` la biblioteca estándar de Python proporciona herramientas para probar el código. Una prueba *unitaria* verifica que un aspecto específico del comportamiento de una función es correcto. Un caso de *prueba* es una colección de pruebas unitarias que juntas prueban que una función se comporta como se supone que debe h a c e r l o , dentro del rango completo de situaciones que esperas que maneje. Un buen caso de prueba tiene en cuenta todos los tipos posibles de entrada que puede recibir una función e incluye pruebas para representar cada una de estas situaciones. Un caso de prueba con cobertura *total* incluye una gama completa de pruebas unitarias que cubren todas las formas posibles de utilizar una función. Conseguir una c o b e r t u r a c o m p l e t a en un proyecto grande puede ser desalentador. A menudo es suficiente con escribir pruebas para los comportamientos críticos del código y luego aspirar a una cobertura total sólo si el proyecto empieza a tener un uso generalizado.

Una prueba superada

La sintaxis para configurar un caso de prueba requiere un poco de tiempo para acostumbrarse, pero una vez que haya configurado el caso de prueba es sencillo añadir más pruebas unitarias para sus funciones. Para escribir un caso de prueba para una función, importe el módulo `unittest` y la función que desea probar. A continuación, cree una clase que herede de `unittest.TestCase`, y escriba una serie de métodos para probar diferentes aspectos del comportamiento de su función.

He aquí un caso de prueba con un método que verifica que la función `get_formatted_name()` funciona correctamente cuando se le da un nombre y un apellido:

nombre_prueba function.py

```
importar unittest
from nombre_funcion import get_nombre_formateado

❶ class NamesTestCase(unittest.TestCase):
    """Pruebas para
    'nombre_funcion.py'."""

    def prueba_nombre_apellido(self):
        """¿Funcionan nombres como 'Janis Joplin'?"""
❷       nombre_formateado = get_nombre_formateado('janis', 'joplin')
❸       self.assertEqual(nombre_formateado, 'Janis Joplin')

unittest.main()
```

En primer lugar, importamos `unittest` y la función que queremos probar, get_formatted_name(). En ❶ creamos una clase llamada `NamesTestCase`, que contendrá una serie de pruebas unitarias para get_formatted_name(). Puedes ponerle a la clase el nombre que quieras

que quieras, pero es mejor llamarla algo relacionado con la función que vas a probar y usar la palabra *Test* en el nombre de la clase. Esta clase debe heredar de la clase `unittest.TestCase` para que Python sepa cómo ejecutar las pruebas que escribas.

`NamesTestCase` contiene un único método que comprueba un aspecto de `get_formatted_name()`. Llamamos a este método `test_first_last_name()` porque estamos verificando que los nombres que sólo tienen nombre y apellido se formatean correctamente. Cualquier método que empiece por `test_` se ejecutará automáticamente cuando ejecutemos *test_nombre_funcion.py*. Dentro de este método de prueba, llamamos a la función que queremos probar y almacenamos un valor de retorno que nos interese probar. En este ejemplo llamamos a `obtener_nombre_formateado()` con los argumentos `'janis'` y `'joplin'`, y almacenamos el resultado en `nombre_formateado` ❷.

En ❸ usamos una de las características más útiles de `unittest`: un método *assert*. Los métodos assert verifican que un resultado que has recibido coincide con el resultado que esperabas recibir. En este caso, como sabemos que `get_formatted_name()` debe devolver un nombre completo en mayúsculas y espaciado correctamente, esperamos que el valor de `formatted_name` sea `Janis Joplin`. Para comprobar si esto es cierto, utilizamos el método `assertEqual()` de `unittest` y le pasamos `formatted_name` y `'Janis Joplin'`. La línea

```
self.assertEqual(nombre_formateado, 'Janis Joplin')
```

dice: "Compare el valor en `formatted_name` con la cadena `'Janis Joplin'`. Si son iguales como se esperaba, bien. Pero si no coinciden, ¡házmelo saber!".

La línea `unittest.main()` indica a Python que ejecute las pruebas de este archivo. Cuando ejecutamos *test_nombre_funcion.py*, obtenemos la siguiente salida:

```
.
----------------------------------------------------------------------
Ejecutó 1 prueba en

0.000s OK
```

El punto en la primera línea de salida nos indica que se ha superado una única prueba. La siguiente línea nos dice que Python ejecutó una prueba, y que tardó menos de 0.001 segundos en ejecutarse. El `OK` final nos dice que todas las pruebas unitarias del caso de prueba se han superado.

Este resultado indica que la función `get_formatted_name()` siempre funcionará para nombres que tengan nombre y apellidos a menos que modifiquemos la `función`. Cuando modifiquemos get_formatted_name(), podremos volver a ejecutar esta prueba. Si el caso de prueba pasa, sabemos que la función seguirá funcionando para nombres como Janis Joplin.

Una prueba fallida

¿Qué aspecto tiene una prueba que falla? Modifiquemos `get_formatted_name()` para que pueda manejar segundos nombres, pero lo haremos de forma que la función falle para nombres con sólo nombre y apellido, como Janis Joplin.

He aquí una nueva versión de `get_formatted_name()` que requiere un argumento de segundo nombre:

nombre function.py

```
def obtener_nombre_formateado(nombre, medio, apellido):
    """Generar un nombre completo con formato
    ordenado."""  nombre_completo = first + ' ' +
    middle + ' ' + last return
    nombre_completo.title()
```

Esta versión debería funcionar para personas con segundo nombre, pero cuando la probamos, vemos que hemos roto la función para personas con sólo nombre y apellidos. Esta vez, al ejecutar el archivo *test_name_function.py se obtiene* este resultado:

```
❶ E
======================================================================
❷ ERROR: test_first_last_name ( main .NamesTestCase)
----------------------------------------------------------------------
❸ Traceback (most recent call last):
    File "test_name_function.py", line 8, in test_first_last_name
      nombre_formateado = get_formatted_name('janis', 'joplin')
TypeError: get_formatted_name() missing 1 required positional argument: 'last'

----------------------------------------------------------------------
❹ Realizó 1 prueba en 0.000s

❺ FALLÓ (errores=1)
```

Hay un montón de información aquí, porque hay mucho que usted puede necesitar saber cuando una prueba falla. El primer elemento en la salida es una sola `E` ❶, que nos dice que una prueba unitaria en el caso de prueba dio lugar a un e r r o r. A continuación, vemos que `test_first_last_name()` en `NamesTestCase` causó un error ❷. Saber qué prueba falló es crítico cuando tu caso de prueba contiene muchas pruebas unitarias. En ❸ vemos un traceback estándar, que informa de que la llamada a la función `get_formatted_name('janis', 'joplin')` ya no funciona porque le falta un argumento posicional requerido.

También vemos que se ejecutó una prueba unitaria ❹. Por último, vemos un mensaje adicional que indica que el caso de prueba general falló y que se produjo un error al ejecutar el caso de prueba ❺. Esta información aparece al final de la salida para que la veas de inmediato; no quieres desplazarte a través de una larga lista de salida para averiguar cuántas pruebas fallaron.

Respuesta a un examen fallido

¿Qué hacer cuando falla una prueba? Suponiendo q u e esté comprobando las condiciones correctas, una prueba positiva significa que la función se comporta correctamente y una prueba negativa significa que hay un error

en el nuevo código que ha escrito. Así que cuando una prueba

falla, no cambie la prueba. En su lugar, corrija el código que ha hecho que falle la prueba. Examina los cambios que acabas de hacer en la función, y averigua cómo esos cambios rompieron el comportamiento deseado.

En este caso, `get_formatted_name()` solía requerir sólo dos parámetros: un nombre y un apellido. Ahora requiere un nombre, un segundo nombre y un apellido. La adición de ese parámetro obligatorio del segundo nombre rompió el comportamiento deseado de get_formatted_name(). La mejor opción es hacer que el segundo nombre sea opcional. Una vez que lo hagamos, nuestra prueba para nombres como `Janis Joplin` debería pasar de nuevo, y deberíamos ser capaces de aceptar segundos nombres también. Modifiquemos `get_formatted_name()` para que los segundos nombres sean opcionales y volvamos a ejecutar el caso de prueba. Si pasa, pasaremos a asegurarnos de que la función maneja los segundos nombres correctamente.

Para que los segundos nombres sean opcionales, movemos el parámetro `middle` al final de la lista de parámetros en la definición de la función y le damos un valor vacío por defecto. También añadimos una prueba `if` que construye el nombre completo correctamente, dependiendo de si se proporciona o no un segundo nombre:

nombre_function.py

```
def get_formatted_name(first, last, middle=''):
    """Generar un nombre completo bien
    formateado.""" if middle:
        full_name = first + ' ' + middle + ' ' + last
    else:
        nombre_completo = first + ' '
    + last return
    nombre_completo.title()
```

En esta nueva versión de `get_formatted_name(), el segundo` nombre es opcional. Si se pasa un segundo nombre a la función (`si middle:`), el nombre completo contendrá un nombre, un segundo nombre y un apellido. En caso contrario, el nombre completo consistirá únicamente en el nombre y los apellidos. Ahora la función debería funcionar para ambos tipos de nombres. Para saber si la función sigue funcionando con nombres como `Janis Joplin`, volvamos a ejecutar *test_nombre_funcion.py*:

```
.
----------------------------------------------------------------------
Ejecutó 1 prueba en

0.000s OK
```

El caso de prueba pasa ahora. Esto es ideal; significa que la función funciona de nuevo para nombres como `Janis Joplin` sin que tengamos que probar la función manualmente. Arreglar nuestra función fue fácil porque la prueba fallida nos ayudó a identificar el nuevo código que rompía el comportamiento existente.

Añadir nuevas pruebas

Ahora que sabemos que `get_formatted_name()` vuelve a funcionar para nombres simples, escribamos una segunda prueba para personas que incluyen un segundo nombre. Para ello, añadimos otro método a la clase `NamesTestCase`:

```
importar unittest
from nombre_funcion import get_nombre_formateado

class NamesTestCase(unittest.TestCase):
    """Pruebas para
    'nombre_funcion.py'."""

    def prueba_nombre_apellido(self):
        """¿Funcionan nombres como 'Janis Joplin'?"""
        formatted_name = get_formatted_name('janis', 'joplin')
        self.assertEqual(formatted_name, 'Janis Joplin')

    def prueba_primer_apellido_medio_nombre(self):
        """¿Funcionan nombres como 'Wolfgang Amadeus Mozart'?"""
❶       nombre_formateado = get_nombre_formateado(
            wolfgang', 'mozart', 'amadeus')
        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

unittest.main()
```

Llamamos a este nuevo método `test_first_last_middle_name()`. El nombre del método debe empezar por *test_* para que el método se ejecute automáticamente cuando ejecutemos *test_nombre_funcion.py*. Nombramos el método para dejar claro qué comportamiento de `get_formatted_name()` estamos probando. Como resultado, si la prueba falla, sabemos de inmediato qué tipo de nombres se ven afectados. Está bien tener nombres largos de métodos en sus clases `TestCase`. Necesitan ser descriptivos para que puedas entender la salida cuando tus pruebas fallen, y como Python los llama automáticamente, nunca tendrás que escribir código que llame a estos métodos.

Para probar la función, llamamos a `get_formatted_name()` con un nombre, apellido y segundo nombre ❶, y luego usamos `assertEqual()` para comprobar que el nombre completo devuelto coincide con el nombre completo (nombre, segundo nombre y apellido) que esperamos. Cuando volvemos a ejecutar *test_name_function.py*, ambas pruebas pasan:

```
..
----------------------------------------------------------------
Ejecutó 2 pruebas en

0.000s OK
```

¡Genial! Ahora sabemos que la función sigue funcionando para nombres como `Janis Joplin`, y podemos estar seguros de que también funcionará para nombres como `Wolfgang Amadeus Mozart`.

TRY IT yOURSELF

11-1. Ciudad, País: Escriba una función que acepte dos parámetros: el nombre de una ciudad y el nombre de un país. La función debe devolver una única cadena de la forma *Ciudad, País*, como por ejemplo `Santiago de Chile`. Guarda la función en un módulo llamado *city_functions.py*.

Crea un archivo llamado *test_cities.py* que pruebe la función que acabas de escribir (recuerda que necesitas importar `unittest` y la función que quieres probar). Escribe un método llamado `test_city_country()` para verificar que al llamar a tu función con valores como `'santiago'` y `'chile'` se obtiene la cadena correcta. Ejecuta *test_cities.py*, y asegúrate de que `test_city_country()` pasa.

11-2. Población: Modifique su función para que requiera un tercer parámetro, `población`. Ahora debería devolver una única cadena de la forma *Ciudad, País - población xxx*, como `Santiago de Chile - población 5000000`. Ejecute *test_cities.py* de nuevo. Asegúrate de que `test_city_country()` falla esta vez.

Modifica la función para que el parámetro `población` sea opcional. Ejecute *test_cities.py* de nuevo, y asegúrate de que `test_city_country()` pasa de nuevo. Escribe una segunda prueba llamada `test_ciudad_país_población()` que verifies puedes llamar a tu función con los valores `'santiago'`, `'chile'`, y `'population=5000000'`. Ejecuta *test_cities.py* de nuevo, y asegúrate de que esta nueva prueba pasa.

Probar una clase

En la primera parte de este capítulo, escribiste pruebas para una única función. Ahora escribirá pruebas para una clase. Utilizará clases en muchos de sus propios programas, por lo que es útil ser capaz de probar que sus clases funcionan correctamente. Si tienes pruebas que pasan para una clase en la que estás trabajando, puedes estar seguro de que las mejoras que hagas a la clase no romperán accidentalmente su comportamiento actual.

Variedad de métodos Assert

Python proporciona una serie de métodos assert en la clase `unittest.TestCase`. Como se mencionó anteriormente, los métodos assert prueban si una condición que usted cree que es verdadera en un punto específico de su código es realmente verdadera. Si la condición es `verdadera` como se esperaba, se confirma su suposición sobre cómo se comporta esa parte de su programa; puede estar seguro de que no existen errores. Si la condición que supones que es cierta en realidad no lo es, Python lanza una excepción.

La Tabla 11-1 describe seis métodos assert de uso común. Con estos métodos puede verificar que los valores devueltos son iguales o no a los valores esperados, que los valores son `True` o `False`, y que los valores `están` o `no dentro de` un determinado campo

lista. Sólo puede utilizar estos métodos en una clase que herede de `unittest.TestCase`, así que vamos a ver cómo podemos utilizar uno de estos métodos en el con- texto de probar una clase real.

Tabla 11-1: Métodos Assert disponibles en el módulo `unittest`

Método	Utilice
`assertEqual(a, b)`	Comprueba que a
`== b assertNotEqual(a, b)`	Comprueba que a
`!= b assertTrue(x)`	Comprueba si x
es verdadero	
`assertFalse(x)`	Comprueba que x es
Falso `assertIn(`*item*`, `*list*`)`	Comprueba que *el*
elemento está en la *lista* `assertNotIn(`*elemento*`,`	
lista`)`	Comprueba que el
elemento no está en la *lista*	

Una clase para probar

Probar una clase es similar a probar una función: gran parte del trabajo consiste en probar el comportamiento de los métodos de la clase. Pero hay algunas diferencias, así que vamos a escribir una clase para probar. Consideremos una clase que ayuda a administrar encuestas anónimas:

encuesta.py

```
clase EncuestaAnónima():
"""Recopilar respuestas anónimas a una pregunta de la encuesta."""

❶    def init (self, question):
         """Almacena una pregunta y prepárate para almacenar las
         respuestas.""" self.question = question
         self.responses = []

❷    def mostrar_pregunta(self):
         """Mostrar la pregunta de la
         encuesta.""" print(question)

❸    def almacenar_respuesta(self, nueva_respuesta):
         """Almacena una única respuesta a la encuesta."""
         self.responses.append(nueva_respuesta)

❹    def mostrar_resultados(self):
         """Mostrar todas las respuestas que se han dado."""
         print("Resultados de la encuesta:"")
         para respuesta en
             respuestas: print('- '
             + respuesta)
```

Esta clase comienza con una pregunta de encuesta que usted proporciona ❶ e incluye una lista vacía para almacenar respuestas. La clase tiene métodos

para imprimir la pregunta de la encuesta ❷, añadir una nueva respuesta a la lista de respuestas ❸, e imprimir todas las respuestas almacenadas en la lista ❹. Para crear una instancia de esta clase, todo lo que hay que hacer es

es una pregunta. Una vez que se tiene una instancia que representa una encuesta concreta, se muestra la pregunta de la encuesta con `show_question()`, se almacena una respuesta con `store_response()` y se muestran los resultados con `show_results()`.

Para demostrar que la clase `AnonymousSurvey` funciona, vamos a escribir un programa que utilice la clase:

idioma encuesta.py

```
from encuesta import EncuestaAnónima

# Define una pregunta y haz una encuesta.
question = "¿En qué idioma aprendió a hablar por primera vez?"
my_survey = AnonymousSurvey(question)

# Mostrar la pregunta, y almacenar las respuestas a la pregunta.
my_survey.show_question()
print("Introduce 'q' en cualquier momento
para salir.\n") while True:
    response = input("Idioma: ")
    if response == 'q':
        break
    mi_encuesta.almacenar_respuesta(res
    puesta)

# Mostrar los resultados de la encuesta.
print("¡Gracias a todos los que han participado en la encuesta!")
my_survey.show_results()
```

Este programa define una pregunta (`"¿En qué idioma aprendió a hablar por primera vez?"`) y crea un objeto `AnonymousSurvey` con esa pregunta. La dirección
llama a `show_question()` para mostrar la pregunta y luego pide respuestas. Cada respuesta se almacena a medida que se recibe. Cuando se han introducido todas las respuestas (el usuario introduce `q` para salir), `show_results()` imprime los resultados de la encuesta:

```
¿En qué idioma aprendió a hablar por
primera vez? Introduce 'q' en cualquier
momento para salir.

Language: Inglés
Lengua: Español
Lengua: Inglés
Idioma: Mandarín
Idioma: q

Gracias a todos los que han participado en la encuesta.
Resultados de la encuesta:
- Inglés
- Español
- Inglés
- Mandarín
```

Esta clase funciona para una simple encuesta anónima. Pero digamos que queremos mejorar `AnonymousSurvey` y el módulo en el que está, `survey`. Podríamos permitir a cada usuario introducir más de una respuesta. Podríamos escribir un método para listar sólo las respuestas únicas e informar de cuántas veces se ha dado cada respuesta. Podríamos escribir otra clase para gestionar las encuestas no anónimas.

Implementar tales cambios podría afectar al comportamiento actual de la clase `EncuestaAnónima`. Por ejemplo, es posible que al tratar de permitir que cada usuario introduzca múltiples respuestas, podríamos cambiar accidentalmente cómo se manejan las respuestas individuales. Para asegurarnos de que no rompemos el comportamiento existente mientras desarrollamos este módulo, podemos escribir pruebas para la clase.

Prueba de la clase AnonymousSurvey

Escribamos una prueba que verifique un aspecto de la forma en que se comporta `AnonymousSurvey`. Escribiremos una prueba para verificar que una única respuesta a la pregunta de la encuesta se almacena correctamente. Utilizaremos el método `assertIn()` para verificar que la respuesta está en la lista de respuestas después de haber sido almacenada:

test_encuesta.py

```
importar unittest
from encuesta import EncuestaAnónima

❶ class PruebaEncuestaAnónima(unittest.TestCase):
    """Pruebas para la clase EncuestaAnónima""".

❷   def test_store_single_response(self):
        """Prueba de que una sola respuesta se almacena
        correctamente""" pregunta = "¿En qué idioma aprendiste a
        hablar por primera vez?""
❸       my_survey = AnonymousSurvey(question)
        my_survey.store_response('Español')

❹       self.assertIn('Español',

mi_encuesta.respuestas) unittest.main()
```

Comenzamos importando el módulo `unittest` y la clase que queremos probar, `AnonymousSurvey`. Llamamos a nuestro caso de prueba `TestAnonymousSurvey`, que de nuevo hereda de `unittest.TestCase` ❶. El primer método de prueba verificará que cuando almacenamos una respuesta a la pregunta de la encuesta, la respuesta termina en la lista de respuestas de la encuesta. Un buen nombre descriptivo para este método es `test_store_single_response()` ❷. Si esta prueba falla, sabremos por el nombre del método que aparece en la salida de la prueba que hubo un problema al almacenar una sola respuesta a la encuesta.

Para probar el comportamiento de una clase, necesitamos hacer una instancia de la clase. En ❸ creamos una instancia llamada `mi_encuesta` con la pregunta `"¿En qué idioma aprendiste a hablar por primera vez?"`.

Almacenamos una única respuesta, inglés, utilizando el método `store_response()`. Luego verificamos que la respuesta se almacenó correctamente afirmando que `English` está en la lista `my_survey.responses` ❹.

Cuando ejecutamos *test_survey.py*, la prueba pasa:

```
.
----------------------------------------------------------------------
Ejecutó 1 prueba en

0,001s OK
```

Esto está bien, pero una encuesta sólo es útil si genera más de una respuesta. Verifiquemos que se pueden almacenar tres respuestas correctamente. Para ello, añadimos otro método a `TestAnonymousSurvey`:

```
importar unittest
from encuesta import EncuestaAnónima

class PruebaEncuestaAnónima(unittest.TestCase):
    """Pruebas para la clase EncuestaAnónima"""

    def test_store_single_response(self):
        """Prueba de que una única respuesta se almacena correctamente."""
        --snip--

    def prueba_almacenar_tres_respuestas(self):
        """Comprueba que tres respuestas individuales se almacenan
        correctamente""" question = "¿En qué idioma aprendiste a hablar
        por primera vez?" my_survey = AnonymousSurvey(question)
❶       responses = ['Inglés', 'Español', 'Mandarín']
        for response in responses:
            mi_encuesta.almacenar_respuesta(respuesta)

❷       para respuesta en respuestas:
            self.assertIn(respuesta,
            mi_encuesta.respuestas)

unittest.main()
```

Llamamos al nuevo método `test_store_three_responses()`. Creamos un objeto sur- vey igual que hicimos en `test_store_single_response()`.

Definimos una lista que contiene tres respuestas ❶ diferentes, y luego llamamos a `store_response()` para cada una de estas respuestas. Una vez almacenadas las respuestas, escribimos otro bucle y afirmamos que cada respuesta está ahora en `mi_encuesta.respuestas` ❷.

Cuando volvemos a ejecutar *test_survey.py*, ambas pruebas (para una sola respuesta y para tres respuestas) pasan:

```
..
----------------------------------------------------------------------
Ejecutó 2 pruebas en

0.000s OK
```

Esto funciona perfectamente. Sin embargo, estas pruebas son un poco repetitivas, así que utilizaremos otra característica de `unittest` para hacerlas más eficientes.

El método setUp()

En *test_survey.py* creamos una nueva instancia de `AnonymousSurvey` en cada método de prueba, y creamos nuevas respuestas en cada método. La clase `unittest.TestCase` tiene un método `setUp()` que te permite crear estos objetos una vez y luego utilizarlos en cada uno de tus métodos de prueba. Cuando incluyes un método setUp() en una clase `TestCase`, Python ejecuta el método `setUp()` antes de ejecutar cada método que comienza con *test_*. Cualquier objeto creado en el método `setUp()` estará disponible en cada método de prueba que escribas.

Utilicemos `setUp()` para crear una instancia de encuesta y un conjunto de respuestas que puedan utilizarse en `test_store_single_response()` y `test_store_three_responses()`:

```
importar unittest
from encuesta import EncuestaAnónima

class PruebaEncuestaAnónima(unittest.TestCase):
    """Pruebas para la clase EncuestaAnónima""".

    def setUp(self):
        """
        Crear una encuesta y un conjunto de respuestas para su uso en todos
        los métodos de prueba. """
        pregunta = "¿En qué lengua aprendió a hablar por primera vez?"
❶       self.mi_encuesta = EncuestaAnonima(pregunta)
❷       self.responses = ['Inglés', 'Español', 'Mandarín']

    def test_store_single_response(self):
        """Comprueba que una única respuesta se almacena
        correctamente."""
        self.mi_encuesta.almacenar_respuesta(self.respuestas[0])
        self.assertIn(self.respuestas[0], self.mi_encuesta.respuestas)

    def prueba_almacenar_tres_respuestas(self):
        """Comprueba que tres respuestas individuales se almacenan
        correctamente.""" for response in self.responses:
            self.my_survey.store_response(response)
        for response in self.responses:
            self.assertIn(respuesta, self.mi_encuesta.respuestas)

unittest.main()
```

El método `setUp()` hace dos cosas: crea una instancia de encuesta ❶, y crea una lista de respuestas ❷. Cada uno de ellos lleva el prefijo `self`, por lo que se pueden utilizar en cualquier parte de la clase. Esto hace que los dos métodos de prueba sean más simples, porque ninguno tiene que hacer una instancia de encuesta o una respuesta. El método `test_store_single_response()` comprueba que la primera respuesta de self.`responses` -self.`responses[0]`- se puede almacenar correctamente, y `test_store_ single_response()` comprueba que las tres respuestas de

`self.responses` se pueden almacenar correctamente.

Cuando ejecutamos *test_survey.py* de nuevo, ambas pruebas siguen pasando. Estas pruebas serían particularmente útiles cuando se trata de ampliar `AnonymousSurvey` para manejar múltiples respuestas para cada persona. Después de modificar el código para aceptar múltiples

podrías realizar estas pruebas y asegurarte de que no has afectado a la capacidad de almacenar una única respuesta o una serie de respuestas individuales.

Al probar sus propias clases, el método `setUp()` puede facilitar la escritura de sus métodos de prueba. Usted crea un conjunto de instancias y atributos en setUp() y luego utiliza estas instancias en todos sus métodos de prueba. Esto es mucho más fácil que hacer un nuevo conjunto de instancias y atributos en cada método de prueba.

NOTE *Cuando se ejecuta un caso de prueba, Python imprime un carácter por cada prueba de unidad que se completa. Una prueba que pasa imprime un punto, una prueba que resulta en un error imprime una E, y una prueba que resulta en una aserción fallida imprime una F. Esta es la razón por la que verás un número diferente de puntos y caracteres en la primera línea de salida cuando ejecutes tus casos de prueba. Si un caso de prueba tarda mucho tiempo en ejecutarse porque contiene muchas pruebas unitarias, puede observar estos resultados para tener una idea de cuántas pruebas están pasando.*

TRY IT yOURSELF

11-3. Empleado: Escribe una clase llamada `Empleado`. El método `init ()` debería tomar un nombre, un apellido y un salario anual, y almacenar cada uno de ellos como atributos. Escribe un método llamado `dar_aumento()` que añada $5000 al salario anual por defecto pero que también acepte una cantidad de aumento diferente.

Escribe un caso de prueba para `Empleado`. Escribe dos métodos de prueba, `test_give_ default_raise()` y `test_give_custom_raise()`. Utiliza el método `setUp()` para no tener que crear una nueva instancia de empleado en cada método de prueba. Ejecute su caso de prueba, y asegúrese de que ambas pruebas pasan.

Resumen

En este capítulo aprendiste a escribir pruebas para funciones y clases utilizando las herramientas del módulo `unittest`. Aprendiste a escribir una clase que hereda de `unittest.TestCase`, y aprendiste a escribir métodos de prueba que verifican comportamientos específicos que tus funciones y clases deben exhibir. Aprendió a utilizar el método `setUp()` para crear eficientemente instancias y atributos de sus clases que pueden ser utilizados en todos los métodos de prueba para una clase.

Las pruebas son un tema importante que muchos principiantes no aprenden. No tienes que escribir pruebas para todos los proyectos sencillos que intentes como principiante. Pero tan pronto como empieces a trabajar en proyectos que impliquen un esfuerzo de desarrollo significativo, deberías probar los comportamientos críticos de tus funciones y clases. Estarás más seguro de que el nuevo trabajo en tu proyecto no romperá las partes que funcionan, y esto te dará libertad para hacer mejoras en tu

código. Si accidentalmente rompes una funcionalidad existente, lo sabrás de inmediato, por lo que podrás solucionar el problema fácilmente. Responder a una prueba fallida que has ejecutado es mucho más fácil que responder a un informe de error de un usuario descontento.

Otros programadores respetan más tus proyectos si incluyes algunas pruebas ini- ciales. Se sentirán más cómodos experimentando con tu código y estarán más dispuestos a trabajar contigo en los proyectos. Si quieres contribuir a un proyecto en el que están trabajando otros programadores, se espera que demuestres que tu código supera las pruebas existentes y, por lo general, se espera que escribas pruebas para los nuevos comportamientos que introduzcas en el proyecto.

Juega con las pruebas para familiarizarte con el proceso de comprobación de tu código. Escribe pruebas para los comportamientos más críticos de tus funciones y clases, pero no busques una cobertura total en los primeros proyectos a menos que tengas una razón específica para hacerlo.

PARTE II

PROJECTOS

¡Enhorabuena! Ahora sabes lo suficiente sobre Python como para empezar a construir proyectos interactivos y significativos. Crear tus propios proyectos te enseñará nuevas habilidades y consolidará tu comprensión de los conceptos introducidos en la Parte I.

La Parte II contiene tres tipos de proyectos, y puedes elegir hacer cualquiera de ellos o todos en el orden que quieras. Aquí tienes una breve descripción de cada proyecto para ayudarte a decidir en cuál profundizar primero.

Invasión alienígena: Crear un juego con Python

En el proyecto Invasión alienígena (Capítulos 12, 13 y 14), utilizarás el paquete Pygame para desarrollar un juego 2D en el que el objetivo es derribar una flota de alienígenas a medida que descienden por la pantalla en niveles que aumentan en velocidad y dificultad. Al final del proyecto, habrás aprendido habilidades que te permitirán desarrollar tus propios juegos 2D en Pygame.

Visualización de datos

El proyecto de visualización de datos comienza en el capítulo 15, en el que aprenderás a generar datos y a crear una serie de visualizaciones funcionales y atractivas de esos datos utilizando matplotlib y Pygal. El capítulo 16 te enseña a acceder a datos de fuentes en línea y a introducirlos en un paquete de visualización para crear gráficos de datos meteorológicos y un mapa de la población mundial. Por último, el capítulo 17

le muestra cómo escribir un programa para descargar y visualizar datos automáticamente. Aprender a hacer visualizaciones le permite explorar el campo de la minería de datos, que es una habilidad muy buscada en el mundo de hoy.

Aplicaciones web

En el proyecto Aplicaciones Web (Capítulos 18, 19 y 20), utilizarás el paquete Django para crear una aplicación web sencilla que permita a los usuarios llevar un diario sobre cualquier tema que hayan estado aprendiendo. Los usuarios crearán una cuenta con un nombre de usuario y una contraseña, introducirán un tema, y luego harán entradas sobre lo que están aprendiendo. También aprenderás a desplegar tu aplicación para que cualquier persona del mundo pueda acceder a ella.

Después de completar este proyecto, serás capaz de empezar a construir tus propias aplicaciones web simples, y estarás listo para profundizar en recursos más completos sobre la construcción de aplicaciones con Django.

PROYECTO 1

ALIEN INVASIÓN

12

UNA HIPA QUE FIRES BULLETES



¡Vamos a construir un juego! Usaremos Pygame, una colección de divertidos y potentes módulos de Python que gestionan gráficos, animaciones e incluso sonido, lo que facilita la creación de juegos sofisticados. Gracias a que Pygame se encarga de tareas como dibujar imágenes en la pantalla, puedes omitir gran parte de la tediosa y difícil codificación y centrarte en la lógica de alto nivel de la dinámica del juego.

En este capítulo, configurarás Pygame y crearás una nave que se mueve a derecha e izquierda, y dispara balas en respuesta a la entrada del jugador. En los dos capítulos siguientes, crearás una flota de alienígenas para destruir, y luego seguirás haciendo mejoras, como establecer límites en el número de naves que puedes usar y añadir un marcador.

En este capítulo también aprenderás a gestionar grandes proyectos que abarcan múltiples archivos. Refactorizaremos mucho código y

gestionaremos el contenido de los archivos para mantener nuestro proyecto organizado y el código eficiente.

Crear juegos es una forma ideal de divertirse mientras se aprende un idioma. Es profundamente satisfactorio ver a otros jugar a un juego que tú has escrito, y escribir un juego sencillo te ayudará a entender cómo se escriben los juegos profesionales. Mientras trabajas en este capítulo, introduce y ejecuta el código para entender cómo cada bloque de código contribuye a la jugabilidad general. Experimenta con diferentes valores y configuraciones para obtener una mejor comprensión de cómo refinar las interacciones en tus propios juegos.

NOTA *Alien Invasion abarcará una serie de archivos diferentes, así que cree una nueva carpeta en su sistema llamada* alien_invasion*. Asegúrese de guardar todos los archivos del proyecto en esta carpeta para que las instrucciones de importación funcionen correctamente.*

Planificación del proyecto

Cuando se construye un gran proyecto, es importante preparar un plan antes de empezar a escribir el código. Tu plan te mantendrá centrado y hará más probable que completes el proyecto.

Escribamos una descripción de la jugabilidad general. Aunque esta descripción no cubre todos los detalles de Alien Invasion, proporciona una idea clara de cómo empezar a construir el juego:

> En Alien Invasion, el jugador controla una nave que aparece en l a parte inferior central de la pantalla. El jugador puede mover la nave a derecha e izquierda con las teclas de flecha y disparar balas con la barra espaciadora. Cuando empieza el juego, una flota de alienígenas llena el cielo y se desplaza por la pantalla. El jugador dispara y destruye a los alienígenas. Si el jugador dispara a todos los alienígenas, aparece una nueva flota que se mueve más rápido que la anterior. Si algún alienígena golpea la nave del jugador o llega al fondo de la pantalla, el jugador pierde una nave. Si el jugador pierde tres naves, el juego termina.

Para la primera fase de desarrollo, haremos una nave que pueda moverse a derecha e izquierda. La nave debe poder disparar balas cuando el jugador pulse la barra espaciadora. Después de configurar este comportamiento, podemos centrar nuestra atención en los alienígenas y refinar la jugabilidad.

Instalación de Pygame

Antes de empezar a programar, instala Pygame. Aquí te explicamos cómo hacerlo en Linux, OS X y Microsoft Windows.

Si estás usando Python 3 en Linux o si estás usando OS X, necesitarás usar pip para instalar Pygame. pip es un programa que maneja la descarga e instalación de paquetes Python por ti. Las siguientes secciones te mostrarán cómo instalar paquetes con pip.

Si estás usando Python 2.7 en Linux o si estás usando Windows, no

necesitarás pip para instalar Pygame. En su lugar, pasa a "Instalar Pygame en Linux" en la página 238 o "Instalar Pygame en Windows" en la página 240.

NOTE *Las instrucciones para instalar pip en todos los sistemas se incluyen en las secciones que siguen porque necesitará pip para los proyectos de visualización de datos y aplicaciones web. Estas instrucciones también se incluyen en los recursos en línea en* https://www.nostarch.com/pythoncrashcourse/. *Si tienes problemas con las instrucciones aquí, comprueba si las instrucciones en línea te funcionan.*

Instalación de paquetes Python con pip

Las versiones más recientes de Python vienen con pip instalado, así que primero comprueba si pip ya está en tu sistema. Con Python 3, pip a veces se llama *pip3*.

Comprobación de pip en Linux y OS X

Abra una ventana de terminal e introduzca el siguiente comando:

```
$ pip --version
❶ pip 7.0.3 from /usr/local/lib/python3.5/dist-packages (python 3.5)
$
```

Si sólo tienes una versión de Python instalada en tu sistema y ves una salida similar a ésta, pasa a "Instalar Pygame en Linux" en la página 238 o "Instalar Pygame en OS X" en la página 239. Si recibes un mensaje de error, intenta usar `pip3` en lugar de `pip`. Si ninguna de las dos versiones está instalada en tu sistema, ve a "Instalar pip" en la página 238.

Si tienes más de una versión de Python en tu sistema, verifica que pip está asociado con la versión de Python que estás usando-por ejemplo, `python 3.5` en ❶. Si pip está asociado con la versión correcta de Python, pasa a "Instalar Pygame en Linux" en la página 238 o "Instalar Pygame en OS X" en la página 239. Si pip está asociado con la versión incorrecta de Python, intenta usar `pip3` en lugar de `pip`. Si ninguno de los dos comandos funciona para la versión de Python que estás usando, ve a "Instalar pip" en la página 238.

Comprobación de pip en Windows

Abra una ventana de terminal e introduzca el siguiente comando:

```
$ python -m pip --version
❶ pip 7.0.3 desde C:\Python35\lib\site-packages (python 3.5)
$
```

Si tu sistema sólo tiene una versión de Python instalada y ves una salida similar a ésta, pasa a "Instalar Pygame en Windows" en la página 240. Si recibe un mensaje de error, intente usar `pip3` en lugar de `pip`. Si ninguna de las dos versiones está instalada en tu sistema, pasa a "Instalar pip" en la página 238.

Si tu sistema tiene más de una versión de Python instalada, verifica que pip está asociado con la versión de Python que estás usando-por ejemplo, `python 3.5` en ❶. Si pip está asociado con la versión correcta de Python, pasa a

"Instalar Pygame en Windows" en la página 240. Si pip está asociado con

la versión incorrecta de Python, intente usar `pip3` en lugar de `pip`. Si ninguno de los dos com- mandos funciona para la versión de Python que estás utilizando, pasa a "Instalar pip" a continuación.

Instalación de pip

Para instalar pip, vaya a https://bootstrap.pypa.io/get-pip.py. Guarde el archivo si se le pide que lo haga. Si el código de *get-pip.py* aparece en su navegador, copie y pegue el programa en su editor de texto y guarde el archivo como *get-pip.py*. Una vez que *get-pip.py* esté guardado en tu ordenador, tendrás que ejecutar
con privilegios administrativos porque pip instalará nuevos
p a q u e t e s en su sistema.

NOTE *Si no encuentra get-pip*.py*, vaya a https://pip.pypa.io/, haga clic en* ***Instalación*** *en el panel izquierdo y, a continuación, en "Instalar pip", siga el enlace a* get-pip.py.

Instalación de pip en Linux y OS X

Utilice el siguiente comando para ejecutar *get-pip.py* con privilegios administrativos:

```
$ sudo python get-pip.py
```

NOTE *Si utiliza el comando* ***python3*** *para iniciar una sesión de terminal, debe utilizar aquí* ***sudo python3 get-pip.py****.*

Después de ejecutar el programa, utilice el comando **pip --version** (o **pip3 --version**) para asegurarse de que pip se ha instalado correctamente.

Instalación de pip en Windows

Utilice el siguiente comando para ejecutar *get-pip.py*:

```
$ python get-pip.py
```

Si utiliza un comando diferente para ejecutar Python en una terminal, asegúrese d e utilizar ese comando para ejecutar get-pip.*py*. Por ejemplo, su comando podría ser `python3 get-pip.py` o `C:\Python35\python get-pip.py`.

Después de ejecutar el programa, ejecute el comando **python -m pip --version** para asegurarse de que pip se ha instalado correctamente.

Instalación de Pygame en Linux

Si estás usando Python 2.7, instala Pygame usando el gestor de paquetes. Abre una ventana de terminal y ejecuta el siguiente

comando, que descargará e instalará Pygame en tu sistema:

```
$ sudo apt-get install python-pygame
```

Pruebe su instalación en una sesión de terminal introduciendo lo siguiente:

```
$ python
>>> import pygame
>>>
```

Si no aparece ninguna salida, Python ha importado Pygame y estás listo para pasar a "Iniciar el proyecto de juego" en la página 240.

Si estás ejecutando Python 3, son necesarios dos pasos: instalar las librerías de las que depende Pygame, y descargar e instalar Pygame.

Introduce lo siguiente para instalar las librerías que Pygame necesita. (Si utilizas un comando como `python3.5` en tu sistema, sustituye `python3-dev` por `python3.5-dev`).

```
$ sudo apt-get install python3-dev mercurial
$ sudo apt-get install libsdl-image1.2-dev libsdl2-dev libsdl-ttf2.0-dev
```

Esto instalará las librerías necesarias para ejecutar Alien Invasion con éxito. Si quieres habilitar alguna funcionalidad más avanzada en Pygame, como la posibilidad de añadir sonidos, también puedes añadir las siguientes librerías:

```
$ sudo apt-get install libsdl-mixer1.2-dev libportmidi-dev
$ sudo apt-get install libswscale-dev libsmpeg-dev libavformat-dev libavcode-dev
$ sudo apt-get install python-numpy
```

Ahora instale Pygame introduciendo lo siguiente (use `pip3` si es apropiado para su sistema):

```
$ pip install --user hg+http://bitbucket.org/pygame/pygame
```

La salida se detendrá un momento después de informarle de qué bibliotecas ha encontrado Pygame. Pulsa ENTER, aunque falten algunas librerías. Deberías ver un mensaje indicando que Pygame se ha instalado correctamente.

Para confirmar la instalación, ejecuta una sesión de terminal Python e intenta importar Pygame introduciendo lo siguiente:

```
$ python3
>>> import pygame
>>>
```

Si esto funciona, pasa a "Iniciar el proyecto de juego" en la página 240.

Instalación de Pygame en OS X

Necesitarás Homebrew para instalar algunos paquetes de los que depende Pygame. Si aún no has instalado Homebrew, consulta el Apéndice A para obtener instrucciones.

Para instalar las librerías de las que depende Pygame, introduce lo

siguiente:

```
$ brew install hg sdl sdl_image sdl_ttf
```

Esto instalará las librerías necesarias para ejecutar Alien Invasion. Deberías ver cómo se desplaza la salida a medida que se instala cada biblioteca.

Si además quieres activar funciones más avanzadas, como incluir sonido en los juegos, puedes instalar dos librerías adicionales:

```
$ brew install sdl_mixer portmidi
```

Utilice el siguiente comando para instalar Pygame (utilice `pip` en lugar de `pip3` si está ejecutando Python 2.7):

```
$ pip3 install --user hg+http://bitbucket.org/pygame/pygame
```

Inicia una sesión de terminal Python e importa Pygame para comprobar si la instalación se ha realizado correctamente (introduce `python` en lugar de `python3` si estás ejecutando Python 2.7):

```
$ python3
>>> import pygame
>>>
```

Si la declaración de `importación` funciona, pase a "Iniciar el proyecto de juego" más abajo.

Instalación de Pygame en Windows

El proyecto Pygame está alojado en un sitio para compartir código llamado Bitbucket. Para instalar Pygame en tu versión de Windows, busca un instalador de Windows en *https://bitbucket.org/pygame/pygame/downloads/* que coincida con la versión de Python que estás ejecutando. Si no encuentras un instalador apropiado en Bitbucket, consulta *http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame*.

Una vez descargado el archivo correspondiente, ejecute el instalador si es un *.exe*.

Si tiene un archivo que termina en *.whl*, cópielo en la dirección de su proyecto. Abra una ventana de comandos, vaya a la carpeta en la que ha copiado el instalador y utilice pip para ejecutarlo:

```
> python -m pip install --user pygame-1.9.2a0-cp35-none-win32.whl
```

Inicio del proyecto de juego

Ahora vamos a empezar a construir nuestro juego creando primero un Pygame win- dow vacío en el que más tarde podremos dibujar nuestros elementos de juego, como la nave y los alienígenas. También haremos que nuestro juego responda a las entradas del usuario, establezca el color de

fondo y cargue una imagen de la nave.

Creación de una ventana Pygame y respuesta a las entradas del usuario

Primero, crearemos una ventana vacía de Pygame. Esta es la estructura básica de un juego escrito en Pygame:

alien_invasión.py

```
importar sys

importar pygame

def ejecutar_juego():
    # Inicializa el juego y crea un objeto pantalla.
❶   pygame.init()
❷   screen = pygame.display.set_mode((1200, 800))
    pygame.display.set_caption("Invasión
    alienígena")

    # Inicia el bucle principal del juego.
❸   mientras que True:

        # Vigila los eventos de teclado y ratón.
❹       para evento en pygame.event.get():
❺           if event.type == pygame.QUIT:
                sys.exit()

        # Haz visible la pantalla dibujada más recientemente.
❻       pygame.display.flip()

run_game()
```

En primer lugar, importamos los módulos `sys` y `pygame`. El módulo `pygame` contiene la funcionalidad necesaria para crear un juego. Usaremos el módulo `sys` para salir del juego cuando el jugador lo abandone.

Alien Invasion se inicia como la función `run_game()`. La línea `pygame.init()` en ❶ inicializa los ajustes de fondo que Pygame necesita para funcionar correctamente. En ❷, llamamos a `pygame.display.set_mode()` para crear una ventana de visualización llamada `screen`, 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. Pasando estas dimensiones a `pygame.display.set_mode()`, creamos una ventana de juego de 1200 píxeles de ancho por 800 píxeles de alto. (Puedes ajustar estos valores dependiendo del tamaño de tu pantalla).

El objeto `pantalla` se llama superficie. Una *superficie* en Pygame es una parte de la pantalla donde se muestra un elemento del juego. Cada elemento del juego, como los aliens o la nave, es una superficie. La superficie devuelta por `display.set_mode()` representa toda la ventana del juego. Cuando activamos el bucle de animación del juego, esta superficie se redibuja automáticamente en cada pasada por el bucle.

El juego se controla mediante un bucle `while` ❸ que contiene un bucle de eventos y código que gestiona las actualizaciones de pantalla. Un *evento* es una acción que el usuario realiza mientras juega, como pulsar una tecla o

mover el ratón. Para hacer que nuestro programa responda a los eventos, escribiremos un bucle de *eventos* para escuchar un evento y realizar una tarea apropiada dependiendo del tipo de evento que se haya producido. El bucle `for` de ❹ es un bucle de eventos.

Para acceder a los eventos detectados por Pygame, usaremos el método `pygame.event.get()`. Cualquier evento de teclado o ratón hará que se ejecute el bucle `for`. Dentro del bucle, escribiremos una serie de sentencias `if` para detectar y responder a eventos específicos. Por ejemplo, cuando el jugador hace clic en el botón de cierre de la ventana del juego, se detecta un evento `pygame.QUIT` y llamamos a `sys.exit()` para salir del juego ❺.

La llamada a `pygame.display.flip()` en ❻ le dice a Pygame que haga visible la pantalla dibujada más recientemente. En este caso dibuja una pantalla vacía cada vez que pasa por el bucle `while` para borrar la pantalla antigua y que sólo sea visible la nueva. Cuando movemos los elementos del juego, `pygame.display.flip()` actualizará continuamente la pantalla para mostrar las nuevas posiciones de los elementos y ocultar las antiguas, creando la ilusión de movimiento suave.

La última línea de esta estructura básica de juego llama a `run_game()`, que inicializa el juego e inicia el bucle principal.

Ejecuta este código ahora, y deberías ver una ventana vacía de Pygame.

Configuración del color de fondo

Pygame crea una pantalla negra por defecto, pero eso es aburrido. Pongamos un color de fondo diferente:

alien_invasión.py

```
--snip--
def ejecutar_juego():
    --snip--
    pygame.display.set_caption("Invasión alienígena")

 # Establece el color de fondo.
❶    bg_color = (230, 230, 230)

    # Inicia el bucle principal del
    juego. while True:

        # Vigila los eventos de teclado y ratón.
        --snip--

        # Redibuja la pantalla durante cada pasada por el bucle.
❷       screen.fill(bg_color)

        # Hacer visible la pantalla dibujada más recientemente.
        pygame.display.flip()

run_game()
```

En primer lugar, creamos un color de fondo y lo almacenamos en `bg_color` ❶. Este color debe especificarse una sola vez, por lo que definimos su valor antes de entrar en el bucle `while` principal.

Los colores en Pygame se especifican como colores 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. Puedes mezclar valores RGB
para crear 16 millones de colores. El valor de color (230, 230, 230) mezcla cantidades iguales de rojo, azul y verde, lo que produce un color de fondo gris claro.

En ❷, rellenamos la pantalla con el color de fondo utilizando la función `screen.fill()` que sólo recibe un argumento: un color.

Creación de una clase de configuración

Cada vez que introducimos una nueva funcionalidad en nuestro juego, normalmente introducimos también algunos ajustes nuevos. En lugar de añadir ajustes en todo el código, vamos a escribir un módulo llamado `ajustes` que contiene una clase llamada `Ajustes` para almacenar todos los ajustes en un solo lugar. Este enfoque nos permite para pasar un objeto de configuración en lugar de muchas configuraciones individuales.
Además, simplifica nuestras llamadas a funciones y facilita la modificación de la apariencia del juego a medida que crece nuestro proyecto. Para modificar el juego, simplemente cambiaremos algunos valores en *settings.py* en lugar de buscar diferentes configuraciones en todos nuestros archivos.

Esta es la clase inicial de `Configuración`:

settings.py

```
clase Ajustes():
    """Una clase para almacenar todos los ajustes de Alien Invasion."""

    def init (self):
        """Inicializar la configuración del
        juego.""" # Ajustes de pantalla
        self.ancho_pantalla = 1200
        self.altura_pantalla = 800
        self.bg_color = (230, 230, 230)
```

Para crear una instancia de `Settings` y utilizarla para acceder a nuestra configuración, modifique *alien_invasion.py* como sigue:

alien_invasión.py

```
--snip--
importar pygame

from settings import

Configuración def

ejecutar_juego():
    # Inicializa pygame, la configuración y el objeto pantalla.
    pygame.init()
❶   ai_settings = Ajustes()
❷   screen = pygame.display.set_mode(
        (ai_settings.screen_width, ai_settings.screen_height))
    pygame.display.set_caption("Invasión alienígena")

    # Inicia el bucle principal del
    juego. while True:
        --snip--
        # Redibuja la pantalla durante cada pasada por el bucle.
```

```
❸         screen.fill(ai_settings.bg_color)

          # Hacer visible la pantalla dibujada más recientemente.
          pygame.display.flip()

run_game()
```

Importamos `Settings` en el archivo principal del programa, y luego creamos una instancia de `Settings` y la almacenamos en `ai_settings` después de hacer la llamada a `pygame.init()` ❶. Cuando creamos una pantalla ❷, usamos los atributos `screen_width` y `screen_height` de `ai_settings`, y luego usamos `ai_settings` para acceder al color de fondo al rellenar la pantalla en ❸ también.

Añadir la imagen del buque

Ahora vamos a añadir la nave a nuestro juego. Para dibujar la nave del jugador en pantalla, cargaremos una imagen y luego usaremos el método `blit()` de Pygame para dibujar la imagen.

Cuando elijas material gráfico para tus juegos, asegúrate de prestar atención a las licencias. La forma más segura y barata de empezar es utilizar gráficos con licencia libre que puedas modificar desde un sitio web como http://pixabay.com/.

Puedes usar casi cualquier tipo de archivo de imagen en tu juego, pero es más fácil si usas un archivo de mapa de bits (*.bmp*) porque Pygame carga mapas de bits por defecto. Aunque puedes configurar Pygame para usar otros tipos de archivo, algunos tipos de archivo dependen de ciertas librerías de imágenes que deben estar instaladas en tu ordenador. (La mayoría de las imágenes que encontrarás están en formato *.jpg*, *.png* o *.gif*, pero puedes convertirlas a mapas de bits usando herramientas como Photoshop, GIMP y Paint).

Presta especial atención al color de fondo de la imagen elegida. Intenta encontrar un archivo con fondo transparente que puedas sustituir por cualquier color de fondo utilizando un editor de imágenes. Tus juegos tendrán mejor aspecto
si el color de fondo de la imagen coincide con el color de fondo del juego. Alternativamente, puedes hacer coincidir el fondo de tu juego con el fondo de la imagen.

Para Alien Invasion, puedes usar el archivo *ship.bmp* (Figura 12-1), que está disponible en los recursos del libro a través de *https://www.nostarch.com/ pythoncrashcourse/*. El color de fondo del archivo coincide con la configuración que estamos utilizando en este proyecto. Crea una carpeta llamada *images* dentro de la carpeta principal de tu proyecto (*alien_invasion*). Guarda el archivo *ship.bmp* en la carpeta *images*.

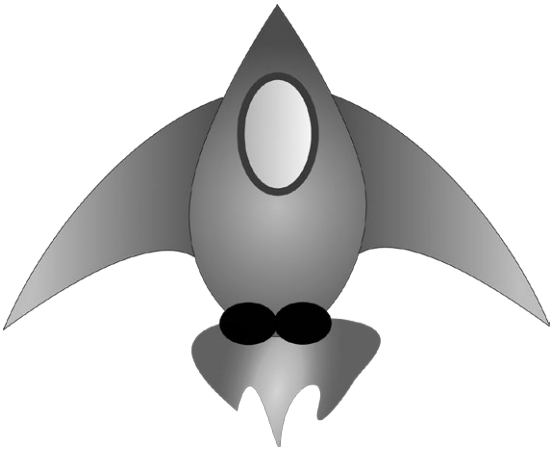


Figura 12-1: La nave de Alien Invasion

Creación de la clase de buque

Después de elegir una imagen para la nave, necesitamos mostrarla en pantalla. Para usar nuestra nave, escribiremos un módulo llamado `nave`, que contiene la clase `Nave`. Esta clase gestionará la mayor parte del comportamiento de la nave del jugador.

barco.py

```
importar pygame

clase Nave():

    def init (self, screen):
        """Inicializar la nave y fijar su posición inicial."""
        self.screen = screen

    # Cargar la imagen de la nave y obtener su rect.
❶       self.image = pygame.image.load('images/ship.bmp')
❷       self.rect = self.image.get_rect()
❸       self.screen_rect = screen.get_rect()

        # Comienza cada nuevo barco en la parte inferior central de la
        pantalla.
❹       self.rect.centerx = self.screen_rect.centerx
        self.rect.bottom = self.screen_rect.bottom

❺   def blitme(self):
        """Dibuja el barco en su posición actual."""
        self.screen.blit(self.image, self.rect)
```

Primero, importamos el módulo `pygame`. El método `init ()` de `Ship` toma dos parámetros: la `auto` referencia y la `pantalla` donde dibujaremos la nave.

Para cargar la imagen, llamamos a `pygame.image.load()` ❶. Esta función devuelve una superficie que representa la nave, que almacenamos en self.`image`.

Una vez cargada la imagen, usamos `get_rect()` para acceder al atributo `rect` de la superficie ❷. Una de las razones por las que Pygame es tan eficiente es que te permite tratar los elementos del juego como rectángulos (*rects*), incluso si no tienen exactamente la forma de rectángulos. Tratar un elemento como un rectángulo es eficiente porque los rectángulos son formas geométricas simples. Este enfoque suele funcionar lo suficientemente bien como para que nadie que juegue se dé cuenta de que no estamos trabajando con la forma exacta de cada elemento del juego.

Al trabajar con un objeto `rect`, puede utilizar las coordenadas x e y de los bordes superior, inferior, izquierdo y derecho del rectángulo, así como el centro. Puede establecer cualquiera de estos valores para determinar la posición actual del rectángulo.

Cuando estés centrando un elemento del juego, trabaja con los atributos `center`, `centerx`, o `centery` de un rect. Cuando estés trabajando en un borde de la pantalla, trabaja con los atributos `top`, `bottom`, `left` o `right`. Para ajustar la colocación horizontal o vertical del rectángulo, sólo tienes que utilizar los

atributos `x` e `y`, que son las coordenadas x e y de su esquina superior izquierda. Estos atributos te evitan tener que hacer cálculos que antes tenían que hacer manualmente los desarrolladores de juegos, y verás que los usarás a menudo.

NOTE *En Pygame, el origen (0, 0) está en la esquina superior izquierda de la pantalla, y las coordenadas aumentan a medida que vas hacia abajo y hacia la derecha. En una pantalla de 1200 por 800, el origen está en la esquina superior izquierda, y la esquina inferior derecha tiene las coordenadas (1200, 800).*

Posicionaremos la nave en el centro inferior de la pantalla. Para ello, primero almacena el rectángulo de la pantalla en self.screen_rect ❸, y luego haz que el valor de self.rect.centerx (la coordenada x del centro del barco) coincida con el atributo centerx del rectángulo de la pantalla ❹. Haz que el valor de self.rect.bottom (la coordenada coordenada y de la parte inferior de la nave) igual al valor del atributo bottom del rectángulo de la pantalla. Pygame utilizará estos atributos rect para posicionar la imagen del barco de forma que esté centrada horizontalmente y alineada con el fondo de la pantalla.

En ❺ definimos el método blitme(), que dibujará la imagen en la pantalla en la posición especificada por self.rect.

Dibujar el barco en la pantalla

Ahora actualicemos *alien_invasion.py* para que cree una nave y llame al comando de la nave método blitme():

alien_invasión.py

```
--snip--
from settings import Configuración
from barco import Barco

def ejecutar_juego():
    --snip--
    pygame.display.set_caption("Invasión alienígena")

    # Haz una nave.
❶   nave = Nave(pantalla)

    # Inicia el bucle principal del
    juego. while True:
        --snip--
        # Redibuja la pantalla durante cada pasada por el bucle.
        screen.fill(ai_settings.bg_color)
❷       barco.blitme()

        # Hacer visible la pantalla dibujada más recientemente.
        pygame.display.flip()

run_game()
```

Importamos Ship y luego hacemos una instancia de Ship (llamada ship) después de que la pantalla haya sido creada. Debe venir antes del bucle while principal ❶ para que no hagamos una nueva instancia de la nave en cada pasada por el bucle. Dibujamos el barco en la pantalla llamando a

`ship.blitme()` después de rellenar el fondo, para que el barco aparezca sobre el fondo ❷.

Cuando ejecutes ahora *alien_invasion.py*, deberías ver una pantalla de juego vacía con nuestro cohete espacial en la parte inferior central, como se muestra en la Figura 12-2.

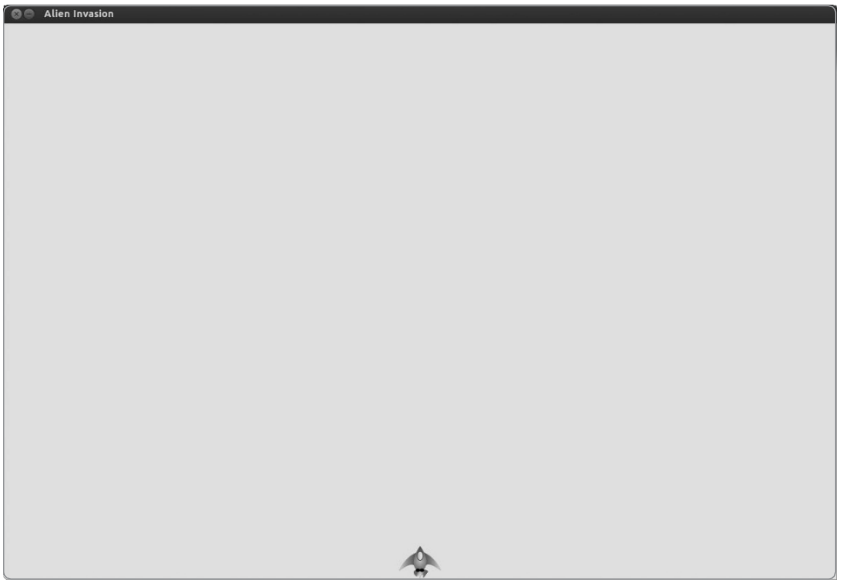


Figura 12-2: Invasión alienígena con la nave en la parte inferior central de la pantalla

Refactorización: el módulo game_functions

En proyectos de mayor envergadura, a menudo se refactoriza el código ya escrito antes de añadir más código. *La refactorización* simplifica la estructura del código que ya has escrito, haciendo que sea más fácil construir sobre él. En esta sección vamos a crear un nuevo módulo llamado `game_functions`, que almacenará una serie de funciones que hacen que Alien Invasion funcione. El módulo game_functions pre evita que *alien_invasion.py* se haga demasiado largo y hará que la lógica de *alien_invasion.py sea* más fácil de seguir.

Función check_events()

Empezaremos moviendo el código que gestiona los eventos a una función separada llamada `check_events()`. Esto simplificará `run_game()` y aislará el bucle de gestión de eventos. Aislar el bucle de eventos te permite gestionar los eventos de forma separada de otros aspectos del juego, como la actualización de la pantalla.

Coloca `check_events()` en un módulo separado llamado `game_functions`:

juego funciones.py

```
importar sys

importar pygame

def comprobar_eventos():
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        if evento.type == pygame.QUIT:
            sys.exit()
```

Este módulo importa `sys` y `pygame`, que se utilizan en el bucle de comprobación de eventos. La función no necesita parámetros en este punto, y el cuerpo se copia del bucle de eventos en *alien_invasion.py*.

Ahora vamos a modificar *alien_invasion.py* para que importe el módulo `game_functions`, y sustituiremos el bucle de eventos por una llamada a `check_events()`:

alien_invasión.py

```
importar pygame

from settings import
Configuración from ship
import Nave
import game_functions as gf

def ejecutar_juego():
    --snip--
    # Inicia el bucle principal del
    juego. while True:
        gf.check_events()

        # Redibuja la pantalla durante cada pasada por el bucle.
        --snip--
```

Ya no necesitamos importar `sys` directamente en el archivo principal del programa, porque ahora sólo se utiliza en el módulo `game_functions`. Damos al módulo `game_functions` importado el alias `gf` para simplificar.

Función update_screen()

Movamos el código para actualizar la pantalla a una función separada llamada `update_screen()` en *game_functions.py* para simplificar aún más `run_game()`:

juego_funciones.py

```
--snip--

def comprobar_eventos():
    --snip--

def update_screen(ai_settings, screen, ship):
    """Actualizar imágenes en la pantalla y pasar a la nueva
    pantalla.""" # Redibuja la pantalla durante cada pasada por
    el bucle. screen.fill(ai_settings.bg_color)
    barco.blitme()

    # Hacer visible la pantalla dibujada más recientemente.
    pygame.display.flip()
```

La nueva función `update_screen()` toma tres parámetros: `ai_settings`, `screen`, y `ship`. Ahora necesitamos actualizar el bucle `while` de *alien_invasion.py* con una llamada a `update_screen()`:

alien_invasion.py

```
--snip--
    # Inicia el bucle principal del juego.
```

```
    mientras sea verdad:
        gf.check_events()
        gf.update_screen(ai_settings, screen, ship)

run_game()
```

Estas dos funciones simplifican el bucle `while` y facilitarán el desarrollo posterior. En lugar de trabajar dentro de `run_game()`, podemos hacer la mayor parte de nuestro trabajo en el módulo `game_functions`.

Como queríamos empezar a trabajar con código en un único archivo, no introdujimos el módulo `game_functions` de inmediato. Este enfoque te da una idea de un proceso de desarrollo realista: empiezas escribiendo tu código de la forma más simple posible, y lo refactorizas a medida que tu proyecto se vuelve más complejo.

Ahora que nuestro código está reestructurado para que sea más fácil de añadir, ¡podemos trabajar en los aspectos dinámicos del juego!

TRY IT yOURSELF

12-1. Cielo azul: Crea una ventana Pygame con un fondo azul.

12-2. Personaje del juego: Encuentre una imagen bitmap de un personaje de juego que le guste o convierta una imagen a bitmap. Haga 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

Vamos a dar al jugador la posibilidad de mover la nave a derecha e izquierda. Para ello, escribiremos código que responda cuando el jugador pulse la tecla de flecha derecha o izquierda. Primero nos centraremos en el movimiento hacia la derecha, y luego aplicaremos los mismos principios para controlar el movimiento hacia la izquierda. Mientras haces esto, aprenderás cómo controlar el movimiento de las imágenes en la pantalla.

Respuesta a una pulsación de tecla

Cada vez que el jugador pulsa una tecla, esa pulsación se registra en Pygame como un evento. Cada evento es recogido por el método `pygame.event.get()`, por lo que necesitamos especificar en nuestra función `check_events()` qué tipo de eventos comprobar. Cada pulsación de tecla se registra como un evento `KEYDOWN`.

Cuando se detecta un evento `KEYDOWN`, necesitamos comprobar si la tecla que se ha pulsado es una de las que activa un determinado evento. Por ejemplo, si la tecla

se pulsa la tecla de flecha derecha, aumentamos el valor `rect.centerx` del barco para moverlo hacia la derecha:

juego_funciones.py

```
def comprobar_eventos(barco):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        if evento.type == pygame.QUIT:
            sys.exit()

    ❶ elif event.type == pygame.KEYDOWN:
    ❷     if evento.key == pygame.K_RIGHT:
              # Mueve la nave a la derecha.
    ❸         ship.rect.centerx += 1
```

Damos a la función `check_events()` un parámetro de nave, porque la nave necesita moverse a la derecha cuando se pulsa la tecla de flecha derecha. 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 es la de la flecha derecha (`pygame.K_RIGHT`) leyendo el atributo `event.key` ❷. Si la tecla de flecha derecha fue presionada, movemos la nave a la derecha incrementando el valor de `ship.rect.centerx` en 1 ❸.

Tenemos que actualizar la llamada a `check_events()` en *alien_invasion.py* para que pase `la nave` como argumento:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    while True:
        gf.check_events(nave)
        gf.update_screen(ai_settings, screen, ship)
```

Si ejecutas *alien_invasion.py* ahora, deberías ver que la nave se mueve a la derecha un píxel cada vez que pulsas la tecla de flecha derecha. Eso es un comienzo, pero no es una forma eficiente de controlar la nave. Mejoremos este control permitiendo el movimiento continuo.

Permitir el movimiento continuo

Cuando el jugador mantiene pulsada la tecla de flecha derecha, queremos que la nave continúe moviéndose hacia la derecha hasta que el jugador suelte la tecla. Haremos que nuestro juego detecte un evento `pygame.KEYUP` para saber cuándo se suelta la tecla de flecha derecha; luego usaremos los eventos `KEYDOWN` y `KEYUP` junto con una bandera llamada `moving_right` para implementar el movimiento continuo.

Cuando la nave está inmóvil, la bandera `moving_right` será `False`. Cuando se pulse la tecla de flecha derecha, pondremos la bandera a `True`, y cuando se suelte, volveremos a poner la bandera a `False`.

La clase `Ship` controla todos los atributos de la nave, así que le daremos un atributo 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 cada vez que queramos actualizar la posición del barco.

Estos son los cambios en la clase `Barco`:

barco.py

```
clase Nave():

    def init (self, screen):
        --snip--
        # Comienza cada nuevo barco en el centro inferior de la
        pantalla. self.rect.centerx = self.screen_rect.centerx
        self.rect.bottom = self.screen_rect.bottom

    # Bandera de movimiento
❶       self.moving_right = False

❷   def update(self):
        """Actualiza la posición del barco en función de la bandera
        de movimiento.""" if self.moving_right:
            self.rect.centerx += 1

    def blitme(self):
        --snip--
```

Añadimos un atributo `self.moving_right` en el método `init ()` y lo ponemos a `False` inicialmente ❶. Luego añadimos `update()`, que mueve la nave a la derecha si la bandera es `True` ❷.

Ahora modifica `check_events()` para que `moving_right` sea `True` cuando la tecla de flecha derecha es presionada y `False` cuando la tecla es liberada:

juego_funciones.py

```
def comprobar_eventos(barco):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
❶               ship.moving_right = True

❷       elif evento.type == pygame.KEYUP:
            if evento.key == pygame.K_RIGHT:
                nave.mover_derecha = False
```

En ❶, modificamos cómo responde el juego cuando el jugador pulsa la tecla de flecha derecha: en lugar de cambiar la posición de la nave directamente, simplemente establecemos `moving_right` a `True`. En ❷, añadimos un nuevo bloque `elif`, que responde a los eventos `KEYUP`. Cuando el jugador suelta la tecla de flecha derecha (`K_RIGHT`), ponemos `moving_right` a `False`.

Por último, modificamos el bucle `while` de *alien_invasion.py* para que llame a la nave `update()` en cada pasada por el bucle:

*alien_*

```
    # Inicia el bucle principal del juego.
```

invasión.py

```
while True:
    gf.check_events(nave)
    nave.update()
    gf.update_screen(ai_settings, screen, ship)
```

La posición de la nave se actualizará después de comprobar los eventos de teclado y antes de actualizar la pantalla. Esto permite que la posición de la nave se actualice en respuesta a la entrada del jugador y asegura que la posición actualizada se utilice al dibujar la nave en la pantalla.

Cuando ejecutes *alien_invasion.py* y mantengas pulsada la tecla de flecha derecha, la nave debería moverse continuamente hacia la derecha hasta que sueltes la tecla.

Moverse tanto a la izquierda como a la derecha

Ahora que la nave puede moverse continuamente hacia la derecha, añadir movimiento hacia la izquierda es fácil. Vamos a modificar de nuevo la clase `Ship` y la función `check_events()`. Aquí están los cambios relevantes a `init ()` y `update()` en `Ship`:

nave.py

```
    def init (self, screen):
        --snip--
        # Banderas de movimiento
        self.moving_right = False
        self.moving_left = False

    def update(self):
        """Actualiza la posición del barco basándote en las
        banderas de movimiento.""" if self.moving_right:
            self.rect.centerx += 1
        if self.moving_left:
            self.rect.centerx -= 1
```

En `init ()`, añadimos una bandera `self.moving_left`. En `update()`, usamos dos bloques `if` separados en lugar de un `elif` en `update()` para permitir que el valor `rect.centerx` de la nave aumente y luego disminuya si ambas teclas de flecha se mantienen pulsadas. El resultado es que la nave se queda quieta. Si usáramos `elif` para el movimiento a la izquierda, la tecla de flecha derecha siempre tendría prioridad. Hacerlo de esta manera hace que los movimientos sean más precisos cuando se cambia de izquierda a derecha, cuando el jugador puede mantener momentáneamente pulsadas ambas teclas.

Tenemos que hacer dos ajustes en `check_events()`:

juego_funciones.py

```
def comprobar_eventos(barco):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                ship.moving_right = True
            elif event.key == pygame.K_LEFT:
                ship.moving_left = True

        elif evento.type == pygame.KEYUP:
            if evento.key == pygame.K_RIGHT:
                nave.mover_derecha = False
```

```
            elif event.key == pygame.K_LEFT:
                nave.mover_izquierda = False
```

Si ocurre un evento KEYDOWN para la tecla K_LEFT, establecemos moving_left a True. Si ocurre un evento KEYUP para la tecla K_LEFT, establecemos moving_left a False. Podemos usar bloques elif aquí porque cada evento está conectado a una sola tecla. Si el jugador presiona ambas teclas a la vez, dos eventos separados serán detectados.

Si ejecutas *alien_invasion.py* ahora, deberías poder mover la nave continuamente a derecha e izquierda. Si mantienes pulsadas ambas teclas, la nave debería dejar de moverse.

A continuación, perfeccionaremos el movimiento de la nave. Vamos a ajustar la velocidad de la nave y a limitar la distancia a la que puede moverse para que no desaparezca por los lados de la pantalla.

Ajustar la velocidad del barco

Actualmente, la nave se mueve un píxel por ciclo a través del bucle while, pero podemos tener un control más preciso de la velocidad de la nave añadiendo un atributo ship_speed_factor a la clase Settings. Utilizaremos este atributo para determinar cuánto se moverá la nave en cada pasada por el bucle. Aquí está el nuevo atributo en *settings.py*:

ajustes.py

```
clase Ajustes():
    """Una clase para almacenar todos los ajustes de Alien Invasion."""

    def init (self):
        --snip--

        # Configuración de la nave
        self.ship_speed_factor = 1.5
```

Fijamos el valor inicial de ship_speed_factor en 1,5. Cuando queramos mover la nave, ajustaremos su posición en 1,5 píxeles en lugar de 1 píxel.

Estamos utilizando valores decimales para el ajuste de la velocidad para tener un control más preciso de la velocidad de la nave cuando aumentemos el ritmo del juego más adelante. Sin embargo, los atributos rect como centerx sólo almacenan valores enteros, por lo que tenemos que hacer algunas modificaciones en Ship:

barco.py

```
clase Nave():

❶   def init (self, ai_settings, screen):
        """Inicializar la nave y fijar su posición inicial."""
        self.screen = pantalla
❷       self.ai_settings = ai_settings
        --snip--

        # Comienza cada nuevo barco en la parte inferior central de la
        pantalla.
        --snip--

        # Almacena un valor decimal para el centro de la nave.
❸       self.center = float(self.rect.centerx)
```

```
        # Banderas de movimiento
        self.moving_right = False
        self.moving_left = False
```

```
    def update(self):
        """Actualiza la posición del barco basándote en las
        banderas de movimiento.""" # Actualiza el valor central
        del barco, no el recto.
        if self.mover_derecha:
❹           self.center += self.ai_settings.ship_speed_factor
        if self.moving_left:
            self.center -= self.ai_settings.ship_speed_factor

        # Actualiza el objeto rect desde self.center.
❺       self.rect.centerx = self.center

    def blitme(self):
        --snip--
```

En ❶, añadimos `ai_settings` a la lista de parámetros de `init_()`, para que la nave tenga acceso a su ajuste de velocidad. Luego convertimos el parámetro `ai_settings` en un atributo, para que podamos usarlo en `update()` ❷. Ahora que estamos ajustando la posición de la nave por fracciones de píxel, necesitamos almacenar la posición en una variable que pueda almacenar un valor decimal. Puede utilizar un valor decimal para establecer el atributo de un `rect`, pero el `rect` almacenará sólo el inte- de ese valor. Para almacenar la posición del `barco` con precisión, definimos un nuevo atributo self.`center`, que puede contener valores decimales ❸. Utilizamos la función `float()` para convertir el valor de self.`rect.centerx` en un decimal y almacenamos este valor en self.`center`.

Ahora, cuando cambiamos la posición de la nave en `update()`, el valor de self.`center` se ajusta por la cantidad almacenada en `ai_settings.ship_speed_ factor` ❹. Una vez actualizado self.center, utilizamos el nuevo valor para actualizar `self.rect.centerx`, que controla la posición de la nave ❺. Sólo la parte entera de self.`center` se almacenará en self.rect.`centerx`, pero eso está bien para mostrar la nave.

Necesitamos pasar `ai_settings` como argumento cuando creamos una instancia de `Nave` en *alien_invasion.py*:

alien_invasión.py

```
--snip--
def ejecutar_juego():
    --snip--
    # Haz una nave.
    nave = Nave(ai_configuración, pantalla)
    --snip--
```

Ahora cualquier valor de `ship_speed_factor` mayor que uno hará que la nave se mueva más rápido. Esto será útil para hacer que la nave responda lo suficientemente rápido como para derribar alienígenas, y nos permitirá cambiar el tempo del juego a medida que el jugador progresa en la partida.

Limitar el alcance de la nave

En este punto, la nave desaparecerá de cualquiera de los bordes de la pantalla si mantienes pulsada una tecla de flecha el tiempo suficiente. Vamos a corregir esto para que la nave deje de moverse cuando llegue al borde de la pantalla. Lo haremos modificando el método `update()` de `Ship`:

ship.py

```
    def update(self):
        """Actualiza la posición del barco basándote en las
        banderas de movimiento.""" # Actualiza el valor central
        del barco, no el recto.
❶       if self.moving_right and self.rect.right < self.screen_rect.right:
            self.center += self.ai_settings.ship_speed_factor
❷       if self.moving_left and self.rect.left > 0:
            self.center -= self.ai_settings.ship_speed_factor

        # Actualiza el objeto rect desde
        self.center. self.rect.centerx =
        self.center
```

Este código comprueba la posición del barco antes de cambiar el valor de `self.center`. El código `self.rect.right` devuelve el valor de la coordenada x del borde derecho del `rectángulo` del barco. Si este valor es menor que el valor devuelto por `self.screen_rect.right`, la nave no ha alcanzado el borde derecho de la pantalla ❶. Lo mismo ocurre con el borde izquierdo: si el valor del lado izquierdo del `rect` es mayor que cero, la nave no ha alcanzado el borde izquierdo de la pantalla ❷. Esto asegura que la nave está dentro de estos límites antes de ajustar el valor de `self.center`.

Si ejecutas *alien_invasion.py* ahora, la nave debería dejar de moverse en cualquiera de los bordes de la pantalla.

Refactorización de check_events()

La función `check_events()` aumentará en longitud a medida que continuemos desarrollando el juego, así que vamos a dividir `check_events()` en dos funciones más: una que maneje los eventos `KEYDOWN` y otra que maneje los eventos `KEYUP`:

juego_functions.py

```
def check_keydown_events(evento, nave):
    """Responder a pulsaciones de teclas."""
    if evento.key == pygame.K_RIGHT:
        nave.mover_derecha = True
    elif event.key == pygame.K_LEFT:
        nave.mover_izquierda = True

def check_keyup_events(evento,
    nave): """Responde a los
    lanzamientos de llave.""" if
    evento.llave == pygame.K_RIGHT:
        ship.moving_right = False
    elif event.key == pygame.K_LEFT:
        ship.moving_left = False
```

```
def comprobar_eventos(barco):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        if evento.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            check_keydown_events(event, ship)
        elif event.type == pygame.KEYUP:
            check_keyup_events(event, ship)
```

Creamos dos nuevas funciones: `check_keydown_events()` y `check_keyup_events()`. Cada una necesita un `parámetro de evento` y un parámetro de `barco`. Los cuerpos de estas dos funciones se copian de check_events(), y hemos sustituido el código antiguo por llamadas a las nuevas funciones. La función `check_events()` es más simple ahora con esta estructura de código más limpia, lo que hará más fácil desarrollar más respuestas a la entrada del jugador.

Resumen rápido

En la siguiente sección, `vamos` a añadir la capacidad de disparar balas, lo que implica un nuevo archivo llamado *bullet.py* y algunas modificaciones a algunos de los archivos que nos
ya tenemos. En este momento, tenemos cuatro archivos que contienen una serie de clases, funciones y métodos. Para tener claro cómo está organizado el proyecto, vamos a revisar cada uno de estos archivos antes de añadir más funcionalidad.

alien_invasion.py

El archivo principal, *alien_invasion.py*, crea una serie de objetos importantes utilizados a lo largo del juego: los ajustes se almacenan en `ai_settings`, la superficie de juego principal se almacena en `screen`, y una instancia de la `nave` se crea en este archivo también. También se almacena en *alien_invasion.py* el bucle principal del `juego`, que es un bucle `while` que llama a `check_events()`, `ship.update()`, y `update_screen()`.

alien_invasion.py es el único archivo que necesitas ejecutar cuando quieras jugar a Alien Invasion. Los otros archivos-settings*.py*, *game_functions.py*, *ship.py*- contienen código que se importa, directa o indirectamente, en este archivo.

settings.py

El archivo *settings.py* contiene la clase `Settings`. Esta clase sólo tiene un `__init__()`, que inicializa los atributos que controlan la apariencia del juego y la velocidad de la nave.

game_functions.py

El archivo *game_functions.py* contiene una serie de funciones que realizan la mayor parte del trabajo en el juego. La función `check_events()` detecta

eventos rel- evantes, como pulsaciones y liberaciones de teclas, y procesa cada uno de estos tipos de eventos a través de las funciones ayudantes `check_keydown_events()` y

check_keyup_events(). Por ahora, estas funciones gestionan el movimiento de la nave. El módulo game_functions también contiene update_screen(), que redibuja la pantalla en cada pasada por el bucle principal.

ship.py

El fichero *ship.py* contiene la clase Ship. Ship 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 real de la nave se almacena en *ship.bmp*, que está en la carpeta *images*.

TRY IT yOURSELF

12-3. Cohete: Haga un juego que comience con un cohete en el centro de la pantalla. Permita al jugador mover el cohete hacia arriba, abajo, izquierda o derecha usando las cuatro teclas de flecha. Asegúrese de que el cohete nunca se mueve más allá de cualquier borde de la pantalla.

12-4. Llaves: Crea un archivo Pygame que cree una pantalla vacía. En el bucle de eventos, imprime el atributo event.key cada vez que se detecte un evento pygame.KEYDOWN. Ejecuta el programa y pulsa varias teclas para ver cómo responde Pygame.

Disparar balas

Ahora vamos a añadir la posibilidad de disparar balas. Escribiremos un código que dispare una bala (un pequeño rectángulo) cuando el jugador pulse la barra espaciadora. Las balas subirán por la pantalla hasta desaparecer.

Añadir la configuración de las viñetas

En primer lugar, actualice *settings.py* para incluir los valores que necesitaremos para una nueva Viñeta
al final del método init ():

settings.py

```
    def init (self):
        --snip--
        # Configuración de la bala
        self.bullet_speed_factor = 1
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
```

Estos ajustes crean balas de color gris oscuro con una anchura de 3 píxeles y una altura de 15 píxeles. Las balas se desplazarán ligeramente más despacio que la nave.

Creación de la clase Bullet

Ahora crea un archivo *bullet.py* para almacenar nuestra clase `Bullet`. Aquí está la primera parte de
bullet.py:

bala.py

```
importar pygame
from pygame.sprite import Sprite

clase Bullet(Sprite):
    """Una clase para gestionar las balas disparadas desde el barco"""

    def init (self, ai_settings, screen, ship):
        """Crear un objeto bala en la posición actual de la nave."""
        super(Bullet, self). init ()
        self.screen = pantalla

        # Crea un bullet rect en (0, 0) y luego fija la posición correcta.
❶       self.rect = pygame.Rect(0, 0, ai_settings.bullet_width,
            ai_settings.bullet_height)
❷       self.rect.centerx = ship.rect.centerx
❸       self.rect.top = ship.rect.top

        # Almacena la posición de la bala como un valor decimal.
❹       self.y = float(self.rect.y)

❺       self.color = ai_settings.bullet_color
        self.speed_factor = ai_settings.bullet_speed_factor
```

La clase `Bullet` hereda de `Sprite`, que importamos del módulo `pygame.sprite`. Cuando usas sprites, puedes agrupar elementos relacionados en tu juego y actuar sobre todos los elementos agrupados a la vez. Para crear una instancia bullet, `init ()` necesita las instancias `ai_settings`, `screen`, y `ship`, y llamamos a `super()` para heredar correctamente de `Sprite`.

NOTE *La llamada* `super(Bullet, self). init ()` *utiliza la sintaxis de Python 2.7. Esto también funciona en Python 3, o también puedes escribir esta llamada más simplemente como* `super(). init ()`.

En ❶, creamos el atributo `rect` de la bala. La bala no está basada en una imagen, así que tenemos que construir un rect desde cero usando la clase `pygame.Rect()`. Esta clase requiere las coordenadas x e y de la esquina superior izquierda del `rect`, y el ancho y alto del `rect`. Inicializamos el `rect` en (0, 0), pero lo moveremos a la posición correcta en las dos líneas siguientes, porque la posición de la bala depende de la posición del barco. Obtenemos el ancho y alto de la bala de los valores almacenados en `ai_settings`.

En ❷, fijamos el `centerx` de la bala para que sea el mismo que el `rect.centerx` del barco.

La bala debe salir de la parte superior de la nave, así que fijamos la parte superior del `rectángulo de` la bala para que coincida con la parte superior del

`rectángulo` de la nave, haciendo que parezca que la bala se dispara desde la nave ❸.

Almacenamos un valor decimal para la coordenada y de la bala para poder hacer ajustes finos a la velocidad de la bala ❹. En ❺, almacenamos los ajustes de color y velocidad de la bala en `self.color` y `self.speed_factor`.

Aquí está la segunda parte de *bullet.py*, `update()` y `draw_bullet()`:

bullet.py

```
    def update(self):
        """Mueve la bala hacia arriba en la pantalla."""
        # Actualiza la posición decimal de la bala.
❶       self.y -=
        self.factor_velocidad #
        Actualiza la posición del
        recto.
❷       self.rect.y = self.y

    def dibujar_bullet(self):
        """Dibuja la viñeta en la pantalla."""
❸       pygame.draw.rect(pantalla.auto, color.auto, rect.auto)
```

El método `update()` gestiona la posición de la bala. Cuando se dispara una bala, se mueve hacia arriba en la pantalla, lo que corresponde a una disminución de y; así que para actualizar la posición, restamos la cantidad almacenada en `self.speed_factor` de `self.y` ❶. A continuación, utilizamos el valor de self.y para establecer el valor de `self.rect.y` ❷. El atributo `speed_factor` nos permite aumentar la velocidad de las balas a medida que el juego progresa o según sea necesario para refinar el comportamiento del juego. Una vez disparada, el valor de la coordenada x de una bala nunca cambia, por lo que sólo viajará verticalmente en línea recta.

Cuando queramos dibujar una bala, llamaremos a `draw_bullet()`. La función `draw.rect()` rellena la parte de la pantalla definida por el rectángulo de la bala con el color almacenado en `self.color` ❸.

Almacenamiento de balas en un grupo

Ahora que tenemos una clase `Bullet` y los ajustes necesarios definidos, podemos escribir código para disparar una bala cada vez que el jugador pulse la barra espaciadora. En primer lugar, vamos a crear un grupo en *alien_invasion.py* para almacenar todas las balas vivas para que podamos gestionar las balas que ya han sido disparadas. Este grupo será una instancia de la clase `pygame.sprite.Group`, que se comporta como una lista con algunos funcionalidad extra que es útil cuando se construyen juegos. Usaremos este grupo para dibujar balas en la pantalla en cada pasada por el bucle principal y para actualizar la posición de cada bala:

alien_invasión.py

```
importar pygame
from pygame.sprite import Grupo
--snip--

def ejecutar_juego():
     --snip--
    # Haz una nave.
    nave = Nave(ai_configuración, pantalla)
    # Crea un grupo para almacenar las balas.
```

```
❶ balas = Grupo()

# Inicia el bucle principal del
juego. while True:
    gf.check_events(ai_settings, screen, ship, bullets)
    ship.update()
```

```
❷          balas.actualizar()
           gf.update_screen(ai_settings, screen, ship, bullets)

run_game()
```

Importamos `Group` de `pygame.sprite`. En ❶, hacemos una instancia de `Group` y la llamamos `bullets`. Este grupo se crea fuera del bucle `while` para no crear un nuevo grupo de balas cada vez que el bucle hace un ciclo.

NOTE *Si haces un grupo como este dentro del bucle, estarás creando miles de grupos de balas y tu juego probablemente se ralentizará hasta detenerse. Si tu juego se congela, mira cuidadosamente lo que está pasando en tu bucle* while principal.

Pasamos `balas` a `check_events()` y `update_screen()`. Tendremos que trabajar con `viñetas` en check_events() cuando se pulse la barra espaciadora, y tendremos que actualizar las viñetas que se están dibujando en la pantalla en `update_screen()`.

Cuando llamas a `update()` en un grupo ❷, el grupo automáticamente llama a `update()` para cada sprite en el grupo. La línea `bullets.update()` llama a `bullet.update()` para cada bala que colocamos en el grupo `bullets`.

Disparar balas

En *game_functions*.py, necesitamos modificar `check_keydown_events()` para disparar una bala cuando se pulsa la barra espaciadora. No necesitamos modificar `check_keyup_events()` porque no pasa nada cuando se suelta la tecla. También necesitamos modificar `update_screen()` para asegurarnos de que cada bala se redibuja en la pantalla antes de llamar a `flip()`. Estos son los cambios relevantes en *game_functions.py*:

juego_funciones.py

```
--snip--
from bala import Bala

❶ def check_keydown_events(event, ai_settings, screen, ship, bullets):
      --snip--
❷     elif evento.key == pygame.K_SPACE:
          # Crea una nueva bala y añádela al grupo de balas.
          nueva_bala = Bala(ai_configuración, pantalla, nave)
          balas.añadir(nueva_bala)
        --snip--

❸ def check_events(ai_settings, screen, ship,
      bullets): """Responde a eventos de teclado y
      ratón.""" for event in pygame.event.get():
          --snip--
          elif event.type == pygame.KEYDOWN:
              check_keydown_events(event, ai_settings, screen, ship, bullets)
          --snip--

❹ def update_screen(ai_settings, screen, ship, bullets):
      --snip--
```

```
        # Redibuja todas las balas detrás de la nave y los aliens.
❺       for bala in balas.sprites():
            bala.dibujar_bala()
```

```
barco.blitme()
--snip--
```

El grupo `balas` se pasa a `check_keydown_events()` ❶. Cuando el jugador pulsa la barra espaciadora, creamos una nueva bala (una instancia de `Bullet` a la que llamamos `new_bullet`) y la añadimos al grupo `bullets` ❷ mediante el método `add()`; el código `bullets.add(new_bullet)` almacena la nueva bala en el grupo `bullets`.

Necesitamos añadir `viñetas` como parámetro en la definición de `check_ events()` ❸, y necesitamos pasar `viñetas` como argumento en la llamada a `check_keydown_events()` también.

Damos el parámetro `bullets` a `update_screen()` en ❹, que dibuja las balas en la pantalla. El método `bullets.sprites()` devuelve una lista de todos los sprites en el grupo `balas`. Para dibujar todas las balas disparadas en la pantalla, hacemos un bucle a través de los sprites en `balas` y llamamos a `draw_bullet()` en cada una ❺.

Si ejecutas *alien_invasion.py* ahora, deberías poder mover la nave a derecha e izquierda, y disparar tantas balas como quieras. Las balas suben por la pantalla y desaparecen cuando llegan arriba, como se muestra en la Figura 12-3. Puedes alterar el tamaño, el color y la velocidad de las balas en settings. Puedes alterar el tamaño, color y velocidad de las balas en *settings.py*.

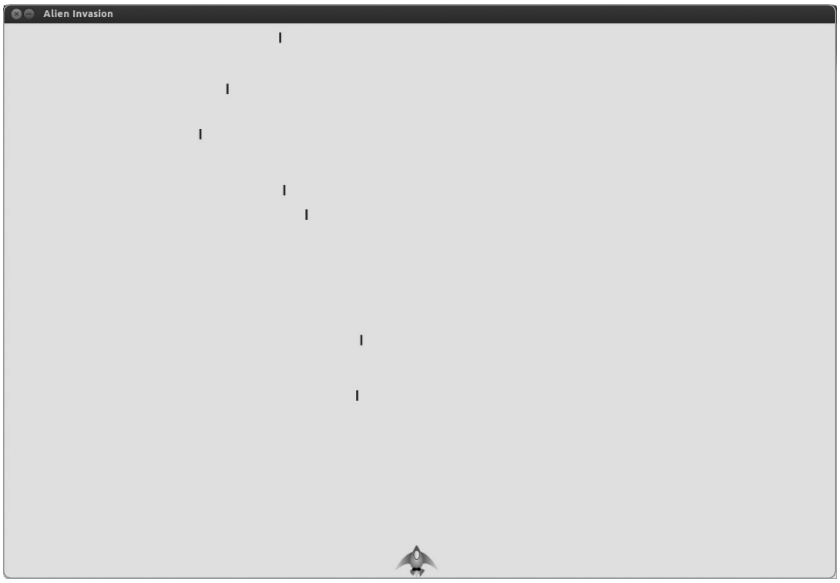


Figura 12-3: La nave después de disparar una serie de balas

Borrar balas antiguas

Por el momento, las balas desaparecen cuando llegan a la parte superior, pero

sólo porque Pygame no puede dibujarlas por encima de la parte superior de la pantalla. En realidad, las balas siguen existiendo, pero sus valores de coordenada y son cada vez más negativos. Esto es un problema, porque siguen consumiendo memoria y capacidad de proceso.

Tenemos que deshacernos de estas viejas balas, o el juego se ralentizará por hacer tanto trabajo innecesario. Para ello, tenemos que detectar cuando el valor `inferior del rectángulo` de una bala tiene un valor de 0, lo que indica que la bala ha pasado de la parte superior de la pantalla:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        balas.actualizar()

        # Deshazte de las balas que han desaparecido.
❶       para bullet en bullets.copy():
❷           si bullet.rect.bottom <= 0:
❸               bullets.remove(bullet)
❹       print(len(balas))

        gf.update_screen(ai_settings, screen, ship, bullets)
```

No deberías eliminar elementos de una lista o grupo dentro de un bucle `for`, así que tenemos que hacer un bucle sobre una copia del grupo.

Utilizamos el método `copy()` para configurar el bucle `for` ❶, lo que nos permite modificar `las viñetas` dentro del bucle. Comprobamos cada viñeta para ver si ha desaparecido de la parte superior de la pantalla en ❷. Si lo ha hecho, la eliminamos de las `viñetas` ❸. En ❹ insertamos una sentencia `print` para mostrar cuántas balas existen actualmente en el juego y verificar que se están eliminando.

Si este código funciona correctamente, podemos observar la salida de la terminal mientras se disparan las balas y ver que el número de balas disminuye a cero después de que cada conjunto de balas ha despejado la parte superior de la pantalla. Después de ejecutar el juego y verificar que las balas se eliminan correctamente, elimine la instrucción `print`. Si la deja, el juego se ralentizará significativamente porque lleva más tiempo escribir la salida en el terminal que dibujar los gráficos en la ventana del juego.

Limitar el número de balas

Muchos juegos de disparos limitan el número de balas que un jugador puede tener en la pantalla a la vez para animar a los jugadores a disparar con precisión. Haremos lo mismo en Alien Invasion.

Primero, almacena el número de viñetas permitidas en *settings.py*:

settings.py

```
        # Configuración de
        viñetas
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.balas_permitidas = 3
```

Esto limita al jugador a tres balas a la vez. Usaremos este ajuste en *game_functions.py* para comprobar cuántas balas existen antes de crear una nueva bala en `check_keydown_events()`:

juego_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    --snip--
    elif evento.key == pygame.K_SPACE:
        # Crea una nueva viñeta y añádela al grupo de viñetas.
        if len(viñetas) < ai_settings.bullets_allowed:
            nueva_bala = Bala(ai_configuración, pantalla, nave)
            balas.añadir(nueva_bala)
```

Cuando se pulsa la barra espaciadora, comprobamos la longitud de las `balas`. Si `len(balas)` es menor que tres, creamos una nueva bala. Pero si ya hay tres balas activas, no pasa nada cuando se pulsa la barra espaciadora. Si ejecutas el juego ahora, deberías poder disparar balas sólo en grupos de tres.

Creación de la función update_bullets()

Queremos mantener nuestro archivo de programa principal *alien_invasion.py* lo más simple posible, así que ahora que hemos escrito y comprobado el código de gestión de balas podemos moverlo al módulo `game_functions`. Crearemos una nueva función llamada `update_bullets()` y la añadiremos al final de *game_functions.py*:

juego_functions.py

```
def actualizar_balas(balas):
    """Actualizar posición de balas y deshacerse de balas
    viejas.""" # Actualiza la posición de las balas.
    balas.actualizar()

    # Deshacerse de las viñetas que han
    desaparecido. for viñeta in
    viñetas.copiar():
        if bullet.rect.bottom <= 0:
            bullets.remove(bullet)
```

El código de `update_bullets()` está cortado y pegado de *alien_invasion.py*; el único parámetro que necesita es el grupo de `balas`.

El bucle `while` de *alien_invasion.py* vuelve a parecer sencillo:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
❶       gf.check_events(ai_settings, pantalla, nave, balas)
❷       barco.actualizar()
❸       gf.update_bullets(bullets)
❹       gf.update_screen(ai_settings, screen, ship, bullets)
```

Hemos hecho que nuestro bucle principal contenga sólo el código mínimo para que podamos leer rápidamente los nombres de las funciones y entender lo que está sucediendo en el juego. El bucle principal comprueba la entrada del jugador en ❶, y luego actualiza la posición de

la nave en ❷ y las balas que se han disparado en ❸. A continuación, utilizamos las posiciones actualizadas para dibujar una nueva pantalla en ❹.

Creación de la función fire_bullet()

Vamos a mover el código para disparar una bala a una función separada para que podamos utilizar una sola línea de código para disparar una bala y mantener el bloque `elif` en `check_keydown_events()` simple:

juego_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    """Responder a la pulsación de teclas."""
    --snip--
    elif event.key == pygame.K_SPACE:
        fire_bullet(ai_settings, pantalla, nave, balas)

def disparar_balas(ai_configuracion, pantalla, nave,
    balas): """Dispara una bala si aún no se ha
    alcanzado el límite""".
    # Crea una nueva viñeta y añádela al grupo de viñetas.
    if len(viñetas) < ai_settings.bullets_allowed:
        nueva_bala = Bala(ai_configuración, pantalla, nave)
        balas.añadir(nueva_bala)
```

La función `fire_bullet()` simplemente contiene el código que se utilizó para disparar una bala cuando se pulsa la barra espaciadora, y añadimos una llamada a `fire_bullet()` en `check_keydown_events()` cuando se pulsa la barra espaciadora.

Ejecuta *alien_invasion.py* una vez más, y asegúrate de que todavía puedes disparar balas sin errores.

TRY IT yOURSELF

12-5. Tirador Lateral: Escribe un juego que coloque una nave en el lado izquierdo de la pantalla y permita al jugador mover la nave arriba y abajo. Haz que la nave dispare una bala que viaje a la derecha a través de la pantalla cuando el jugador presiona la barra espaciadora. Asegúrate de que las balas se borran una vez que desaparecen de la pantalla.

Resumen

En este capítulo, aprendiste a hacer un plan para un juego. Aprendiste la estructura básica de un juego escrito en Pygame. Aprendiste a establecer un color de fondo y almacenar configuraciones en una clase separada donde pueden estar disponibles para todas las partes del juego. Usted vio cómo dibujar una imagen en la pantalla y dar al jugador el control sobre el movimiento de los elementos del juego. Aprendiste a crear elementos que se mueven solos, como balas que vuelan por la pantalla, y a eliminar objetos que ya no son necesarios. Aprendiste a refactorizar el código de un proyecto de forma regular para facilitar el desarrollo continuo.

En el capítulo 13, añadiremos alienígenas a Alien Invasion. Al final

del capítulo 13, serás capaz de derribar alienígenas, ¡con suerte antes de que lleguen a tu nave!

13

ALIENS



En este capítulo añadiremos alienígenas a Alien Invasion. En primer lugar, vamos a añadir un extranjero cerca de la parte superior de la pantalla, y luego vamos a gener-
se comió a toda una flota de alienígenas. Haremos que la flota avance hacia los lados y hacia abajo, y nos desharemos de cualquier alienígena alcanzado por una bala. Por último, limitaremos el número de naves de un jugador y terminaremos la partida cuando el jugador se quede sin naves.

A medida que trabajes en este capítulo, aprenderás más sobre Pygame y sobre la gestión de un proyecto más grande. También aprenderás a detectar colisiones entre objetos del juego, como balas y alienígenas. Detectar colisiones ayuda a
definirás las interacciones entre los elementos de tus juegos: puedes encerrar a un personaje dentro de las paredes de un laberinto o pasar una pelota entre dos personajes. También seguiremos trabajando a partir de un plan que

revisaremos de vez en cuando para mantener el enfoque de nuestras sesiones de escritura de código.

Antes de empezar a escribir nuevo código para añadir una flota de alienígenas a la pantalla, echemos un vistazo al proyecto y actualicemos nuestro plan.

Revisión del proyecto

Cuando comienzas una nueva fase de desarrollo en un proyecto más grande, siempre es una buena idea revisar tu plan y aclarar lo que quieres lograr con el código que estás a punto de escribir. En este capítulo lo haremos:

- Examinar nuestro código y determinar si necesitamos refactorizarlo antes de implementar nuevas funciones.
- Añade un único alienígena en la esquina superior izquierda de la pantalla con el espacio adecuado a su alrededor.
- Usa el espacio alrededor del primer alienígena y el tamaño total de la pantalla para determinar cuántos alienígenas caben en la pantalla. Escribiremos un bucle para crear alienígenas que llenen la parte superior de la pantalla.
- Haz que la flota se mueva hacia los lados y hacia abajo hasta que toda la flota sea derribada, un alien alcance la nave o un alien llegue al suelo. Si toda la flota es derribada, crearemos una nueva flota. Si un alien alcanza la nave o el suelo, destruiremos la nave y crearemos una nueva flota.
- Limitar el número de naves que el jugador puede utilizar y finalizar la partida cuando el jugador haya agotado su asignación de naves.

Iremos perfeccionando este plan a medida que implementemos funciones, pero para empezar es suficiente.

También deberías revisar el código cuando estés a punto de empezar a trabajar en una nueva serie de características de un proyecto. Como cada nueva fase suele hacer más complejo el proyecto, lo mejor es limpiar el código desordenado o ineficiente.

Aunque no tenemos mucho que limpiar ahora mismo porque hemos estado refactorizando sobre la marcha, es molesto usar el ratón para cerrar el juego cada vez que lo ejecutamos para probar una nueva característica. Vamos a añadir rápidamente un atajo de teclado para terminar el juego cuando el usuario pulse Q:

juego_functions.py

```
def check_keydown_events(event, ai_settings, screen, ship, bullets):
    --snip--
    elif event.key == pygame.K_q:
        sys.exit()
```

En `check_keydown_events()` añadimos un nuevo bloque que termina el juego cuando se pulsa Q. Este es un cambio bastante seguro porque la tecla Q está lejos de las teclas de flecha y de la barra espaciadora, por lo que es poco probable que un jugador pulse accidentalmente Q y salga del juego. Ahora, al hacer pruebas, se puede pulsar Q para cerrar el juego en lugar de usar el ratón para cerrar la ventana.

Creación del primer alienígena

Colocar un alienígena en la pantalla es como colocar una nave en la

pantalla. El comportamiento de cada alien es controlado por una clase llamada `Alien`, que estructuraremos como la clase `Ship`. Seguiremos usando imágenes de mapa de bits por simplicidad. Puedes encontrar tu propia imagen para un alienígena o utilizar la que

que se muestra en la Figura 13-1, disponible en los recursos del libro a través de https://www.nostarch.com/pythoncrashcourse/. Esta imagen tiene un fondo gris, que coincide con el color de fondo de la pantalla. Asegúrese de guardar el archivo de imagen que elija en la carpeta de *imágenes*.

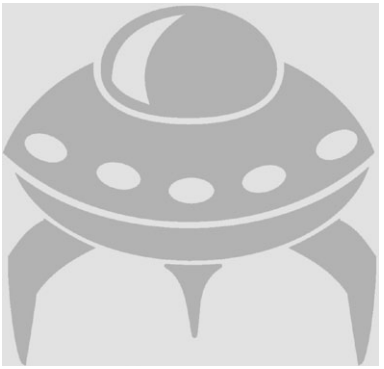


Figura 13-1: El alienígena que utilizaremos para construir la flota

Creación de la clase Alien

Ahora escribiremos la clase `Alien`:

alien.py

```
importar pygame
from pygame.sprite import Sprite

clase Alien(Sprite):
    """Una clase para representar a un único alienígena de la flota""".

    def init (self, ai_settings, screen):
        """Inicializar el alien y fijar su posición inicial."""
        super(Alien, self). init ()
        self.screen = pantalla
        self.ai_settings = ai_settings

        # Cargar la imagen alienígena y establecer su
        atributo        rect.        self.image        =
        pygame.image.load('images/alien.bmp')  self.rect  =
        self.image.get_rect()

        # Comienza cada nuevo alien cerca de la parte superior izquierda de la
        pantalla.
(1)     self.rect.x = self.rect.width
        self.rect.y = self.rect.height

        # Almacena la posición exacta del
        alien. self.x = float(self.rect.x)

    def blitme(self):
        """Dibuja al alienígena en su ubicación actual."""
        self.screen.blit(self.image, self.rect)
```

La mayor parte de esta clase es como la clase `Nave` excepto por la colocación del alienígena. Inicialmente colocamos cada alien cerca de la esquina superior izquierda de la pantalla, añadiendo un espacio a su izquierda igual a la anchura del alien y un espacio por encima igual a su altura ❶.

Creación de una instancia del Alien

Ahora creamos una instancia de `Alien` en *alien_invasion.py*:

alien_invasión.py

```
--snip--
from nave import Nave
from alien import Alien
import game_functions as gf

def ejecutar_juego():
    --snip--
    # Haz un extraterrestre.
    alien = Alien(ai_settings, screen)

    # Inicia el bucle principal del
    juego. while True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.actualizar_balas(balas)
        gf.update_screen(ai_settings, screen, ship, alien, bullets)

run_game()
```

Aquí estamos importando la nueva clase `Alien` y creando una instancia de `Alien` justo antes de entrar en el bucle `while` principal. Como todavía no estamos cambiando la posición del alien, no estamos añadiendo nada nuevo dentro del bucle; sin embargo, modificamos la llamada a `update_screen()` para pasarle la instancia del `alien`.

Cómo hacer que el alien aparezca en pantalla

Para que el alien aparezca en pantalla, llamamos a su método `blitme()` en `actualizar_pantalla()`:

juego_functions.py

```
def update_screen(ai_settings, screen, ship, alien, bullets):
    --snip--

    # Redibuja todas las balas detrás de la nave
    y los aliens. para bala en balas:
        bullet.draw_bullet()
    ship.blitme()
    alien.blitme()

    # Hacer visible la pantalla dibujada más recientemente.
    pygame.display.flip()
```

Dibujamos el alienígena en pantalla después de que la nave y las balas hayan sido dibujadas, por lo que los alienígenas serán la capa superior de la pantalla. La Figura 13-2 muestra el primer alienígena en la pantalla.

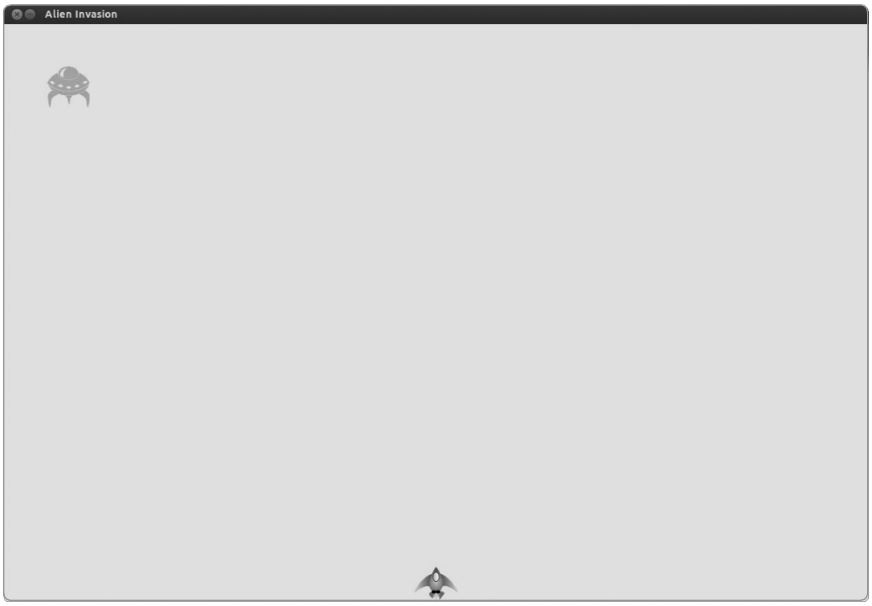


Figura 13-2: Aparece el primer alienígena.

Ahora que el primer alienígena aparece correctamente, escribiremos el código para dibujar una flota entera.

Construir la flota alienígena

Para dibujar una flota, tenemos que calcular cuántos alienígenas caben en la pantalla y cuántas filas de alienígenas caben en la pantalla. Primero calcularemos el espacio horizontal entre alienígenas y crearemos una fila; después determinaremos el espacio vertical y crearemos una flota entera.

Determinar cuántos alienígenas caben en una fila

Para saber cuántos alienígenas caben en una fila, veamos cuánto espacio horizontal tenemos. El ancho de la pantalla se almacena en `ai_settings.screen_width`, pero necesitamos un margen vacío a cada lado de la pantalla. Haremos que este margen tenga el ancho de un alienígena. Como tenemos dos márgenes, el espacio disponible para los alienígenas es el ancho de la pantalla menos el ancho de dos alienígenas:

```
espacio_disponible_x = ai_settings.ancho_pantalla - (2 * ancho_alien)
```

También tenemos que establecer el espacio entre los extraterrestres; lo haremos de un ancho de extraterrestre. El espacio necesario para mostrar un alienígena es el doble de su anchura: una anchura para el alienígena y una anchura para el espacio vacío a su derecha. Para saber el número de alienígenas que caben en la pantalla, dividimos el espacio disponible entre dos veces la anchura de un alienígena:

```
número_alienígenas_x = espacio_disponible_x / (2 * anchura_alienígena)
```

Incluiremos estos cálculos cuando creemos la flota.

NOTE *Un aspecto estupendo de los cálculos en programación es que no tienes que estar seguro de que tu fórmula es correcta cuando la escribes por primera vez. Puedes probarla y ver si funciona. En el peor de los casos, tendrás una pantalla atestada de alienígenas o con muy pocos alienígenas.*
Puedes revisar tu cálculo en función de lo que veas en la pantalla.

Creación de filas de alienígenas

Para crear una fila, primero crea un grupo vacío llamado `aliens` en *alien_invasion.py* para contener todos nuestros aliens, y luego llama a una función en *game_functions.py* para crear una flota:

alien_invasión.py

```
importar pygame
from pygame.sprite import Grupo
from settings import
Configuración from ship import
Nave
import game_functions as gf

def ejecutar_juego():
     --snip--
    # Crea una nave, un grupo de balas y un grupo de
    alienígenas. nave = Nave(ai_settings, pantalla)
      balas = Grupo()
❶ extranjeros = Grupo()

    # Crea la flota de alienígenas.
❷   gf.create_fleet(ai_settings, screen, aliens)

    # Inicia el bucle principal del
    juego. while True:
        --snip-
❸       gf.update_screen(ai_settings, screen, ship, aliens, bullets)

run_game()
```

Como ya no vamos a crear aliens directamente en *alien_invasion.py*, no necesitamos importar la clase `Alien` en este archivo.

Crea un grupo vacío para contener todos los alienígenas del juego ❶. A continuación, llama a la nueva función `create_fleet()` ❷, que escribiremos

en breve, y pasa
it los `ai_settings`, el objeto `pantalla`, y el grupo vacío `aliens`. A continuación, modifique la llamada a `update_screen()` para darle acceso al grupo de aliens ❸.

También tenemos que modificar `update_screen()`:

juego_funciones.py

```
def update_screen(ai_settings, screen, ship, aliens, bullets):
    --snip--
    ship.blitme()
    aliens.draw(screen)

    # Hacer visible la pantalla dibujada más recientemente.
    pygame.display.flip()
```

Cuando llamas a `draw()` sobre un grupo, Pygame dibuja automáticamente cada elemento del grupo en la posición definida por su atributo `rect`. En este caso, `aliens.draw(screen)` dibuja cada alienígena del grupo en la pantalla.

Creación de la flota

Ahora podemos crear la flota. Aquí está la nueva función `create_fleet()`, que colocamos al final de *game_functions.py*. También tenemos que importar la clase `Alien`, así que asegúrate de añadir una declaración de `importación en la` parte superior del archivo:

juego_funciones.py

```
--snip--
from bala import Bala
from alien import Alien
--snip--

def crear_flota(ai_configuracion, pantalla,
    aliens): """Crea una flota completa de
    alienígenas""".
    # Crea un alien y encuentra el número de alien en una
    fila. # El espacio entre cada alien es igual a un alien
    de ancho.
❶   alien = Alien(ai_settings, screen)
❷   alien_width = alien.rect.width
❸   espacio_disponible_x = ai_settings.ancho_pantalla - 2 * ancho_extraterrestre
❹   número_alienígenas_x = int(espacio_disponible_x / (2 * anchura_alienígena))

    # Crea la primera fila de alienígenas.
❺   para numero_alien en rango(numero_alien_x):
        # Crea un alien y colócalo en la fila.
❻       alien = Alien(ai_settings, screen)
        alien.x = alien_ancho + 2 * alien_ancho * alien_número
        alien.rect.x = alien.x
        aliens.add(alien)
```

Ya hemos pensado la mayor parte de este código. Necesitamos saber el ancho y alto del alien para poder colocar aliens, así que creamos un alien en ❶ antes de realizar los cálculos. Este alien no será parte de la flota, así que no lo agregues al grupo de `aliens`. En ❷ obtenemos el ancho del alien de su atributo rect y almacenamos este valor en `alien_width` para que no tengamos que seguir trabajando a través del atributo `rect`. En ❸ calculamos el espacio horizontal disponible para los aliens y el número de aliens que caben en ese espacio.

El único cambio aquí de nuestras fórmulas originales es que estamos usando `int()` para asegurar que terminamos con un número entero de alienígenas ❹ porque no queremos crear alienígenas parciales, y la función `range()` necesita un

entero. La función `int()` elimina la parte decimal de un número, redondeando hacia abajo. (Esto es útil porque preferimos tener un poco de espacio extra en cada fila que una fila demasiado llena).

A continuación, establece un bucle que cuente desde 0 hasta el número de alienígenas que necesitamos hacer ❺. En el cuerpo principal del bucle, crea un nuevo alienígena y luego establece su valor de coordenada x para colocarlo en la fila ❻. Cada alien es empujado a la derecha un alien de ancho desde el margen izquierdo. A continuación, multiplicamos la anchura por 2 para tener en cuenta el espacio que ocupa cada alienígena, incluido el espacio vacío a su derecha, y multiplicamos esta cantidad por la posición del alienígena en la fila. A continuación, añadimos cada nuevo alienígena al grupo de `alienígenas`.

Cuando ejecute Alien Invasion, debería ver aparecer la primera fila de alienígenas, como en la Figura 13-3.

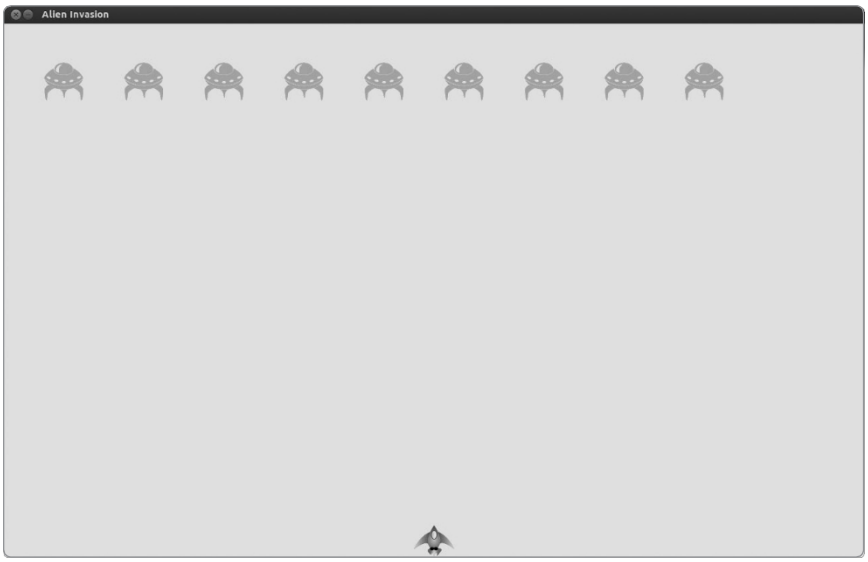


Figura 13-3: La primera fila de extranjeros

La primera fila está desplazada hacia la izquierda, lo cual es bueno para el juego, porque queremos que la flota se mueva hacia la derecha hasta que toque el borde de la pantalla, luego baje un poco, luego se mueva hacia la izquierda, y así sucesivamente. Como en el clásico juego *Space Invaders*, este movimiento es más interesante que hacer que la flota caiga directamente hacia abajo. Continuaremos este movimiento hasta que todos los alienígenas sean derribados o hasta que un alienígena golpee la nave o el fondo de la pantalla.

NOTE *Dependiendo del ancho de pantalla que haya elegido, la alineación de la primera fila de alienígenas puede tener un aspecto ligeramente diferente en su sistema.*

Refactorización de create_fleet()

Si hubiéramos terminado de crear una flota, probablemente dejaríamos `create_fleet()` como está, pero tenemos más trabajo que hacer, así que vamos a limpiar un poco la función. Aquí está `create_fleet()` con dos nuevas funciones: `get_number_aliens_x()` y `create_alien()`:

game_funciones.py

```
❶ def get_number_aliens_x(ai_settings, alien_width):
    """Determinar el número de alienígenas que caben en una
    fila."""
    available_space_x = ai_settings.screen_width - 2 *
    alien_width
    number_aliens_x = int(available_space_x / (2 *
    alien_width))
    return number_aliens_x

def crear_alien(ai_configuracion, pantalla, aliens,
    numero_alien):
    """Crea un alienígena y colócalo en la
    fila""".
    alien = Alien(ai_settings, screen)
❷   alien_width = alien.rect.width
    alien.x = alien_ancho + 2 * alien_ancho * alien_número
    alien.rect.x = alien.x
    aliens.add(alien)

def crear_flota(ai_configuracion, pantalla,
    aliens):
    """Crea una flota completa de
    alienígenas""".
    # Crea un alien y encuentra el número de alien en una
    fila.
    alien = Alien(ai_settings, screen)
❸   número_alienígenas_x = get_número_alienígenas_x(ai_configuración,
alien.rect.anchura)

    # Crea la primera fila de alienígenas.
    para número_alienígena en rango(número_alienígenas_x):
❹       create_alien(ai_settings, screen, aliens, alien_number)
```

El cuerpo de `get_number_aliens_x()` es exactamente igual que en `create_fleet()` ❶. El cuerpo de `create_alien()` tampoco cambia respecto a `create_fleet()` excepto que usamos el alien que se acaba de crear para obtener el ancho del alien ❷. En ❸ reemplazamos el código para determinar el espaciado horizontal con una llamada a `get_ number_aliens_x()`, y eliminamos la línea que hace referencia a `alien_width`, porque ahora se maneja dentro de `create_alien()`. En ❹ llamamos a `create_alien()`. Esta refactorización hará más fácil añadir nuevas filas y crear una flota entera.

Añadir filas

Para terminar la flota, determina el número de filas que caben en la pantalla y luego repite el bucle (para crear los alienígenas en una fila) ese número de veces. Para determinar el número de filas, encontramos la vertical disponible
espacio restando la altura del alienígena de la parte superior, la altura de la nave de la parte inferior y dos alturas del alienígena de la parte inferior de la pantalla:

```
espacio_disponible_y = ai_settings.altura_pantalla - 3 * altura_alien -
altura_nave
```

El resultado creará algo de espacio vacío sobre la nave, de modo que el jugador tenga algo de tiempo para empezar a disparar a los alienígenas al principio de cada nivel.

Cada fila necesita un espacio vacío debajo, que haremos igual a la altura de un alienígena. Para calcular el número de filas, dividimos el espacio disponible por dos veces la altura de un alienígena. (De nuevo, si estos cálculos están mal, lo veremos enseguida y ajustaremos hasta que tengamos un espaciado razonable).

```
número_filas = altura_disponible_y / (2 * altura_alien)
```

Ahora que sabemos cuántas filas caben en una flota, podemos repetir el código para crear una fila:

game_functions.py

```
❶ def get_number_rows(ai_settings, ship_height, alien_height):
      """Determinar el número de filas de alienígenas que caben en la pantalla""".
❷     espacio_disponible_y = (ai_settings.screen_height -
                              (3  *  altura_extraterrestre)  -
      altura_nave) número_filas = int(espacio_disponible_y / (2
      * altura_extraterrestre)) return número_filas

  def create_alien(ai_settings, screen, aliens, alien_number, row_number):
      --snip--
      alien.x = alien_ancho + 2 * alien_ancho * alien_número
      alien.rect.x = alien.x
❸     alien.rect.y = alien.rect.height + 2 * alien.rect.height * row_number
      aliens.add(alien)

  def crear_flota(ai_configuracion, pantalla, nave, aliens):
      --snip--
      número_alienígenas_x = get_número_alienígenas_x(ai_configuración,
      alien.rect.anchura) número_filas = get_número_filas(ai_configuración,
      nave.rect.altura,
          alien.rect.height)

     # Crea la flota de alienígenas.
❹    para número_filas en rango(número_filas):
          para número_alien en rango(número_aliens_x):
              create_alien(ai_settings, pantalla, aliens, número_alien,
                  número_de_fila)
```

Para calcular el número de filas que nos caben en la pantalla, escribimos nuestros cálculos de `available_space_y` y `number_rows` en la función `get_ number_rows()` ❶, que es similar a `get_number_aliens_x()`. El cálculo se envuelve entre paréntesis para que el resultado pueda dividirse en dos líneas, lo que resulta en líneas de 79 caracteres o menos, como se recomienda ❷. Usamos `int()` porque no queremos crear una fila parcial de aliens.

Para crear varias filas, utilizamos dos bucles anidados: uno exterior y otro interior ❹. El bucle interior crea los extraterrestres en una sola fila. El bucle exterior cuenta desde 0 hasta el número de filas que queramos; Python usará el código para hacer una sola fila y lo repetirá `número_de_filas` veces.

Para anidar los bucles, escriba el nuevo bucle `for` y sangrar el código que desea repetir. (La mayoría de los editores de texto facilitan la indentación y desindentación de bloques de código, pero para obtener ayuda consulte el Apéndice B). Ahora, cuando llamemos a `create_alien()`, incluiremos un argumento para el número de fila, de modo que cada fila pueda colocarse más abajo en la pantalla.

La definición de `create_alien()` necesita un parámetro que contenga el número de fila. Dentro de `create_alien()`, cambiamos el valor de la coordenada y de un alien cuando no está en la primera fila ❸ empezando con la altura de un alien para crear un espacio vacío en la parte superior de la pantalla. Cada fila empieza dos alturas de alienígena por debajo de la última fila, así que multiplicamos la altura del alienígena por dos y luego por el número de fila. El número de la primera fila es 0, por lo que la colocación vertical de la primera fila no cambia. Todas las filas siguientes se colocan más abajo en la pantalla.

La definición de `create_fleet()` también tiene un nuevo parámetro para el objeto `nave`, lo que significa que necesitamos incluir el argumento `nave` en la llamada a `create_fleet()` en *alien_invasion.py*:

alien_invasión.py

```
# Crea la flota de alienígenas.
gf.create_fleet(ai_settings, screen, ship, aliens)
```

Cuando ejecute el juego ahora, debería ver una flota de alienígenas, como en la Figura 13-4.

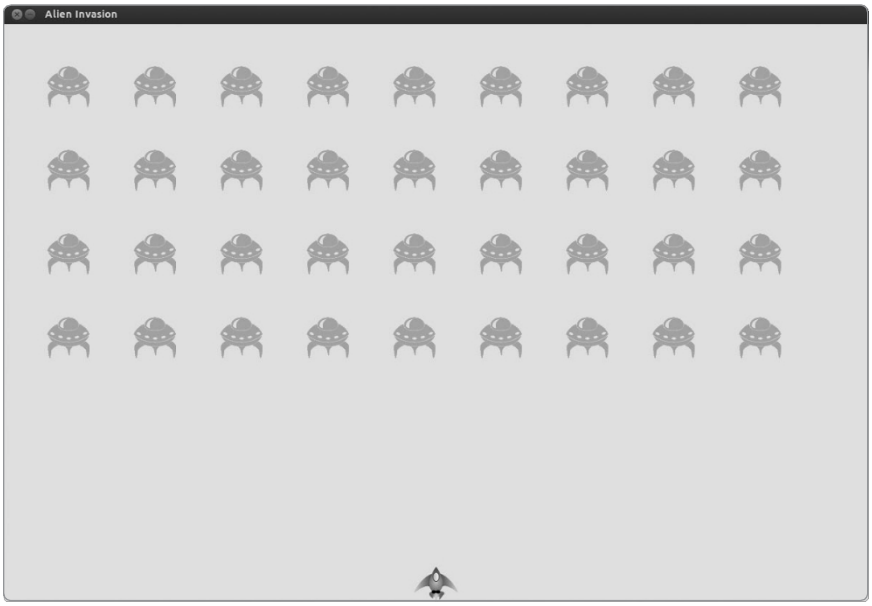


Figura 13-4: Aparece la flota completa.

En la próxima sección, haremos que la flota se mueva.

TRY IT yOURSELF

13-1. Estrellas: Busca la imagen de una estrella. Haz que aparezca una cuadrícula de estrellas en la pantalla.

13-2. Mejores estrellas: Puedes hacer un patrón de estrellas más realista introduciendo aleatoriedad al colocar cada estrella. Recuerda que puedes obtener un número aleatorio así:

```
from random import randint
número_aleatorio = randint(-10,10)
```

Este código devuelve un entero aleatorio entre -10 y 10. Usando el código del Ejercicio 13-1, ajusta la posición de cada estrella en una cantidad aleatoria.

Traslado de la flota

Ahora vamos a hacer que nuestra flota de alienígenas se mueva hacia la derecha a través de la pantalla hasta que llegue al borde, y entonces haremos que caiga una cantidad determinada y se mueva en la otra dirección. Continuaremos este movimiento hasta que todos los alienígenas hayan sido derribados, uno colisione con la nave, o uno llegue al fondo de la pantalla. Empecemos haciendo que la flota se mueva hacia la derecha.

Mover a los alienígenas hacia la derecha

Para mover a los alienígenas, utilizaremos un método `update()` en *alien.py*, que llamaremos para cada alienígena del grupo de alienígenas. En primer lugar, añadir un ajuste para controlar la velocidad de cada extranjero:

settings.py

```
def init (self):
    --snip--
    # Alien settings
    self.alien_speed_factor = 1
```

A continuación, utilice esta configuración para implementar `update()`:

alien.py

```
def update(self):
    """Mueve al alienígena a la derecha."""
❶   self.x += self.ai_settings.factor_velocidad_extraterrestre
❷   auto.rect.x = auto.x
```

Cada vez que actualizamos la posición de un alienígena, lo movemos hacia la derecha la cantidad almacenada en `alien_speed_factor`. Seguimos la posición exacta del alienígena con el atributo self.x, que puede contener valores decimales ❶. Luego usamos el valor de self.x para actualizar la posición del `rectángulo del alien` ❷.

En el bucle `while` principal, tenemos llamadas para actualizar el barco y las balas.

Ahora tenemos que actualizar también la posición de cada alienígena:

alien_invasion.py

```
# Inicia el bucle principal del juego.
mientras True:
    gf.check_events(ai_settings, screen, ship, bullets)
    ship.update()
    gf.update_bullets(balas)
    gf.update_aliens(aliens)
    gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Actualizamos las posiciones de los alienígenas después de que se hayan actualizado las balas, porque pronto comprobaremos si alguna bala ha alcanzado a algún alienígena.

Por último, añada la nueva función `update_aliens()` al final del archivo *game_functions.py*:

juego_functions.py

```
def actualizar_alienígenas(alienígenas):
    """Actualizar las posiciones de todos los alienígenas
    de la flota.""" aliens.update()
```

Usamos el método `update()` en el grupo de `aliens`, que automáticamente llama al método `update()` de cada alien. Cuando ejecutes Alien Invasion ahora, deberías ver la flota moverse a la derecha y desaparecer del lado de la pantalla.

Creación de ajustes para la dirección de la flota

Ahora crearemos los ajustes que harán que la flota se mueva hacia abajo en la pantalla y hacia la izquierda cuando toque el borde derecho de la pantalla. He aquí cómo implementar este comportamiento:

settings.py

```
        # Alien settings
        self.alien_speed_factor = 1
        self.fleet_drop_speed = 10
        # fleet_direction de 1 representa la derecha; -1 representa la
        izquierda. self.fleet_direction = 1
```

El ajuste `fleet_drop_speed` controla la rapidez con la que la flota desciende por la pantalla cada vez que un alienígena alcanza cualquiera de los bordes. Es útil separar esta velocidad de la velocidad horizontal de los alienígenas para poder ajustar las dos velocidades de forma independiente.

Para implementar el ajuste `fleet_direction`, podríamos usar un valor de texto, como `'izquierda'` o `'derecha'`, pero acabaríamos con sentencias `if-elif` comprobando la dirección de la flota. En su lugar, ya que sólo tenemos dos direcciones para tratar, vamos a utilizar los valores 1 y -1 y cambiar entre ellos cada vez que la flota cambia de dirección. (Usar números también tiene sentido porque moverse a la derecha implica sumar al valor de la coordenada x de cada alienígena, y moverse a la izquierda implica restar del valor de la coordenada x de cada alienígena).

Comprobar si un alienígena ha tocado el borde

Ahora necesitamos un método para comprobar si un alienígena está en cualquiera de los bordes, y necesitamos modificar `update()` para permitir que cada alienígena se mueva en la dirección apropiada:

alien.py

```
    def check_edges(self):
        """Devuelve True si el alienígena está en el borde de
        la pantalla.""" screen_rect = self.screen.get_rect()
❶       if self.rect.right >= screen_rect.right:
            return True
❷       elif self.rect.left <= 0:
            return True

    def update(self):
        """Mueve al alienígena a la derecha o a la izquierda."""
❸       self.x += (self.ai_settings.factor_velocidad_extraterrestre *
                        self.ai_settings.fleet_direction)
        self.rect.x = self.x
```

Podemos llamar al nuevo método `check_edges()` sobre cualquier alien para ver si está en el borde izquierdo o derecho. El alienígena está en el borde derecho si el atributo `derecho` de su `rect` es mayor o igual que el atributo `derecho` del `rect` de la pantalla ❶. Está en el borde izquierdo si su valor `izquierdo` es menor o igual que 0 ❷.

Modificamos el método `update()` para permitir el movimiento a izquierda o derecha ❸ multiplicando el factor de velocidad del alienígena por el valor de `fleet_direction`. Si `fleet_direction` es 1, el valor de `alien_speed_factor` se sumará a la posición actual del alien, moviéndolo a la derecha; si `fleet_direction` es -1, el valor se restará de la posición del alien, moviéndolo a la izquierda.

Abandonar la flota y cambiar de rumbo

Cuando un alienígena alcanza el borde, toda la flota debe descender y cambiar de dirección. Por lo tanto, tenemos que hacer algunos cambios sustanciales en *game_functions.py* porque ahí es donde comprobamos si algún alienígena está en el borde izquierdo o derecho. Haremos esto escribiendo las funciones `check_fleet_edges()` y `change_fleet_direction()`, y luego modificando `update_aliens()`:

juego_functions.py

```
def check_fleet_edges(ai_settings, aliens):
    """Responde apropiadamente si algún alienígena ha alcanzado un borde."""
❶   for alien in aliens.sprites():
        if alien.check_edges():
            change_fleet_direction(ai_settings, aliens)
            break
```

```
def cambiar_direccion_flota(ai_configuracion, extranjeros):
    """Suelta toda la flota y cambia la dirección de la flota"""
    for alien in aliens.sprites():
❷       alien.rect.y += ai_settings.fleet_drop_speed
    ai_settings.fleet_direction *= -1

def update_aliens(ai_settings, aliens):
    """
    Comprueba si la flota está en un borde,
      y luego actualiza las posiciones de todos los alienígenas
    de la flota. """
❸   check_fleet_edges(ai_settings, aliens)
    aliens.update()
```

En `check_fleet_edges()`, hacemos un bucle a través de la flota y llamamos a `check_edges()` en cada ❶ alienígena. Si check_edges() devuelve `True`, sabemos que un alienígena está en un borde y toda la flota necesita cambiar de dirección, así que llamamos a `change_fleet_direction()` y salimos del bucle. En `change_fleet_direction()`, hacemos un bucle a través de todos los alienígenas y soltamos a cada uno usando el ajuste `fleet_drop_speed` ❷; entonces cambiamos el valor de `fleet_direction` multiplicando su valor actual por -1.

Hemos modificado la función `update_aliens()` para determinar si hay algún alienígena en un borde llamando a `check_fleet_edges()` ❸. Esta función necesita un parámetro `ai_settings`, por lo que incluimos un argumento para `ai_settings` en la llamada a update_aliens():

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(bullets)
        gf.update_aliens(ai_settings, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Si ejecutas el juego ahora, la flota debería moverse hacia adelante y hacia atrás entre los bordes de la pantalla y bajar cada vez que toca un borde. Ahora podemos empezar a disparar a los alienígenas y vigilar si alguno choca contra la nave o llega al fondo de la pantalla.

TRY IT yOURSELF

13-3. Gotas de lluvia: Busca una imagen de una gota de lluvia y crea una cuadrícula de gotas de lluvia. Haga que las gotas de lluvia caigan hacia la parte inferior de la pantalla hasta que desaparezcan.

13-4. Lluvia Continua: Modifica tu código en el Ejercicio 13-3 para que cuando una fila de gotas de lluvia desaparezca de la parte inferior de la pantalla, una nueva fila aparezca en la parte superior de la pantalla y comience a caer.

Disparar a los alienígenas

Hemos construido nuestra nave y una flota de alienígenas, pero cuando las balas alcanzan a los alienígenas, simplemente los atraviesan porque no estamos comprobando si hay colisiones. En programación de juegos, *las colisiones* se producen cuando los elementos del juego se superponen. Para hacer que las balas derriben a los alienígenas, usaremos el método `sprite.groupcollide()` para buscar colisiones entre miembros de dos grupos.

Detección de colisiones de balas

Queremos saber inmediatamente cuándo una bala alcanza a un alienígena para poder hacer que desaparezca en cuanto sea alcanzado. Para ello, buscaremos colisiones inmediatamente después de actualizar la posición de una bala.

El método `sprite.groupcollide()` compara el `rectángulo de` cada bala con el `rectángulo` de cada alien y devuelve un diccionario que contiene las balas y los alien que han colisionado. Cada clave en el diccionario es una bala, y el valor correspondiente es el alien que fue golpeado. (Usaremos este diccionario cuando implementemos un sistema de puntuación en el capítulo 14).

Utilice este código para comprobar si hay colisiones en la función `update_bullets()`:

juego_functions.py

```
def actualizar_balas(aliens, balas):
    """Actualizar posición de viñetas y deshacerse de viñetas antiguas."""
    --snip--
    # Comprueba si hay alguna bala que haya
    alcanzado al alien. # Si es así, deshazte de
    la bala y del alien.
    colisiones = pygame.sprite.groupcollide(balas, alienígenas, Verdadero,
    Verdadero)
```

La nueva línea que hemos añadido recorre cada bala del grupo `balas` y luego recorre cada alienígena del grupo `alienígenas`. Cada vez que los `recuadros` de una bala y un alienígena se solapan, `groupcollide()` añade un par clave-valor al diccionario que devuelve. Los dos argumentos `True` indican a Pygame si debe borrar las balas y alienígenas que han colisionado. (Para hacer una bala de alta potencia que es capaz de viajar a la parte superior de la pantalla, destruyendo todos los alienígenas en su camino, se podría establecer el primer argumento booleano en `False` y mantener el segundo argumento booleano en `True`. Los alienígenas alcanzados desaparecerían, pero todas las balas permanecerían activas hasta que desaparecieran de la parte superior de la pantalla).

Pasamos el argumento `aliens` en la llamada a `update_bullets()`:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(aliens, balas)
        gf.update_aliens(ai_settings, aliens)
```

```
        gf.update_screen(ai_settings, pantalla, nave, aliens, balas)
```

Cuando ejecutes ahora Invasión Alienígena, los alienígenas que golpees deberían desaparecer.

La Figura 13-5 muestra una flota que ha sido parcialmente derribada.



Figura 13-5: ¡Podemos disparar a los alienígenas!

Fabricación de balas más grandes para pruebas

Puedes probar muchas características del juego simplemente ejecutando el juego, pero algunas características son tediosas de probar en la versión normal de un juego. Por ejemplo, es mucho trabajo derribar a todos los alienígenas de la pantalla varias veces para probar si tu código responde correctamente a una flota vacía.

Para probar determinadas características, puedes cambiar ciertos ajustes del juego para centrarte en un área concreta. Por ejemplo, puedes encoger la pantalla para que haya menos alienígenas a los que disparar o aumentar la velocidad de las balas y disponer de muchas balas a la vez.

Mi cambio favorito para probar Invasión Alienígena es usar balas superanchas que permanecen activas incluso después de haber alcanzado a un alienígena (ver Figura 13-6). Prueba a poner `bullet_width` a 300 para ver lo rápido que puedes derribar la flota.

Cambios como estos te ayudarán a probar el juego de forma más eficaz y posiblemente te aporten ideas para dar a los jugadores poderes adicionales. (Recuerda restablecer la configuración normal cuando hayas terminado de probar una función).



Figura 13-6: Las balas extra potentes hacen que algunos aspectos del juego sean más fáciles de probar.

Repoblar la flota

Una característica clave de Alien Invasion es que los alienígenas son implacables: cada vez que se destruye la flota, aparece una nueva.

Para hacer aparecer una nueva flota de alienígenas después de que una flota haya sido destruida, primero comprobamos si el grupo `aliens` está vacío. Si lo está, llamamos a `create_fleet()`. Realizaremos esta comprobación en `update_bullets()` porque ahí es donde se destruyen los alienígenas individuales:

juego_functions.py

```
def update_bullets(ai_settings, screen, ship, aliens, bullets):
    --snip--
    # Comprueba si hay alguna bala que haya
    alcanzado al alien. # Si es así, deshazte de
    la bala y del alien.
    colisiones = pygame.sprite.groupcollide(balas, alienígenas, Verdadero,
    Verdadero)

    si len(aliens) == 0:                                    ❶
     # Destruye las balas existentes y crea una nueva flota.
        balas.empty()                                       ❷
        create_fleet(ai_settings, screen, ship, aliens)
```

En ❶ comprobamos si el grupo `aliens` está vacío. Si lo está, nos deshacemos de cualquier bala existente usando el método `empty()`, que elimina todos los sprites restantes de un grupo ❷. También llamamos a `create_fleet()`, que vuelve a llenar la pantalla de aliens.

La definición de `update_bullets()` tiene ahora los parámetros adicionales `ai_settings`, `screen`, y `ship`, por lo que necesitamos actualizar la llamada a `update_bullets()` en *alien_invasion.py*:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
        gf.update_aliens(ai_settings, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Ahora aparece una nueva flota en cuanto destruyes la flota actual.

Acelerar las balas

Si has intentado disparar a los alienígenas en el estado actual del juego, habrás notado que las balas se han ralentizado un poco. Esto se debe a que Pygame
está haciendo ahora más trabajo en cada pasada por el bucle. Podemos aumentar la velocidad de las balas ajustando el valor de `bullet_speed_factor` en *settings.py*. Si aumentamos este valor (a 3, por ejemplo), las balas deberían volver a subir por la pantalla a una velocidad razonable:

settings.py

```
        # Ajustes de bala
        self.factor_velocidad_bala =
        3
        self.bullet_width = 3
        --snip--
```

El mejor valor para este ajuste depende de la velocidad de tu sistema, así que encuentra un valor que te funcione.

Refactorización de update_bullets()

Vamos a refactorizar `update_bullets()` para que no haga tantas tareas diferentes. Vamos a mover el código para hacer frente a las colisiones bala-alien a una función separada:

juego_functions.py

```
def update_bullets(ai_settings, screen, ship, aliens, bullets):
    --snip--
    # Deshacerse de las viñetas que han
    desaparecido. for viñeta in
    viñetas.copiar():
        if bullet.rect.bottom <= 0:
            bullets.remove(bullet)

    check_bullet_alien_collisions(ai_settings, screen, ship, aliens, bullets)

def check_bullet_alien_collisions(ai_settings, screen, ship, aliens, bullets):
    """Responder a colisiones bala-alien."""
    # Elimina las balas y los alienígenas que hayan colisionado.
```

```
colisiones = pygame.sprite.groupcollide(balas, alienígenas, Verdadero,
Verdadero)
```

```
    si len(aliens) == 0:
        # Destruye las balas existentes y crea una nueva
        flota. bullets.empty()
        create_fleet(ai_settings, screen, ship, aliens)
```

Hemos creado una nueva función, `check_bullet_alien_collisions()`, para buscar colisiones entre balas y alienígenas, y responder apropiadamente si toda la flota ha sido destruida. Esto evita que `update_bullets()` se alargue demasiado y simplifica el desarrollo posterior.

TRY IT yOURSELF

13-5. Atrapar: Crea un juego que coloque en la parte inferior de la pantalla un personaje que puedas mover a izquierda y derecha. Haz que aparezca una bola en una posición aleatoria en la parte superior de la pantalla y que caiga hacia abajo a un ritmo constante. Si tu p e r s o n a j e "atrapa" la bola chocando con ella, haz que la bola desaparezca. Crea una nueva bola cada vez que tu personaje atrape la bola o cada vez que la bola desaparezca d e l a parte inferior de la pantalla.

Fin del juego

¿Cuál es la diversión y el desafío en un juego si no se puede perder? Si el jugador no derriba la flota lo bastante rápido, haremos que los alienígenas destruyan la nave si la alcanzan. Al mismo tiempo, limitaremos el número de naves que puede utilizar un jugador y destruiremos la nave cuando un alien llegue al fondo de la pantalla. Terminaremos el juego cuando el jugador haya usado todas sus naves.

Detección de colisiones entre naves alienígenas

Empezaremos comprobando si hay colisiones entre los alienígenas y la nave, para poder responder adecuadamente cuando un alienígena choque contra ella. Comprobaremos las colisiones entre alienígenas y naves inmediatamente después de actualizar la posición de cada alienígena:

juego_funciones.py

```
def update_aliens(ai_settings, ship, aliens):
    """
    Comprueba si la flota está en un borde,
      y luego actualiza las posiciones de todos los alienígenas
    de la flota. """
    check_fleet_edges(ai_settings, aliens)
    aliens.update()

    # Busca colisiones entre naves alienígenas.
1   if pygame.sprite.spritecollideany(nave, aliens):
2       print("¡¡¡Barco alcanzado!!!")
```

El método spritecollideany() toma dos argumentos: un sprite y un grupo. El método busca cualquier miembro del grupo que haya colisionado con el sprite y para el bucle a través del grupo tan pronto como encuentra un miembro que ha colisionado con el sprite. En este caso, recorre el grupo de los alienígenas y devuelve el primer alienígena que encuentre que haya colisionado con la nave.

Si no hay colisiones, spritecollideany() devuelve None y el bloque if en ❶ no se ejecuta. Si encuentra un alienígena que ha colisionado con la nave, devuelve ese alienígena y el bloque if se ejecuta: ¡¡¡Imprime *Ship hit!!!* ❷.

(Cuando un alien impacta con la n a v e, tendremos que hacer una serie de tareas: tendremos que borrar todos los aliens y balas restantes, recentrar la n a v e, y crear una nueva flota.
Antes de escribir código para hacer todo esto, necesitamos saber que nuestro enfoque para detectar colisiones entre naves alienígenas funciona correctamente. Escribir una sentencia print es una forma sencilla de asegurarnos de que estamos detectando las colisiones correctamente).

Ahora tenemos que pasar el barco a update_aliens():

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, ship, bullets)
        ship.update()
        gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
        gf.update_aliens(ai_settings, ship, aliens)
        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

Ahora, cuando ejecutes Invasión alienígena, *¡¡¡Nave impactada!!!* debería aparecer en el ter- minal cada vez que un alienígena choque contra la nave. Cuando pruebes esta función, ajusta alien_drop_speed a un valor más alto, como 50 o 100, para que los alienígenas alcancen tu nave más rápido.

Respuesta a las colisiones de naves alienígenas

Ahora tenemos que averiguar qué ocurre cuando un alienígena colisiona con la nave. En lugar de destruir la instancia de la nave y crear una nueva, contaremos cuántas veces ha sido golpeada la nave mediante las estadísticas de seguimiento del juego. (Las estadísticas de seguimiento también serán útiles para la puntuación).

Escribamos una nueva clase, GameStats, para realizar un seguimiento de las estadísticas del juego, y guardémosla como *game_stats.py*:

game_stats.py

```
clase GameStats():
    """Estadísticas de seguimiento de Invasión alienígena."""

    def init (self, ai_settings):
        """Inicializar estadísticas."""
        self.ai_settings = ai_settings
❶       self.reset_stats()

    def reset_stats(self):
```

```
        """Inicializar las estadísticas que pueden cambiar durante el juego."""
        self.ships_left = self.ai_settings.ship_limit
```

Crearemos una instancia de `GameStats` para todo el tiempo que se ejecute Alien Invasion, pero tendremos que restablecer algunas estadísticas cada vez que se inicie el jugador

un nuevo juego. Para ello, inicializaremos la mayoría de las estadísticas en el método
`reset_stats()` en lugar de directamente en `init ()`. Llamaremos a este método desde
`init ()` para que las estadísticas se establezcan correctamente cuando se cree por primera vez la instancia `GameStats` ❶, pero también podremos llamar a `reset_stats()` cada vez que el jugador comience una nueva partida.

Ahora mismo sólo tenemos una estadística, `ships_left`, cuyo valor cambiará a lo largo del juego. El número de naves con las que comienza el jugador se almacena en *settings.py* como `ship_limit`:

settings.py

```
        # Configuración de la nave
        self.ship_speed_factor = 1.5
        self.ship_limit = 3
```

También tenemos que hacer algunos cambios en *alien_invasion.py*, para crear una instancia de `GameStats`:

alien_invasión.py

```
--snip--
from settings import Configuración
❶ from game_stats import GameStats
--snip--

def ejecutar_juego():
    --snip--
    pygame.display.set_caption("Invasión alienígena")

    # Crear una instancia para almacenar las estadísticas del juego.
❷   stats = GameStats(ai_settings)
    --snip--
    # Inicia el bucle principal del
    juego. while True:
        --snip--
        gf.update_bullets(ai_settings, pantalla, nave, aliens, balas)
❸       gf.update_aliens(ai_settings, stats, screen, ship, aliens, bullets)
        --snip--
```

Importamos la nueva clase `GameStats` ❶, hacemos una instancia de `stats` ❷, y luego añadimos los argumentos `stats`, `screen`, y `ship` en la llamada a `update_aliens()` ❸.

Utilizaremos estos argumentos para hacer un seguimiento del número de naves que le quedan al jugador y para construir una nueva flota cuando un alienígena alcance la nave.

Cuando un alienígena impacta contra la nave, restamos uno al número de naves restantes, destruimos todos los alienígenas y balas existentes, creamos una nueva flota y volvemos a colocar la nave en el centro de la pantalla. (También pausaremos el juego para
un momento para que el jugador pueda darse cuenta de la colisión y reagruparse antes de que aparezca una nueva flota).

Pongamos la mayor parte de este código en la función `ship_hit()`:

juego functions.py

```
importar sys
❶ from tiempo import sueño.

importar pygame
--snip--
```

```
def ship_hit(ai_settings, stats, screen, ship, aliens, bullets):
    """Responde a la nave que es golpeada por un alienígena."""
    # Decrement ships_left.
❷   stats.barcos_izquierda -= 1

    # Vacía la lista de alienígenas y balas.
❸   aliens.empty()
    bullets.empty()

    # Crea una nueva flota y centra la nave.
❹   create_fleet(ai_settings, screen, ship, aliens)
    ship.center_ship()

    # Pausa.
❺   sleep(0.5)

❻ def update_aliens(ai_settings, stats, screen, ship, aliens, bullets):
    --snip--
    # Busca colisiones entre naves alienígenas.
    if pygame.sprite.spritecollideany(nave, aliens):
        ship_hit(ai_settings, stats, pantalla, nave, aliens, balas)
```

Primero importamos la función `sleep()` del módulo de `tiempo` para pausar el juego ❶. La nueva función `ship_hit()` coordina la respuesta cuando la nave es alcanzada por un alienígena. Dentro de `ship_hit()`, el número de naves restantes se reduce en 1 ❷, tras lo cual vaciamos los grupos `aliens` y `balas` ❸.

A continuación, creamos una nueva flota y centramos la nave ❹. (Añadiremos el método `center_ship()` a `Ship` en un momento). Por último, hacemos una pausa después de que las actualizaciones se han hecho a todos los elementos del juego, pero antes de que cualquier cambio se haya dibujado en la pantalla para que el jugador pueda ver que su nave ha sido golpeada ❺. La pantalla se congelará momentáneamente, y el jugador verá que el alienígena ha golpeado la nave. Cuando la función `sleep()` termine, el código pasará a la función `update_screen()`, que dibujará la nueva flota en la pantalla.

También actualizamos la definición de `update_aliens()` para incluir los parámetros `stats`, `screen`, y `bullets` ❻ para que pueda pasar estos valores en la llamada a `ship_hit()`.

Aquí está el nuevo método `center_ship()`; añádelo al final de *ship.py*:

barco.py

```
    def centrar_barco(self):
        """Centra la nave en la pantalla"""
        self.center = self.screen_rect.centerx
```

Para centrar la nave, establecemos el valor del atributo `center` de la nave para que coincida con el centro de la pantalla, que obtenemos a

través del atributo `screen_rect`.

NOTE *Fíjate en que nunca hacemos más de una nave; sólo hacemos una instancia de nave para toda la partida y la recentramos cada vez que la nave ha sido golpeada. La estadística* ships_left *nos dirá cuando el jugador se ha quedado sin naves.*

Ejecuta el juego, dispara a unos cuantos alienígenas y deja que un alienígena golpee la nave. El juego debería pausarse, y debería aparecer una nueva flota con la nave centrada de nuevo en la parte inferior de la pantalla.

Alienígenas que llegan al fondo de la pantalla

Si un alien llega a la parte inferior de la pantalla, responderemos de la misma manera que cuando un alien golpea la nave. Añade una nueva función para realizar esta comprobación, y llámala desde `update_aliens()`:

juego_funciones.py

```
def check_aliens_bottom(ai_settings, stats, screen, ship, aliens, bullets):
    """Comprueba si algún alienígena ha llegado a la parte inferior
    de la pantalla."""
    for alien in aliens.sprites():
        if alien.rect.bottom >= screen.get_rect().bottom:
            # Trata esto igual que si la nave fuera alcanzada.
            ship_hit(ai_settings, stats, screen, ship, aliens, bullets)
            break

def update_aliens(ai_settings, stats, screen, ship, aliens, bullets):
    --snip--
    # Busca alienígenas golpeando la parte inferior de la pantalla.
    check_aliens_bottom(ai_settings, stats, screen, ship, aliens, bullets)
```

❶ ❷

La función `check_aliens_bottom()` comprueba si algún alienígena ha llegado a la parte inferior de la pantalla. Un alienígena llega al fondo cuando su valor `rect.bottom` es mayor o igual que el atributo `rect.bottom` ❶ de la pantalla. Si un alien alcanza el fondo, llamamos a `ship_hit()`. Si un alien toca el fondo, no hay necesidad de comprobar el resto, así que salimos del bucle después de llamar a `ship_hit()`.

Llamamos a `check_aliens_bottom()` después de actualizar las posiciones de todos los alienígenas y después de buscar colisiones alienígena-nave ❷. Ahora aparecerá una nueva flota cada vez que la nave sea golpeada por un alienígena o un alienígena llegue al fondo de la pantalla.

¡Se acabó el juego!

Alien Invasion parece más completo ahora, pero el juego nunca termina. El valor de `ships_left` simplemente crece cada vez más negativo. Añadamos una bandera `game_active` como atributo a `GameStats` para terminar el juego cuando el jugador se quede sin naves:

game_stats.py

```
    def init (self, settings):
        --snip--
```

```
#                game_active = True

I
n
i
c
i
a

A
l
i
e
n

I
n
v
a
s
i
o
n

e
n

u
n

e
s
t
a
d
o

a
c
t
i
v
o
.

s
e
l
f
.
```

Ahora añadimos código a `ship_hit()` que establece `game_active` a `False` si el jugador ha usado todas sus naves:

juego_functions.py

```
def ship_hit(ai_settings, stats, screen, ship, aliens, bullets):
    """Responder a nave golpeada por
    alien.""" if stats.naves_izquierda > 0."
        # Disminuye naves_izquierda.
        stats.naves_izquierda -= 1
        --snip--
        # Pausa.
        dormir(0.5)

    si no:
        stats.game_active = False
```

La mayor parte de `ship_hit()` no ha cambiado. Hemos movido todo el código existente a un bloque `if`, que comprueba que al jugador le queda al menos una nave. Si es así, creamos una nueva flota, pausamos y continuamos. Si al jugador no le quedan naves, establecemos `game_active` a `False`.

Identificar cuándo deben ejecutarse las partes del juego

En *alien_invasion.py necesitamos* identificar las partes del juego que deben ejecutarse siempre y las partes que deben ejecutarse sólo cuando el juego está activo:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, pantalla, nave, balas)

        if stats.game_active:
            nave.actualizar()
            gf.update_bullets(ai_settings, screen, ship, aliens, bullets)
            gf.update_aliens(ai_settings, stats, screen, ship, aliens, bullets)

        gf.update_screen(ai_settings, screen, ship, aliens, bullets)
```

En el bucle principal, siempre necesitamos llamar a `check_events()`, incluso si el juego está inactivo. Por ejemplo, todavía necesitamos saber si el usuario pulsa Q para salir del juego o hace clic en el botón para cerrar la ventana. También seguimos actualizando la pantalla para poder hacer cambios en la pantalla mientras esperamos a ver si el jugador decide empezar una nueva partida. El resto de las llamadas a funciones sólo tienen que ocurrir cuando el juego está activo, porque cuando el juego está inactivo, no necesitamos actualizar las posiciones de los elementos del juego.

Ahora, cuando juegues a Alien Invasion, el juego debería congelarse cuando hayas usado todas tus naves.

TRY IT yOURSELF

13-6. Se acabó el juego: Usando tu código del Ejercicio 13-5 (página 284), lleva la cuenta del número de veces que el jugador pierde la pelota. Cuando hayan fallado la pelota tres veces, termina el juego.

Resumen

En este capítulo aprendiste cómo añadir un gran número de elementos idénticos a un juego mediante la creación de una flota de alienígenas. Aprendiste a utilizar bucles anidados para crear una cuadrícula de elementos, e hiciste que un gran conjunto de elementos del juego se movieran llamando al método `update()` de cada elemento. Aprendiste a controlar la dirección de los objetos en la pantalla y cómo responder a eventos, como cuando la flota alcanza el borde de la pantalla. También aprendió a detectar y responder a las colisiones cuando las balas golpean a los alienígenas y los alienígenas golpean la nave. Por último, aprendiste a realizar un seguimiento de las estadísticas en un juego y a utilizar una bandera `game_active` para determinar cuándo había terminado el juego.

En el capítulo final de este proyecto, añadiremos un botón Jugar para que el jugador pueda elegir cuándo empezar su primera partida y si quiere volver a jugar cuando ésta termine. Aceleraremos el juego cada vez que el jugador derribe toda la flota y añadiremos un sistema de puntuación. El resultado final será un juego totalmente jugable.

14

S C O R ING



En este capítulo terminaremos el juego Invasión Alienígena. Añadiremos un botón de Play para iniciar una partida bajo demanda o para reiniciar una partida una vez que termine. También cambiaremos el juego para que se acelere cuando el jugador suba de nivel e implementaremos un sistema de puntuación. Al final del capítulo, sabrás lo suficiente para empezar a escribir juegos que aumenten su dificultad a medida que el jugador progresa y que muestren puntuaciones.

Añadir el botón Reproducir

En esta sección añadiremos un botón Jugar que aparecerá antes de que empiece una partida y reaparecerá cuando ésta termine para que el jugador pueda volver a jugar.

Ahora mismo el juego comienza en cuanto ejecutas *alien_invasion.py*. Vamos a iniciar el juego en un estado inactivo y luego pedir al jugador que haga clic en un botón Reproducir para comenzar. Para ello, introduzca lo siguiente en *game_stats.py*:

game_stats.py

```
    def init (self, ai_settings):
        """Inicializar estadísticas."""
        self.ai_settings = ai_settings
        self.reset_stats()

        # Iniciar el juego en un estado
        inactivo. self.game_active = False

    def reset_stats(self):
        --snip--
```

Ahora el juego debe comenzar en un estado inactivo sin forma de que el jugador lo inicie hasta que hagamos un botón de Play.

Creación de una clase Button

Como Pygame no tiene un método incorporado para hacer botones, escribiremos una clase `Button` para crear un rectángulo relleno con una etiqueta. Puedes usar este código para crear cualquier botón en un juego. Aquí está la primera parte de la clase `Button`; guárdala como *button.py*:

botón.py

```
import pygame.font

clase Button():

❶   def init (self, ai_settings, screen, msg):
        """Inicializar los atributos de los
        botones."""  self.screen = screen
        self.screen_rect = screen.get_rect()

        # Establece las dimensiones y propiedades del botón.
❷       auto.anchura, auto.altura = 200, 50
        self.color_botón = (0, 255, 0)
        self.text_color = (255, 255, 255)
❸       self.font = pygame.font.SysFont(None, 48)

        # Construye el objeto rect del botón y céntralo.
❹       self.rect = pygame.Rect(0, 0, self.width, self.height)
        self.rect.center = self.screen_rect.center

        # El mensaje del botón sólo necesita ser preparado una vez.
❺       self.prep_msg(msg)
```

Primero importamos el módulo `pygame.font`, que permite a Pygame renderizar texto en la pantalla. El método `init ()` toma los parámetros `self`, los objetos `ai_settings` y `screen`, y `msg`, que contiene el texto para el botón ❶. Establecemos las dimensiones del botón en ❷, y luego establecemos `button_color` para colorear el objeto `rect` del botón en verde brillante y establecemos `text_color` para renderizar el texto en blanco.

En ❸ preparamos un atributo de `fuente` para renderizar el texto. El argumento `None` le dice a Pygame que use la fuente por defecto, y `48` determina el tamaño del texto. Para centrar el botón en la pantalla, creamos un `rect` para el botón ❹ y establecemos su atributo `center` para que coincida con el de la pantalla.

Pygame trabaja con texto renderizando la cadena que quieres mostrar como una imagen. En ❺ llamamos a `prep_msg()` para manejar este renderizado.

Este es el código de `prep_msg()`:

button.py

```
    def prep_msg(self, msg):
        """Convierte el mensaje en una imagen renderizada y centra el texto en
        el botón."""
❶       self.msg_image = self.font.render(msg, True, self.text_color,
            self.button_color)
❷       self.msg_image_rect = self.msg_image.get_rect()
        self.msg_image_rect.center = self.rect.center
```

El método `prep_msg()` necesita un parámetro `self` y el texto a renderar como imagen (`msg`). La llamada a `font.render()` convierte el texto almacenado en `msg` en una imagen, que luego almacenamos en `msg_image` ❶. El método font.`render()` también toma un valor booleano para activar o desactivar el antialiasing (el antialiasing suaviza los bordes del texto). Los argumentos restantes son el color de fuente y el color de fondo especificados. Fijamos el antialiasing en `True` y el fondo del texto en el mismo color que el botón. (Si no incluyes un color de fondo, Pygame intentará renderizar la fuente con un fondo trans- parente).

En ❷ centramos la imagen de texto en el botón creando un `rect` a partir de la imagen y estableciendo su atributo de `centro` para que coincida con el del botón.

Finalmente, creamos un método `draw_button()` que podemos llamar para mostrar el botón en pantalla:

button.py

```
    def dibujar_botón(self):
        # Dibuja el botón en blanco y luego el mensaje.
        self.screen.fill(self.button_color, self.rect)
        self.screen.blit(self.msg_image, self.msg_image_rect)
```

Llamamos a screen.`fill()` para dibujar la parte rectangular del botón. Luego llamamos a `screen.blit()` para dibujar la imagen del texto en la pantalla, pasándole una imagen y el objeto `rect` asociado con la imagen. Esto completa la clase `Button`.

Dibujar el botón en la pantalla

Usaremos la clase `Button` para crear un botón Play. Como sólo necesitamos un botón Reproducir, crearemos el botón directamente en *alien_invasion.py* como se muestra aquí:

alien_invasión.py

```
--snip--
from game_stats import GameStats
from button import Botón
--snip--

def ejecutar_juego():
    --snip--
    pygame.display.set_caption("Invasión alienígena")

    # Haz el botón Reproducir.
❶   play_button = Button(ai_settings, screen, "Play")
    --snip--

    # Inicia el bucle principal del
    juego. while True:
        --snip--
❷       gf.update_screen(ai_settings, screen, stats, ship, aliens, bullets,
            play_button)

run_game()
```

Importamos `Button` y creamos una instancia llamada `play_button` ❶, y luego pasamos `play_button` a `update_screen()` para que el botón aparezca cuando se actualice la pantalla ❷.

A continuación, modifica `update_screen()` para que el botón Jugar sólo aparezca cuando el juego esté inactivo:

juego_funciones.py

```
def update_screen(ai_settings, screen, stats, ship, aliens, bullets,
        play_button):
    """Actualiza las imágenes de la pantalla y pasa a la nueva pantalla."""
    --snip--

    # Dibuja el botón de jugar si el juego está
    inactivo. if not stats.game_active:
        play_button.draw_button()

    # Hacer visible la pantalla dibujada más recientemente.
    pygame.display.flip()
```

Para que el botón Jugar sea visible por encima de todos los demás elementos de la pantalla, lo dibujamos después de haber dibujado todos los demás elementos del juego y antes de pasar a una nueva pantalla. Ahora cuando ejecute Alien Invasion debería ver un botón Jugar en el centro de la pantalla, como se muestra en la Figura 14-1.

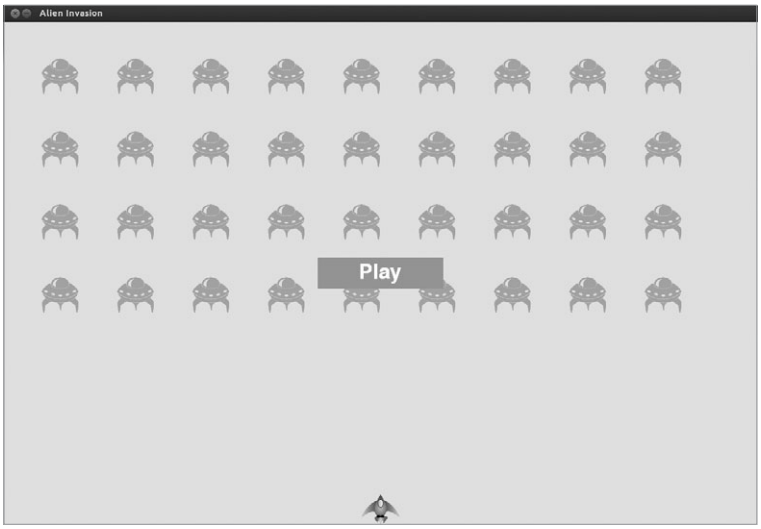


Figura 14-1: Aparece un botón Jugar cuando el juego está inactivo.

Iniciar el juego

Para iniciar una nueva partida cuando el jugador haga clic en Jugar, añada el siguiente código a *game_functions.py* para monitorizar los eventos del ratón sobre el botón:

juego_funciones.py

```
def check_events(ai_settings, screen, stats, play_button, ship, bullets):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        if evento.type == pygame.QUIT:
            --snip--
❶       elif event.type == pygame.MOUSEBUTTONDOWN:
❷           mouse_x, mouse_y = pygame.mouse.get_pos()
❸           check_play_button(stats, play_button, mouse_x, mouse_y)

def check_play_button(stats, play_button, mouse_x, mouse_y):
    """Empieza una nueva partida cuando el jugador haga clic
    en Jugar."""
❹   if play_button.rect.collidepoint(mouse_x, mouse_y):
        stats.game_active = True
```

Hemos actualizado la definición de `check_events()` para que acepte los parámetros `stats` y `play_button`. Utilizaremos `stats` para acceder a la bandera `game_active` y `play_button` para comprobar si se ha pulsado el botón Play.

Pygame detecta un evento `MOUSEBUTTONDOWN` cuando el jugador hace clic en cualquier parte de la pantalla ❶, pero queremos restringir nuestro juego para responder a los clics del ratón sólo en el botón Reproducir. Para conseguirlo, usamos `pygame.mouse.get_pos()`, que devuelve una tupla que contiene las coordenadas x e y del cursor del ratón cuando se pulsa el botón

del ratón ❷. Enviamos estos valores a la función `check_play_button()` ❸, que utiliza `collidepoint()` para ver si el punto del clic del ratón se solapa con la región definida por el `rectángulo del` botón Play ❹. Si es así, establecemos `game_active` a `True`, ¡y el juego comienza!

La llamada a `check_events()` en *alien_invasion.py* necesita pasar dos argumentos adicionales, `stats` y `play_button`:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            bullets)
        --snip--
```

En este punto, deberías poder empezar y jugar una partida completa. Cuando el juego termine, el valor de `game_active` debería convertirse en `False` y el botón Play debería reaparecer.

Reiniciar el juego

El código que acabamos de escribir funciona la primera vez que el jugador hace clic en Jugar, pero no una vez que la primera partida termina, porque las condiciones que provocaron que la partida terminara no se han restablecido.

Para reiniciar el juego cada vez que el jugador haga clic en Jugar, tenemos que reiniciar las estadísticas del juego, eliminar los antiguos alienígenas y balas, construir una nueva flota y centrar la nave, como se muestra aquí:

juego_funciones.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        balas, mouse_x, mouse_y):
    """Iniciar una nueva partida cuando el jugador haga
    clic en Jugar."""  if
    play_button.rect.collidepoint(mouse_x, mouse_y):
        # Reinicia las estadísticas del juego.
❶       stats.reset_stats()
        stats.game_active = True

        # Vacía la lista de alienígenas y balas.
❷       aliens.empty()
        bullets.empty()

        # Crea una nueva flota y centra la nave.
❸       create_fleet(ai_settings, screen, ship, aliens)
        ship.center_ship()
```

Actualizamos la definición de `check_play_button()` para que tenga acceso a `ai_settings`, `stats`, `nave`, `aliens` y `balas`. Necesita estos objetos para restablecer los ajustes que han cambiado durante la partida y para refrescar los elementos visuales del juego.

En ❶ reiniciamos las estadísticas del juego, lo que da al jugador tres nuevas naves. Luego establecemos `game_active` a `True` (para que el juego comience tan pronto como el código de esta función termine de ejecutarse), vaciamos los grupos de `aliens` y `balas` ❷, y creamos una nueva flota y centramos la nave ❸.

Es necesario modificar la definición de `check_events()`, así como la llamada a `check_play_button()`:

juego_funciones.py

```
def check_events(ai_settings, screen, stats, play_button, ship, aliens,
        balas):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        if evento.type == pygame.QUIT:
            --snip--
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
❶           check_play_button(ai_settings, pantalla, estadísticas,
                play_button, nave, aliens, balas, mouse_x, mouse_y)
```

La definición de `check_events()` necesita el parámetro `aliens`, que pasará a check_play_button(). A continuación, actualizamos la llamada a check_play_button() para que pase los argumentos adecuados ❶.

Ahora actualiza la llamada a `check_events()` en *alien_invasion.py* para que pase el comando
argumento de `los alienígenas`:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            aliens, bullets)
        --snip--
```

Ahora el juego se reiniciará correctamente cada vez que hagas clic en Jugar, ¡lo que te permitirá jugar tantas veces como quieras!

Desactivar el botón de reproducción

Un problema con nuestro botón Jugar es que la región del botón en la pantalla seguirá respondiendo a los clics incluso cuando el botón Jugar no esté visible. Haz clic accidentalmente en el área del botón Jugar una vez que la partida haya comenzado y ¡el juego se reiniciará!

Para solucionarlo, configura el juego para que sólo se inicie cuando `game_active` sea `False`:

juego_funciones.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        balas, mouse_x, mouse_y):
    """Iniciar una nueva partida cuando el jugador haga clic en Jugar."""
❶   button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
❷   if button_clicked and not stats.game_active:
        # Reinicia las estadísticas del juego.
        --snip--
```

La bandera `button_clicked` almacena un valor `True` o `False` ❶, y el juego se reiniciará sólo si se pulsa Play *y* el juego no está actualmente activo ❷.

Para probar este comportamiento, inicie un nuevo juego y haga clic

repetidamente donde debería estar el botón Reproducir. Si todo funciona como se espera, hacer clic en el área del botón Reproducir no debería tener ningún efecto en el juego.

Ocultar el cursor del ratón

Queremos que el cursor del ratón esté visible para empezar a jugar, pero una vez que empieza el juego sólo estorba. Para solucionar esto, lo haremos invisible una vez que el juego se active:

juego_funciones.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        balas, mouse_x, mouse_y):
    """Iniciar nueva partida cuando el jugador haga clic en Jugar."""
    button_clicked = play_button.rect.collidepoint(mouse_x, mouse_y)
    if button_clicked and not stats.game_active:
        # Ocultar el cursor del ratón.
        pygame.mouse.set_visible(False)
        --snip--
```

Pasando `False` a `set_visible()` le dice a Pygame que oculte el cursor cuando el ratón está sobre la ventana del juego.

Haremos que el cursor reaparezca una vez que el juego termine para que el jugador pueda hacer clic en Reproducir para comenzar un nuevo juego. Aquí está el código para hacerlo:

juego_functions.py

```
def ship_hit(ai_settings, screen, stats, ship, aliens, bullets):
    """Responder a nave golpeada por
    alien.""" if stats.naves_izquierda > 0."
        --snip-- si
    no:
        stats.game_active = False
        pygame.mouse.set_visible(True)
```

Hacemos el cursor visible de nuevo tan pronto como el juego se vuelve inactivo, lo que ocurre en `ship_hit()`. La atención a detalles como este hace que tu juego parezca más profesional y permite al jugador centrarse en jugar en lugar de en descifrar la interfaz de usuario.

TRY IT yOURSELF

14-1. Pulsar P para jugar: Dado que Alien Invasion utiliza el teclado para controlar la nave, es mejor iniciar el juego pulsando una tecla. Agregue código que permita al jugador presionar P para comenzar. Puede ayudar mover algo de código de `check_play_button()` a una función `start_game()` que pueda ser llamada desde `check_play_button()` y `check_keydown_events()`.

14-2. Práctica de puntería: Cree un rectángulo en el borde derecho de la pantalla que se mueva hacia arriba y hacia abajo a un ritmo constante. A continuación, haz que aparezca una nave en el lado izquierdo de la pantalla que el jugador pueda mover hacia arriba y hacia abajo mientras dispara balas al blanco rectangular en movimiento. Añade un botón Play que inicie el juego, y
cuando el jugador falle el blanco tres veces, finalice el juego y haga reaparecer el botón Jugar. Deja que el jugador reinicie el juego con este botón Play.

Subir de nivel

En nuestro juego actual, una vez que el jugador derriba toda la flota alienígena, alcanza un nuevo nivel, pero la dificultad del juego no cambia. Animemos un poco las cosas y hagamos que el juego sea más desafiante aumentando la velocidad del juego cada vez que un jugador despeja la pantalla.

Modificación de los ajustes de velocidad

Primero reorganizaremos la clase `Settings` para agrupar los ajustes del juego en estáticos y cambiantes. También nos aseguraremos de que los ajustes que cambian en el transcurso de un juego se reinician cuando empezamos un nuevo juego. Aquí está la clase método `init ()` de *settings.py*:

settings.py

```
    def init (self):
        """Inicializar la configuración estática del
        juego.""" # Ajustes de pantalla
        self.ancho_pantalla = 1200
        self.altura_pantalla = 800
        self.bg_color = (230, 230, 230)

        # Ship settings
        self.ship_limit = 3

        # Bullet settings
        self.bullet_width = 3
        self.bullet_height = 15
        self.bullet_color = 60, 60, 60
        self.balas_permitidas = 3

        # Alien settings
        self.fleet_drop_speed = 10

        # Cómo de rápido se acelera el juego
❶       self.speedup_scale = 1.1

❷       self.initialize_dynamic_settings()
```

Continuamos inicializando los ajustes que permanecen constantes en el método `init ()`. En ❶ añadimos un ajuste `speedup_scale` para controlar la velocidad a la que se acelera el juego: un valor de 2 duplicará la velocidad del juego cada vez que el jugador alcance un nuevo nivel; un valor de 1 mantendrá la velocidad constante. Un valor de velocidad como 1,1 debería aumentar la velocidad lo suficiente como para que el juego sea un reto, pero no imposible. Por último, llamamos a `initialize_dynamic_settings()` para inicializar los valores de los atributos que deben cambiar a lo largo de una partida ❷.

Aquí está el código para `initialize_dynamic_settings()`:

settings.py

```
    def inicializar_configuracion_dinamica(self):
```

```
        """Inicializar los ajustes que cambian a lo largo del juego."""
        self.ship_speed_factor = 1.5
        self.factor_velocidad_bala = 3
```

```
        self.factor_velocidad_extraterrestre = 1

        # fleet_direction de 1 representa la derecha; -1 representa la
        izquierda. self.fleet_direction = 1
```

Este método establece los valores iniciales para las velocidades de la nave, la bala y el alien. Aumentaremos estas velocidades a medida que el jugador progrese en el juego y las restableceremos cada vez que el jugador comience una nueva partida. Incluimos `fleet_direction` en este método para que los aliens siempre se muevan a la derecha al principio de una nueva partida. Para incrementar las velocidades de la nave, las balas y los aliens cada vez que el jugador alcanza un nuevo nivel, usa `increase_speed()`:

settings.py

```
    def aumentar_velocidad(self):
        """Aumentar los ajustes de velocidad."""
        factor_velocidad_propia_nave *=
        escala_velocidad_propia factor_velocidad_balas
        *= escala_velocidad_propia
        factor_velocidad_alien *=
        escala_velocidad_propia
```

Para aumentar la velocidad de estos elementos de juego, multiplicamos cada ajuste de velocidad por el valor de `speedup_scale`.

Aumentamos el ritmo del juego llamando a `increase_speed()` en `check_bullet_alien_collisions()` cuando el último alienígena de una flota ha sido derribado pero antes de crear una nueva flota:

juego_functions.py

```
def check_bullet_alien_collisions(ai_settings, screen, ship, aliens, bullets):
    --snip--
    si len(aliens) == 0:
        # Destruye las balas existentes, acelera el juego y crea una nueva
        flota. bullets.empty()
        ai_settings.increase_speed()
        crear_flota(ai_settings, pantalla, nave,
        aliens)
```

Basta con cambiar los valores de los ajustes de velocidad `factor_velocidad_nave`, `factor_velocidad_alienígena` y `factor_velocidad_bala` para acelerar todo el juego.

Restablecer la velocidad

Necesitamos devolver cualquier configuración cambiada a sus valores iniciales cada vez que el jugador comience una nueva partida, o cada nueva partida comenzaría con la configuración de velocidad aumentada de la partida anterior:

juego_funciones.py

```
def check_play_button(ai_settings, screen, stats, play_button, ship, aliens,
        balas, mouse_x, mouse_y):
    """Iniciar una nueva partida cuando el jugador haga clic en
    Jugar."" button_clicked = play_button.rect.collidepoint(mouse_x,
    mouse_y) if button_clicked and not stats.game_active:
```

```
        # Reinicia la configuración del juego.
        ai_settings.initialize_dynamic_settings()

        # Ocultar el cursor del ratón.
        pygame.mouse.set_visible(False)
        --snip--
```

Jugar a Alien Invasion debería ser ahora más divertido y desafiante. Cada vez que despejes la pantalla, el juego debería acelerarse y volverse ligeramente más difícil. Si el juego se vuelve demasiado difícil demasiado rápido, disminuye el valor de `settings.speedup_scale`, o si el juego no es lo suficientemente difícil, aumenta ligeramente el valor. Encuentra el punto óptimo aumentando la dificultad en un tiempo razonable. Las primeras pantallas deberían ser fáciles, las siguientes desafiantes pero factibles, y las siguientes casi imposiblemente difíciles.

TRY IT yOURSELF

14-3. Práctica de objetivo desafiante: Empieza con tu trabajo del Ejercicio 14-2 (página 298). Haz que la diana se mueva más rápido a medida que avanza el juego, y reinicia a la velocidad original cuando el jugador haga clic en Jugar.

puntuación

Implementemos un sistema de puntuación para seguir la puntuación del juego en tiempo real, así como para mostrar la puntuación más alta, el nivel y el número de naves restantes.

La puntuación es una estadística del juego, así que añadiremos un atributo de `puntuación` a `GameStats`:

game_stats.py

```
clase GameStats():
    --snip--
    def reset_stats(self):
        """Inicializar las estadísticas que pueden cambiar durante el juego."""
        self.ships_left = self.ai_settings.ship_limit
        auto.puntuación = 0
```

Para restablecer la puntuación cada vez que comienza una nueva partida, inicializamos la `puntuación` en `reset_stats()` en lugar de `init ()`.

Visualización de la puntuación

Para mostrar la puntuación en la `pantalla`, primero creamos una nueva clase, `Marcador`. Por ahora esta clase sólo mostrará la puntuación actual, pero la usaremos también para informar de la puntuación más alta, el nivel y el número de naves restantes. Aquí está la primera parte de la clase; guárdala como *marcador.py*:

marcador.py

```
importar pygame.font

clase Marcador():
    """Una clase para reportar información de puntuación."""
```

```
❶     def init (self, ai_settings, screen, stats):
          """Inicializar los atributos de
          puntuación."""  self.screen = screen
```

```
        self.screen_rect = screen.get_rect()
        self.ai_settings = ai_settings
        self.stats = stats

        # Ajustes de fuente para la información de puntuación.
❷       self.text_color = (30, 30, 30)
❸       self.font = pygame.font.SysFont(None, 48)

        # Preparar la imagen de puntuación inicial.
❹       self.prep_score()
```

Como `Scoreboard` escribe texto en la pantalla, empezamos importando el módulo `pygame.font`. Luego, le damos a `init()` los parámetros `ai_settings`, `screen`, y `stats` para que pueda reportar los valores que estamos rastreando ❶. A continuación, establecemos un color de texto ❷ e instanciamos un objeto fuente ❸.

Para convertir el texto a mostrar en una imagen, llamamos a `prep_score()` ❹, que definimos aquí:

marcador.py

```
    def prep_score(self):
        """Convierte la partitura en una imagen renderizada."""
❶       score_str = str(self.stats.score)
❷       self.score_image = self.font.render(score_str, True, self.text_color,
            self.ai_settings.bg_color)

        # Muestra la puntuación en la parte superior derecha de la pantalla.
❸       self.score_rect = self.score_image.get_rect()
❹       auto.puntuación_derecha.derecha = auto.pantalla_derecha.derecha - 20
❺       auto.puntuación_recta.top = 20
```

En `prep_score()`, primero convertimos el valor numérico `stats.score` en una cadena ❶, y luego pasamos esta cadena a `render()`, que crea la imagen ❷. Para mostrar la puntuación claramente en pantalla, pasamos el color de fondo de la pantalla a `render()`, así como un color de texto.

Situaremos la puntuación en la esquina superior derecha de la pantalla y haremos que se expanda hacia la izquierda a medida que aumente la puntuación y crezca la anchura del número. Para asegurarnos de que la puntuación siempre se alinea con el lado derecho de la pantalla, creamos un `rectángulo` llamado `score_rect` ❸ y establecemos su borde derecho a 20 píxeles del borde derecho de la pantalla ❹. A continuación, colocamos el borde superior 20 píxeles hacia abajo desde la parte superior de la pantalla ❺.

Por último, creamos un método `show_score()` para mostrar la imagen renderizada de la puntuación:

marcador.py

```
    def mostrar_marcador(self):
        """Dibuja la puntuación en la pantalla."""
        self.screen.blit(self.score_image, self.score_rect)
```

Este método dibuja la imagen de puntuación en la pantalla en el lugar especificado por `score_rect`.

Creación de un marcador

Para mostrar la puntuación, crearemos una instancia de `Marcador` en *alien_invasion.py*:

alien_invasión.py

```
--snip--
from game_stats import GameStats
from marcador import Marcador
--snip--
def ejecutar_juego():
    --snip--
    # Crear una instancia para almacenar las estadísticas del juego y crear
    un marcador. stats = GameStats(ai_settings)
❶   sb = Marcador(ai_configuración, pantalla, estadísticas)
    --snip--
    # Inicia el bucle principal del juego.
    while True:
        --snip--
❷       gf.update_screen(ai_settings, screen, stats, sb, ship, aliens,
            bullets, play_button)

run_game()
```

Importamos la nueva clase `Marcador` y creamos una instancia llamada `sb` después de crear la instancia `stats` ❶. Luego pasamos `sb` a `update_screen()` para que la puntuación se dibuje en la pantalla ❷.

Para mostrar la puntuación, modifica `update_screen()` así:

juego_funciones.py

```
def update_screen(ai_settings, screen, stats, sb, ship, aliens, bullets,
        play_button):
    --snip--
    # Dibuja la información de la
    puntuación. sb.show_score()

    # Dibuja el botón de jugar si el juego está
    inactivo. if not stats.game_active:
        play_button.draw_button()

    # Hacer visible la pantalla dibujada más recientemente.
    pygame.display.flip()
```

Añadimos `sb` a la lista de parámetros que definen `update_screen()` y llamamos a `show_score()` justo antes de que se dibuje el botón Reproducir.

Cuando ejecutes Alien Invasion ahora, deberías ver 0 en la parte superior derecha de la pantalla. (Por ahora sólo queremos asegurarnos de que la puntuación aparece en el lugar correcto antes de desarrollar más el sistema de puntuación). La Figura 14-2 muestra la puntuación tal y como aparece antes de que comience el juego.

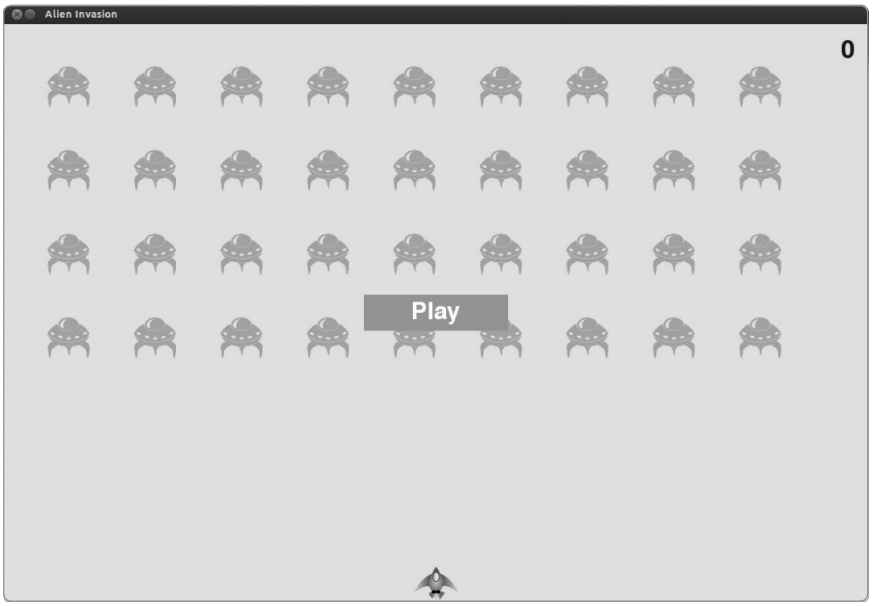


Figura 14-2: La puntuación aparece en la esquina superior derecha de la pantalla.

Ahora, ¡a asignar valores de puntos a cada alienígena!

Actualizar el marcador a medida que se derriban alienígenas

Para escribir una puntuación en directo en la pantalla, actualizamos el valor de `stats.score` cada vez que se derriba a un alien, y luego llamamos a `prep_score()` para actualizar la imagen de la puntuación. Pero primero, vamos a determinar cuántos puntos obtiene un jugador cada vez que derriba a un alienígena:

settings.py

```
    def inicializar_configuracion_dinamica(self):
        --snip--

        # Puntuación
        self.alien_points = 50
```

Aumentaremos el valor en puntos de cada alienígena a medida que avance la partida. Para asegurarnos de que este valor en puntos se restablece cada vez que comienza una nueva partida, establecemos el valor en `initialize_dynamic_settings()`.

Actualiza la puntuación cada vez que un alien es derribado en `check_bullet_alien_ collisions()`:

juego_funciones.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    """Responder a colisiones bala-alien."""
    # Elimina las balas y los alienígenas que hayan colisionado.
    colisiones = pygame.sprite.groupcollide(balas, alienígenas, Verdadero,
    Verdadero)
```

```
    si colisiones:
❶       stats.score += ai_settings.alien_points
        sb.prep_score()
    --snip--
```

Actualizamos la definición de `check_bullet_alien_collisions()` para incluir los parámetros `stats` y `sb` para que pueda actualizar la puntuación y el marcador. Cuando una bala golpea a un alien, Pygame devuelve un diccionario de `colisiones`. Comprobamos si el diccionario existe, y si es así, el valor del alienígena se añade a la puntuación ❶. A continuación, llamamos a `prep_score()` para crear una nueva imagen para la puntuación actualizada.

Tenemos que modificar `update_bullets()` para asegurarnos de que se pasan los argumentos adecuados entre las funciones:

juego_functions.py

```
def update_bullets(ai_settings, screen, stats, sb, ship, aliens, bullets):
    """Actualizar posición de viñetas y deshacerse de viñetas antiguas."""
    --snip--

    check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets)
```

La definición de `update_bullets()` necesita los parámetros adicionales `stats` y `sb`. La llamada a `check_bullet_alien_collisions()` necesita incluir también los argumentos `stats` y `sb`.

También tenemos que modificar la llamada a `update_bullets()` en la aplicación principal
bucle `while`:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, stats, play_button, ship,
            aliens, bullets)
        if stats.game_active:
            nave.actualizar()
            gf.update_bullets(ai_settings, screen, stats, sb, ship, aliens,
                bullets)
            --snip--
```

La llamada a `update_bullets()` necesita los argumentos `stats` y `sb`.

Ahora, cuando juegues a Alien Invasion, ¡podrás acumular puntos!

Asegurarse de anotar todos los golpes

Tal y como está escrito actualmente, nuestro código podría pasar por alto algunos alienígenas. Por ejemplo, si dos balas colisionan con alienígenas durante la misma pasada por el bucle o si hacemos una bala extra ancha para golpear a múltiples alienígenas, el jugador recibirá puntos sólo por uno de los alienígenas matados. Para solucionar esto, vamos a refinar la forma en que se detectan las colisiones de balas alienígenas.

En `check_bullet_alien_collisions()`, cualquier bala que colisiona con un alienígena se convierte en una clave del diccionario de `colisiones`. El valor asociado a cada bala es una lista de alienígenas con los que ha colisionado. Recorremos el diccionario de `colisiones` para asegurarnos de que otorgamos puntos por cada alienígena alcanzado:

juego_funciones.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    --snip--
    si colisiones:
1       para extranjeros en colisiones.valores():
            stats.score += ai_settings.alien_points * len(aliens)
            sb.prep_score()
    --snip--
```

Si se ha definido el diccionario de `colisiones`, hacemos un bucle a través de todos los valores del diccionario de `colisiones`. Recuerde que cada valor es una lista de alienígenas alcanzados por una sola bala. Multiplicamos el valor de cada alienígena por el número de alienígenas de cada lista y añadimos esta cantidad a la puntuación actual. Para probar esto, cambie el ancho de una bala a 300 píxeles y verifique que recibe puntos por cada alienígena que golpea con sus balas extra anchas; luego devuelva el ancho de la bala a normal.

Aumento del valor de los puntos

Dado que el juego se vuelve más difícil cada vez que un jugador alcanza un nuevo nivel, los alienígenas de los niveles posteriores deberían valer más puntos. Para implementar esta funcionalidad, añadiremos código para aumentar el valor de los puntos cuando aumente la velocidad del juego:

settings.py

```
clase Ajustes():
    """Una clase para almacenar todos los ajustes de Alien Invasion."""

    def init (self):
        --snip--
        # Cómo de rápido se acelera el juego
        self.speedup_scale = 1.1
        # La rapidez con la que aumentan los valores de los puntos extraños
1       self.escala_puntuación = 1.5

        self.initialize_dynamic_settings()

    def aumentar_velocidad(self):
        """Aumentar los valores de velocidad y puntos alienígenas"""
        self.factor_velocidad_nave *= self.escala_velocidad
        self.factor_velocidad_bala *= self.escala_velocidad
        self.factor_velocidad_alien *= self.escala_velocidad

2       auto.puntos_alien = int(auto.puntos_alien * auto.escala_puntuación)
```

Definimos una velocidad a la que aumentan los puntos, que llamamos

`score_scale` ❶. Un pequeño aumento de la velocidad (1,1) hace que el juego crezca desafiante rápidamente,

pero para ver una diferencia notable en la puntuación hay que cambiar el valor en puntos del alienígena en una cantidad mayor (1,5). Ahora, cuando aumentamos la velocidad del juego, también aumentamos el valor en puntos de cada golpe ❷. Usamos la función `int()` para aumentar el valor de puntos en enteros.

Para ver el valor de cada extranjero, añada una sentencia `print` al método `increase_speed()` en `Ajustes`:

settings.py

```
    def aumentar_velocidad(self):
        --snip--
        auto.puntos_alien = int(auto.puntos_alien * auto.escala_puntuación)
        print(auto.puntos_alien)
```

Deberías ver el nuevo valor de puntos en el terminal cada vez que alcances un nuevo nivel.

NOTE *Asegúrese de eliminar la instrucción de impresión después de verificar que el valor del punto está aumentando, o puede afectar al rendimiento de su juego y distraer al jugador.*

Redondear el marcador

La mayoría de los juegos de disparos de estilo arcade informan de las puntuaciones como múltiplos de 10, así que vamos a seguir ese ejemplo con nuestra puntuación. También vamos a formatear la puntuación para incluir separadores de comas en los números grandes. Haremos este cambio en el `Marcador`:

marcador.py

```
    def prep_score(self):
        """Convierte la partitura en una imagen renderizada."""
❶       rounded_score = int(round(self.stats.score, -1))
❷       score_str = "{:,}".format(rounded_score)
        self.score_image = self.font.render(score_str, True, self.text_color,
            self.ai_settings.bg_color)
        --snip--
```

La función `round()` redondea normalmente un número decimal a un número determinado de decimales dado como segundo `argumento`. Sin embargo, si pasa un número negativo como segundo `argumento`, `round()` redondeará el valor a
el 10, 100, 1000 más cercano, y así sucesivamente. El código en ❶ le dice a Python que redondee el valor de `stats.score` al 10 más cercano y lo almacene en `rounded_score`.

NOTA *En Python 2.7, round() siempre devuelve un valor decimal, así que usamos int() para asegurarnos de que la puntuación se reporta como un entero. Si utiliza Python 3, puede omitir la llamada a int().*

En ❷, una directiva de formato de cadena le dice a Python que inserte comas en los números cuando convierta un valor numérico en una cadena- por ejemplo, para mostrar `1,000,000` en lugar de `1000000`. Ahora, cuando ejecute el juego, debería ver una puntuación redondeada y con un formato limpio, incluso cuando acumule muchos puntos, como se muestra en la Figura 14-3.



Figura 14-3: Puntuación redondeada con separadores de comas

Puntuaciones altas

Todos los jugadores quieren batir la puntuación máxima de un juego, así que vamos a registrar e informar de las puntuaciones máximas para dar a los jugadores algo por lo que esforzarse. Almacenaremos las puntuaciones en `GameStats`:

game_stats.py

```
    def init (self, ai_settings):
        --snip--
        # La puntuación alta nunca debe
        reiniciarse. self.high_score = 0
```

Dado que la puntuación máxima no debe restablecerse nunca, inicializamos `high_score` en `init ()` en lugar de en `reset_stats()`.

Ahora modificaremos `Scoreboard` para que muestre la puntuación más alta. Empecemos con el método `init ()`:

marcador.py

```
    def init (self, ai_settings, screen, stats):
        --snip--
        # Prepara las imágenes de puntuación
        inicial. self.prep_score()
❶       self.prep_high_score()
```

La puntuación más alta se mostrará separada de la puntuación, por lo que necesitamos un nuevo método, `prep_high_score()`, para preparar la imagen de la puntuación más alta ❶.

Este es el método `prep_high_score()`:

marcador.py

```
    def prep_high_score(self):
        """Convierte la puntuación más alta en una imagen renderizada."""
❶       puntuación_alta = int(round(self.stats.puntuación_alta, -1))
```

```
❷         puntuación_alta_str = "{:,}".format(puntuación_alta)
❸         self.high_score_image = self.font.render(high_score_str, True,
              self.text_color, self.ai_settings.bg_color)

          # Centra la puntuación más alta en la parte superior de la pantalla.
          self.high_score_rect = self.high_score_image.get_rect()
❹         auto.recto_puntuacion_alta.centerx = auto.recto_pantalla.centerx
❺         auto.recto_puntuacion_alta.top = auto.recto_puntuacion.top
```

Redondeamos la puntuación más alta al 10 más cercano ❶ y le damos formato con comas ❷. A continuación, generamos una imagen a partir de la puntuación máxima ❸, centramos horizontalmente el `rectángulo de` puntuación máxima ❹ y establecemos su atributo `superior` para que coincida con la parte superior de la imagen de puntuación ❺.

El método `show_score()` ahora dibuja la puntuación actual en la parte superior derecha y la puntuación más alta en la parte superior central de la pantalla:

marcador.py

```
    def mostrar_marcador(self):
        """Dibuja la puntuación en la pantalla."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
```

Para comprobar las puntuaciones altas, escribiremos una nueva función, `check_high_score()`, en *game_functions.py*:

juego_functions.py

```
def comprobar_puntuación_alta(estadísticas, sb):
    """Comprueba si hay una nueva puntuación máxima."""
❶   if stats.score > stats.high_score:
        stats.high_score = stats.score
        sb.prep_high_score()
```

La función `check_high_score()` toma dos parámetros, `stats` y `sb`. Utiliza `stats` para comprobar la puntuación actual y la puntuación máxima, y necesita `sb` para modificar la imagen de la puntuación máxima cuando sea necesario. En ❶ comprobamos la puntuación actual comparándola con la puntuación alta. Si la puntuación actual es mayor, actualizamos el valor de `high_score` y llamamos a `prep_high_score()` para actualizar la imagen de la puntuación alta.

Necesitamos llamar a `check_high_score()` cada vez que un alien es golpeado después de actualizar la puntuación en `check_bullet_alien_collisions()`:

juego_funciones.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    --snip--
    si colisiones:
        para extranjeros en colisiones.valores():
            stats.score += ai_settings.alien_points * len(aliens)
```

```
            sb.prep_score()
        comprobar_puntuación_alta(stats, sb)
    --snip--
```

Llamamos a `check_high_score()` cuando el diccionario de `colisiones` está presente, y lo hacemos después de actualizar la puntuación de todos los alienígenas que han sido golpeados.

La primera vez que juegues a Invasión Alienígena tu puntuación será la más alta, por lo que se mostrará tanto la puntuación actual como la más alta. Pero cuando empieces una segunda partida, tu puntuación alta debería aparecer en el centro y tu puntuación actual a la d e r e c h a , como se muestra en la Figura 14-4.



Figura 14-4: La puntuación más alta se muestra en la parte superior central de la pantalla.

Visualización del nivel

Para mostrar el nivel del jugador en el juego, primero necesitamos un atributo en `GameStats` que represente el nivel actual. Para restablecer el nivel al comienzo de cada nueva partida, inicialízalo en `reset_stats()`:

game_stats.py

```
    def reset_stats(self):
        """Inicializar las estadísticas que pueden cambiar durante el juego."""
        self.ships_left = self.ai_settings.ship_limit
        auto.puntuación = 0
        auto.nivel = 1
```

Para que `el Marcador` muestre el nivel actual (justo debajo de la puntuación actual), llamamos a un nuevo método, `prep_level()`, desde `init ()`:

marcador.py

```
    def init (self, ai_settings, screen, stats):
        --snip--

        # Prepara las imágenes de puntuación
        inicial. self.prep_score()
        self.prep_high_score()
        self.prep_level()
```

Aquí está `prep_level()`:

marcador.py

```
    def prep_level(self):
        """Convierte el nivel en una imagen renderizada."""
❶       self.level_image = self.font.render(str(self.stats.level), True,
                self.text_color, self.ai_settings.bg_color)

        # Posiciona el nivel debajo del marcador.
        self.level_rect = self.level_image.get_rect()
❷       auto.recto_nivel.derecha = auto.recto_puntuación.derecha
❸       auto.nivel_rect.top = auto.puntuación_rect.bottom + 10
```

El método `prep_level()` crea una imagen a partir del valor almacenado en `stats.level` ❶ y establece el atributo `derecho` de la imagen para que coincida con el atributo `derecho` de la puntuación ❷. A continuación, establece el atributo `superior` 10 píxeles por debajo de la parte inferior de la imagen de puntuación para dejar espacio entre la puntuación y el nivel ❸.

También tenemos que actualizar `show_score()`:

marcador.py

```
    def mostrar_marcador(self):
        """Dibuja puntuaciones y barcos en la pantalla."""
        self.screen.blit(self.score_image, self.score_rect)
        self.screen.blit(self.high_score_image, self.high_score_rect)
        self.screen.blit(self.level_image, self.level_rect)
```

Esto añade una línea para dibujar la imagen del nivel en la pantalla.

Incrementaremos `stats.level` y actualizaremos la imagen de nivel en `check_bullet_ alien_collisions()`:

juego_funciones.py

```
def check_bullet_alien_collisions(ai_settings, screen, stats, sb, ship,
        aliens, bullets):
    --snip--
    si len(aliens) == 0:
        # Si toda la flota es destruida, comienza un nuevo nivel.
        bullets.empty()
        ai_settings.aumentar_velocidad()

      # Aumenta el nivel.
❶       stats.level += 1
❷       sb.prep_level()

        create_fleet(ai_settings, screen, ship, aliens)
```

Si una flota es destruida, incrementamos el valor de `stats.level` ❶ y llamamos a `prep_level()` para asegurarse de que el nuevo nivel se muestra correctamente ❷.

Para asegurarte de que la puntuación y las imágenes de los niveles se actualizan correctamente al comienzo de una nueva partida, activa un reinicio al pulsar el botón Jugar:

juego_funciones.py

```
def check_play_button(ai_settings, screen, stats, sb, play_button, ship,
        aliens, bullets, mouse_x, mouse_y):
    """Iniciar una nueva partida cuando el jugador haga clic en
    Jugar."" button_clicked = play_button.rect.collidepoint(mouse_x,
    mouse_y) if button_clicked and not stats.game_active:
```

```
        --snip--

        # Reinicia las estadísticas del
        juego. stats.reset_stats()
        stats.game_active = True

        # Restablecer las imágenes del marcador.
❶       sb.prep_score()
        sb.prep_high_score()
        sb.prep_level()

        # Vacía la lista de alienígenas y balas.
        aliens.empty()
        balas.empty()

        --snip--
```

La definición de `check_play_button()` necesita el objeto `sb`. Para restablecer las imágenes del marcador, llamamos a `prep_score()`, `prep_high_score()`, y `prep_level()` después de restablecer los ajustes ❶ relevantes del juego.

Ahora pasa `sb` desde `check_events()` para que `check_play_button()` tenga acceso al objeto marcador:

juego_funciones.py

```
def check_events(ai_settings, screen, stats, sb, play_button, ship, aliens,
        balas):
    """Responder a eventos de teclado y ratón."""
    for event in pygame.event.get():
        if evento.type == pygame.QUIT:
            --snip--
        elif event.type == pygame.MOUSEBUTTONDOWN:
            mouse_x, mouse_y = pygame.mouse.get_pos()
❶           check_play_button(ai_settings, pantalla, estadísticas, sb,
                play_button, nave, aliens, balas, mouse_x, mouse_y)
```

La definición de `check_events()` necesita `sb` como parámetro, por lo que la llamada a `check_play_button()` puede incluir `sb` como argumento ❶.

Por último, actualiza la llamada a `check_events()` en *alien_invasion.py* para que pase `sb` también:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        gf.check_events(ai_settings, screen, stats, sb, play_button, ship,
            aliens, bullets)
        --snip--
```

Ahora puedes ver cuántos niveles has completado, como se muestra en la Figura 14-5.

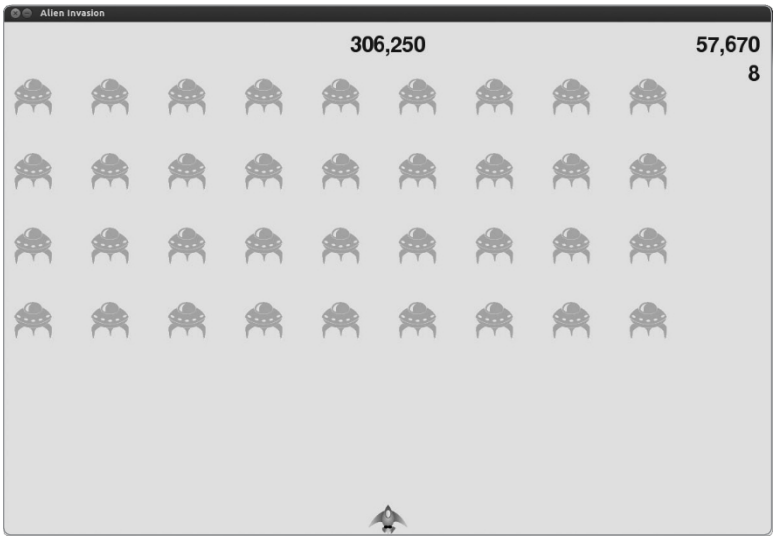


Figura 14-5: El nivel actual se informa justo debajo de la puntuación actual.

NOTA *En algunos juegos clásicos, las puntuaciones tienen etiquetas, como Puntuación, Puntuación máxima y Nivel. Hemos omitido estas etiquetas porque el significado de cada número queda claro una vez que has jugado al juego. Para incluir estas etiquetas, añádelas a las cadenas de puntuación justo antes de las llamadas a font.render() en Marcador.*

Visualización del número de buques

Por último, vamos a mostrar el número de naves que le quedan al jugador, pero esta vez utilizaremos un gráfico. Para ello, dibujaremos naves en la esquina superior izquierda de la pantalla para representar cuántas naves quedan, como hacen muchos juegos arcade clásicos.

Primero, tenemos que hacer que `Ship` herede de `Sprite` para poder crear un grupo de naves:

barco.py

```
importar pygame
from pygame.sprite import Sprite

❶ clase Barco(Sprite):

    def init (self, ai_settings, screen):
        """Inicializa la nave y fija su posición inicial."""
❷       super(Nave, self). init ()
        --snip--
```

Aquí importamos `Sprite`, nos aseguramos de que `Ship` hereda de `Sprite` ❶, y llamamos a `super()` al principio de `init ()` ❷.

A continuación, necesitamos modificar `Scoreboard` para crear un grupo de barcos que podamos mostrar. Aquí están las sentencias `import` e `init ()`:

marcador.py

```
importar pygame.font
from pygame.sprite import Grupo

from ship import Barco

class Marcador():
    """Una clase para reportar información de puntuación."""

    def init (self, ai_settings, screen, stats):
        --snip-- self.prep_level()
        self.prep_ships()
       --snip--
```

Como estamos haciendo un grupo de barcos, importamos el `Grupo` y el `Barco` clases. Llamamos a `prep_ships()` después de la llamada a `prep_level()`.

Aquí está `prep_ships()`:

marcador.py

```
    def prep_ships(self):
        """Muestra cuántos barcos quedan."""
❶       self.ships = Grupo()
❷       for número_nave in
            rango(auto.estadísticas.naves_izquierda):
            nave = Nave(auto.ai_configuración,
            auto.pantalla)
❸           ship.rect.x = 10 + ship_number * ship.rect.width
❹           barco.rect.y = 10
❺           self.ships.add(barco)
```

El método `prep_ships()` crea un grupo vacío, `self.ships`, para contener las instancias de naves ❶. Para llenar este grupo, un bucle se ejecuta una vez por cada nave que el jugador ha dejado ❷. Dentro del bucle creamos una nueva nave y establecemos el valor de la coordenada x de cada nave para que las naves aparezcan una al lado de la otra con un margen de 10 píxeles a la izquierda del grupo de barcos ❸. Fijamos el valor de la coordenada y 10 píxeles hacia abajo desde la parte superior de la pantalla para que los barcos se alineen con la imagen de puntuación ❹. Por último, añadimos cada nuevo barco al grupo de `barcos` ❺.

Ahora tenemos que dibujar los barcos en la pantalla:

marcador.py

```
    def mostrar_marcador(self):
        --snip--
        self.screen.blit(self.level_image, self.level_rect)
        # Dibujar barcos.
        self.ships.draw(self.screen)
```

Para mostrar las naves en la pantalla, llamamos a `draw()` sobre el grupo, y

Pygame dibuja cada nave.

Para mostrar al jugador cuántas naves tiene para empezar, llamamos a `prep_ships()` cuando empieza una nueva partida. Hacemos esto en `check_play_button()` en *game_functions.py*:

juego_funciones.py

```
def check_play_button(ai_settings, screen, stats, sb, play_button, ship,
        aliens, bullets, mouse_x, mouse_y):
    """Iniciar una nueva partida cuando el jugador haga clic en
    Jugar."" button_clicked = play_button.rect.collidepoint(mouse_x,
    mouse_y) if button_clicked and not stats.game_active:
        --snip--
        # Restablecer las imágenes
        del marcador. sb.prep_score()
        sb.prep_high_score()
        sb.prep_level()
        sb.prep_ships()
        --snip--
```

También llamamos a `prep_ships()` cuando un barco es golpeado para actualizar la visualización de las imágenes de los barcos cuando el jugador pierde un barco:

game_functions.py

```
❶ def update_aliens(ai_settings, screen, stats, sb, ship, aliens, bullets):
        --snip--
        # Busca colisiones entre naves alienígenas.
        si pygame.sprite.spritecollideany(nave, aliens):
❷           ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets)

        # Busca alienígenas golpeando la parte inferior de la pantalla.
❸       check_aliens_bottom(ai_settings, pantalla, estadísticas, sb, nave, aliens,
balas)

❹ def ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets):
        """Responde a la nave que es golpeada por un alien""".
        si stats.ships_left > 0:
            # Disminuye naves_izquierda.
            stats.naves_izquierda -= 1

            # Actualizar marcador.
❺           sb.prep_ships()

            # Vacía la lista de alienígenas y balas.
            --snip--
```

Primero añadimos el parámetro `sb` a la definición de `update_aliens()` ❶. Luego pasamos `sb` a `ship_hit()` ❷ y `check_aliens_bottom()` para que cada uno tenga acceso al objeto marcador ❸.

A continuación, actualizamos la definición de `ship_hit()` para incluir `sb` ❹. Llamamos a `prep_ships()` después de reducir el valor de `ships_left` ❺, para que se muestre el número correcto de barcos cada vez que se destruye un barco.

Hay una llamada a `ship_hit()` en `check_aliens_bottom()`, así que actualiza esa función también:

juego_funciones.py

```
def check_aliens_bottom(ai_settings, screen, stats, sb, ship, aliens,
        balas):
    """Comprueba si algún alienígena ha llegado al fondo de la pantalla."""
    screen_rect = screen.get_rect()
    para alien en aliens.sprites():
        si alien.rect.bottom >= screen_rect.bottom:
            # Trata esto igual que si una nave fuera alcanzada.
            ship_hit(ai_settings, screen, stats, sb, ship, aliens, bullets)
            break
```

Ahora `check_aliens_bottom()` acepta `sb` como parámetro, y añadimos un `sb` en la llamada a `ship_hit()`.

Por último, pasa `sb` en la llamada a `update_aliens()` en *alien_invasion.py*:

alien_invasion.py

```
    # Inicia el bucle principal del juego.
    mientras True:
        --snip--
        if stats.game_active:
            nave.actualizar()
            gf.update_bullets(ai_settings, screen, stats, sb, ship, aliens,
                bullets)
            gf.update_aliens(ai_settings, screen, stats, sb, ship, aliens,
                bullets)
            --snip--
```

La Figura 14-6 muestra el sistema de puntuación completo con los barcos restantes mostrados en la parte superior izquierda de la pantalla.

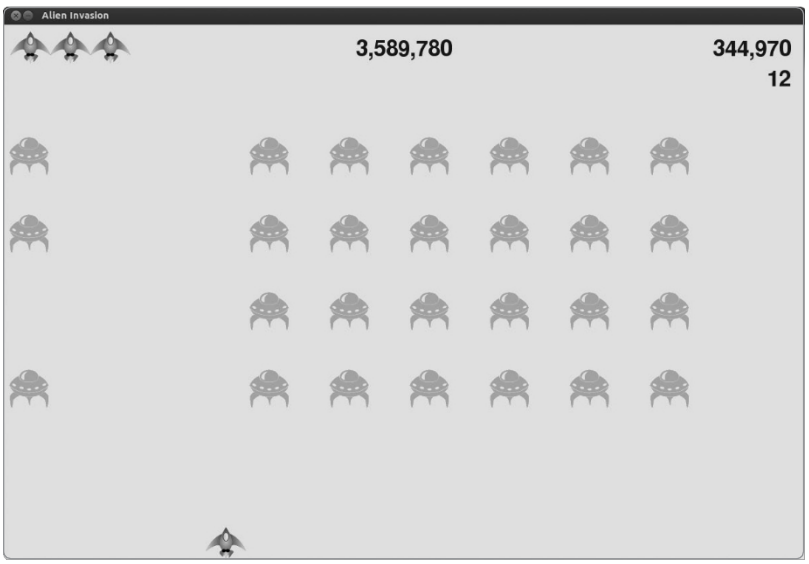


Figura 14-6: El sistema de puntuación completo de Alien Invasion

TRY IT yOURSELF

14-4. Máxima puntuación histórica: La puntuación máxima se reinicia cada vez que un jugador cierra y reinicia Alien Invasion. Corrige esto escribiendo la puntuación máxima en un archivo antes de llamar a `sys.exit()` y leyendo la puntuación máxima al inicializar su valor en `GameStats`.

14-5. Refactorización: Busque funciones y métodos que estén haciendo más de una tarea, y refactorícelos para mantener su código organizado y eficiente. Por ejemplo, mueva parte del código en `check_bullet_alien_collisions()`, que inicia un nuevo nivel cuando la flota de alienígenas ha sido destruida, a una función llamada `start_new_level()`. Además, mueva las cuatro llamadas de método separadas en el método `init ()` en `Scoreboard` a un método llamado prep_images() para acortarinit (). El método `prep_images()` también podría ayudar a `check_play_button()` o `start_game()` si ya has refactorizado `check_play_button()`.

NOTA *Antes de intentar refactorizar el proyecto, consulte el Apéndice D para aprender a restaurar el proyecto a un estado de trabajo si introduce errores durante la refactorización.*

14-6. Ampliar la invasión alienígena: Piensa en una forma de expandir la Invasión Alienígena. Por ejemplo, podrías programar a los alienígenas para que disparen balas a la nave o añadir escudos para que tu nave se esconda detrás, que puedan ser destruidos por balas desde cualquier lado. O utilizar algo como el módulo `pygame.mixer` para añadir efectos de sonido como explosiones y sonidos de disparos.

resumen

En este capítulo has aprendido a crear un botón Jugar para iniciar una nueva partida y a detectar eventos del ratón y ocultar el cursor en partidas activas. Puedes utilizar lo que has aprendido para crear otros botones en tus juegos, como un botón Ayuda para mostrar instrucciones sobre cómo jugar. También has aprendido a modificar la velocidad de un juego a medida que avanza, a implementar un sistema de puntuación progresivo y a mostrar información de forma textual y no textual.

PROYECTO 2

D A T A V I S U A L I Z A C I Ó N

15

GENERATING DATA



La visualización de datos consiste en explorar los datos mediante representaciones visuales. Está estrechamente asociada con la *minería de datos*, que utiliza código para explorar los patrones y conexiones de un conjunto de datos. Un conjunto de datos puede ser solo una pequeña lista de números que cabe en una línea de código o muchos gigabytes de datos.

Representar los datos de forma atractiva es mucho más que hacer fotos bonitas. Cuando se tiene una representación sencilla y visualmente atractiva de un conjunto de datos, su significado queda claro para los espectadores. La gente verá patrones y significados en sus conjuntos de datos que no sabían que existían.

Afortunadamente, no hace falta un superordenador para visualizar datos complejos. Con la eficacia de Python, puedes explorar rápidamente conjuntos de datos formados por millones de puntos de datos individuales en un simple ordenador portátil. Los puntos de datos tampoco tienen por qué ser números. Con los conceptos básicos que aprendiste en la primera parte de este libro, también puedes analizar datos no numéricos.

Python se utiliza para trabajos intensivos en genética, investigación climática, análisis político y económico, y muchos otros. Los científicos de datos han c r e a d o una impresionante variedad de herramientas de visualización y análisis en Python, muchas de las cuales se utilizan en la actualidad.

que también están a tu disposición. Una de las herramientas más populares es matplotlib, una biblioteca de gráficos matemáticos. Utilizaremos matplotlib para hacer trazados sencillos, como gráficos de líneas y de dispersión. Después, crearemos un conjunto de datos más interesante basado en el concepto de paseo aleatorio: una visualización generada a partir de una serie de decisiones aleatorias.

También utilizaremos un paquete llamado Pygal, que se centra en crear visualizaciones que funcionen bien en dispositivos digitales. Puedes usar Pygal para enfatizar y redimensionar elementos a medida que el usuario interactúa con tu visualización, y puedes redimensionar fácilmente toda la representación para que quepa en un pequeño smartwatch o en un monitor gigante. Utilizaremos Pygal para explorar qué ocurre cuando tiras los dados de varias maneras.

Instalación de matplotlib

Primero, necesitarás instalar matplotlib, que usaremos para nuestro conjunto inicial de visualizaciones. Si aún no has utilizado pip, consulta "Instalar paquetes Python con pip" en la página 237.

En Linux

Si estás usando la versión de Python que viene con tu sistema, puedes usar el gestor de paquetes de tu sistema para instalar matplotlib usando sólo una línea:

```
$ sudo apt-get install python3-matplotlib
```

Si está ejecutando Python 2.7, utilice esta línea:

```
$ sudo apt-get install python-matplotlib
```

Si has instalado una versión más reciente de Python, tendrás que instalar algunas bibliotecas de las que depende matplotlib:

```
$ sudo apt-get install python3.5-dev python3.5-tk tk-dev
$ sudo apt-get install libfreetype6-dev g++
```

A continuación, utilice pip para instalar matplotlib:

```
$ pip install --user matplotlib
```

En OS X

Apple incluye matplotlib con su instalación estándar de Python. Para comprobar si está instalado en tu sistema, abre una sesión de terminal e intenta **importar matplotlib**. Si matplotlib no está ya en tu sistema y usaste Homebrew para instalar Python, instálalo así:

```
$ pip install --user matplotlib
```

NOTE *Puede que necesite usar* `pip3` *en lugar de* `pip` *cuando instale paquetes. Además, si este comando no funciona, es posible que deba omitir el indicador* `--user`.

En Windows

En Windows, primero tendrás que instalar Visual Studio. Vaya a *https://dev.windows.com/*, haz clic en **Descargas** y busca Visual Studio Community, que es un conjunto gratuito de herramientas de desarrollo para Windows. Descarga y ejecuta el instalador.

A continuación necesitarás un instalador para matplotlib. Ve a *https://pypi.python.org/ pypi/matplotlib/* y busca un archivo wheel (un archivo que termine en *.whl*) que coincida con la versión de Python que estés utilizando. Por ejemplo, si está utilizando una versión de 32 bits de Python 3.5, tendrá que descargar *matplotlib-1.4.3-cp35-none-win32. whl*.

NOTE *Si no ves un archivo que coincida con tu versión instalada de Python, mira lo que hay disponible en* http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib. *Este sitio suele publicar los instaladores un poco antes que el sitio oficial de matplotlib.*

Copie el archivo *.whl* en la carpeta del proyecto, abra una ventana de comandos y vaya a la carpeta del proyecto. A continuación, utilice pip para instalar matplotlib:

```
> cd python_work
python_work> python -m pip install --user matplotlib-1.4.3-cp35-none-win32.whl
```

Probando matplotlib

Una vez instalados los paquetes necesarios, pruebe la instalación iniciando una sesión de terminal con el comando **python** o **python3** e importando matplotlib:

```
$ python3
>>> import matplotlib
>>>
```

Si no ves ningún mensaje de error, entonces matplotlib está instalado en tu sistema, y puedes pasar a la siguiente sección.

NOTE *Si tiene problemas con la instalación, consulte el Apéndice C. Si todo lo demás falla, pida ayuda. Lo más probable es que un programador experimentado de Python pueda resolver su problema rápidamente con un poco de información de su parte.*

Galería matplotlib

Para ver los tipos de visualizaciones que puedes hacer con matplotlib, visita la

galería de ejemplos en http://matplotlib.org/. Cuando hagas clic en una visualización de la galería, podrás ver el código utilizado para generar el gráfico.

Trazado de una gráfica lineal simple

Vamos a trazar un simple gráfico de líneas utilizando matplotlib, y luego personalizarlo para crear una visualización más informativa de nuestros datos. Utilizaremos la secuencia de números cuadrados 1, 4, 9, 16, 25 como datos para el gráfico.

Simplemente proporcione a matplotlib los números como se muestra aquí, y matplotlib hará el resto:

mpl_cuadrados.py

```
importar matplotlib.pyplot como plt

cuadrados = [1, 4, 9, 16,
25] plt.plot(cuadrados)
plt.show()
```

Primero importamos el módulo `pyplot` usando el alias `plt` para no tener que escribir `pyplot` repetidamente. (`pyplot` contiene una serie de funciones que ayudan a generar gráficos y diagramas.

Creamos una lista para guardar los cuadrados y luego la pasamos a la función `plot()`, que intentará trazar los números de una manera significativa. `plt.show()` abre el visor de matplotlib y muestra el gráfico, como se muestra en la Figura 15-1. El visor permite hacer zoom y navegar por el gráfico. El visor le permite hacer zoom y navegar por el gráfico, y si hace clic en el icono del disco, puede guardar las imágenes del gráfico que desee.

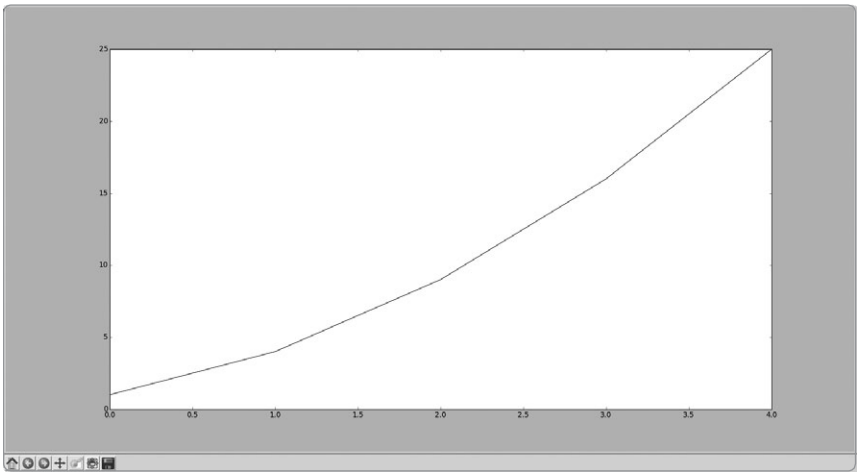


Figura 15-1: Uno de los gráficos más sencillos que puedes hacer en matplotlib

Cambiar el tipo de etiqueta y el grosor del gráfico

Aunque el gráfico de la Figura 15-1 muestra que los números aumentan, el tipo de etiqueta es demasiado pequeño y la línea demasiado fina. Afortunadamente, matplotlib permite ajustar todas las características de una visualización.

Utilizaremos algunas de las personalizaciones disponibles para mejorar la legibilidad de este gráfico, como se muestra aquí:

mpl_cuadrados.py

```
import matplotlib.pyplot como plt

cuadrados = [1, 4, 9, 16, 25]
❶ plt.plot(squares, linewidth=5)

# Establece el título del gráfico y etiqueta los ejes.
❷ plt.title("Números cuadrados", fontsize=24)
❸ plt.xlabel("Valor", fontsize=14)
plt.ylabel("Cuadrado del valor",
fontsize=14)

# Establecer el tamaño de las etiquetas de tick.
❹ plt.tick_params(axis='both', labelize=14)

plt.show()
```

El parámetro `linewidth` en ❶ controla el grosor de la línea que genera `plot()`. La función `title()` en ❷ establece un título para el gráfico. Los parámetros `fontsize`, que aparecen repetidamente a lo largo del código, controlan el tamaño del texto del gráfico.

Las funciones `xlabel()` e `ylabel()` permiten establecer un título para cada uno de los ejes ❸, y la función `tick_params()` da estilo a las marcas de graduación ❹. Los argumentos mostrados aquí afectan a las marcas de graduación tanto en el eje x como en el eje y (`axes='both'`) y establecen el tamaño de fuente de las etiquetas de las marcas de graduación en 14 (`labelize=14`).

Como puede ver en la Figura 15-2, el gráfico resultante es mucho más fácil de leer.

El tipo de etiqueta es más grande y el gráfico de líneas es más grueso.

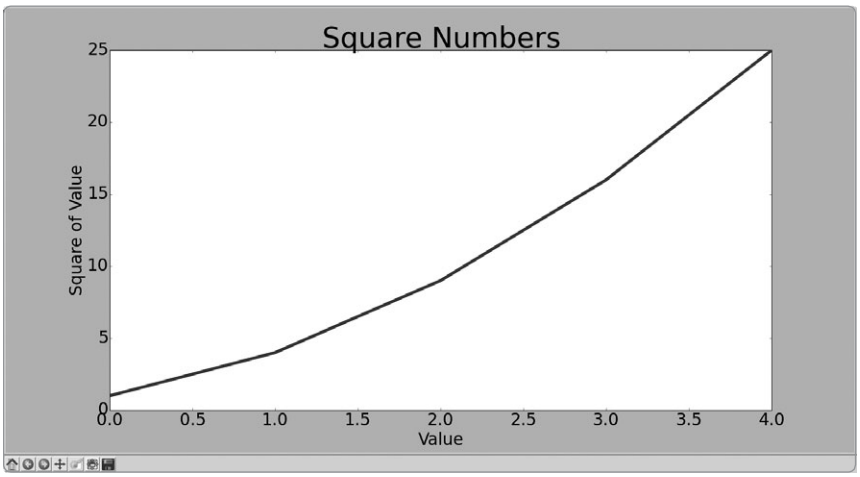


Figura 15-2: Ahora el gráfico es mucho más fácil de leer.

Corrección de la parcela

Pero ahora que podemos leer mejor el gráfico, vemos que los datos no están trazados correctamente. Observa que al final del gráfico el cuadrado de 4,0 aparece como ¡25! Vamos a corregirlo.

Cuando se da a `plot()` una secuencia de números, asume que el primer punto de datos corresponde a un valor de coordenada x de 0, pero nuestro primer punto corresponde a un valor x de 1. Podemos anular el comportamiento por defecto dando a `plot()` tanto los valores de entrada como de salida utilizados para calcular los cuadrados:

mpl_cuadrados.py

```
importar matplotlib.pyplot como plt

valores_entrada = [1, 2, 3, 4, 5]
cuadrados = [1, 4, 9, 16, 25]
plt.plot(valores_entrada, cuadrados,
ancho_línea=5)

# Establece el título del gráfico y etiqueta los ejes.
--snip--
```

Ahora `plot()` graficará los datos correctamente porque hemos proporcionado tanto los valores de entrada como los de salida, por lo que no tiene que asumir cómo se generaron los números de salida. El gráfico resultante, mostrado en la Figura 15-3, es correcto.

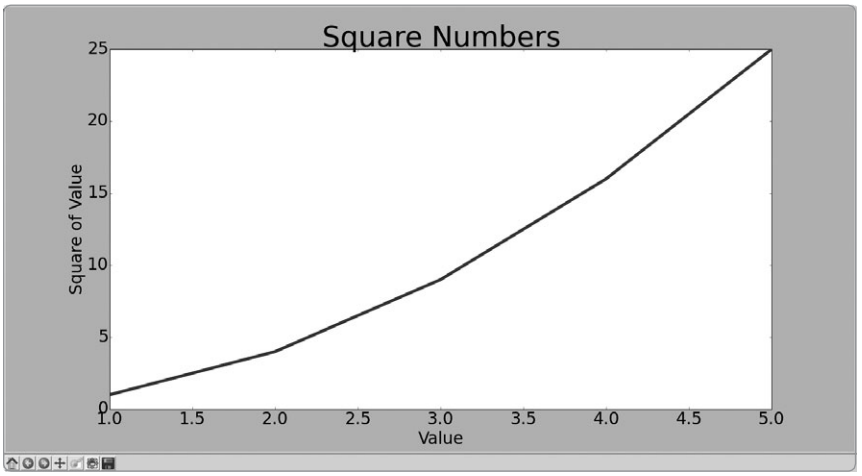


Figura 15-3: Ahora los datos se representan correctamente.

Puede especificar numerosos argumentos cuando utilice `plot()` y utilizar una serie de funciones para personalizar sus gráficos. Seguiremos explorando estas funciones de personalización a medida que trabajemos con conjuntos de datos más interesantes a lo largo de este capítulo.

Trazado y estilización de puntos individuales con scatter()

A veces resulta útil poder trazar y aplicar estilos a puntos individuales en

función de determinadas características. Por ejemplo, puede trazar valores pequeños en un

y los valores más grandes en un color diferente. También puede trazar un gran conjunto de datos con un conjunto de opciones de estilo y, a continuación, resaltar puntos individuales volviendo a trazarlos con opciones diferentes.

Para trazar un único punto, utilice la función `scatter()`. Pase los valores individuales (*x*, *y*) del punto de interés a `scatter()`, y trazará esos valores:

dispersión_cuadrados.py

```
import matplotlib.pyplot as plt

plt.scatter(2, 4)
plt.show()
```

Estilicemos la salida para hacerla más interesante. Añadiremos un título, etiquetaremos los ejes y nos aseguraremos de que todo el texto sea lo suficientemente grande como para poder leerlo:

```
importar matplotlib.pyplot como plt

❶ plt.scatter(2, 4, s=200)

# Establece el título del gráfico y los ejes
de las etiquetas. plt.title("Números
cuadrados", fontsize=24) plt.xlabel("Valor",
fontsize=14) plt.ylabel("Cuadrado del
valor", fontsize=14)

# Establecer el tamaño de las etiquetas de tick.
plt.tick_params(axis='both', which='major', labelize=14)

plt.show()
```

En ❶ llamamos a `scatter()` y utilizamos el argumento `s` para establecer el tamaño de los puntos utilizados para dibujar el gráfico. Cuando ejecute *scatter_squares.py* ahora, debería ver un único punto en el centro del gráfico, como se muestra en la Figura 15-4.

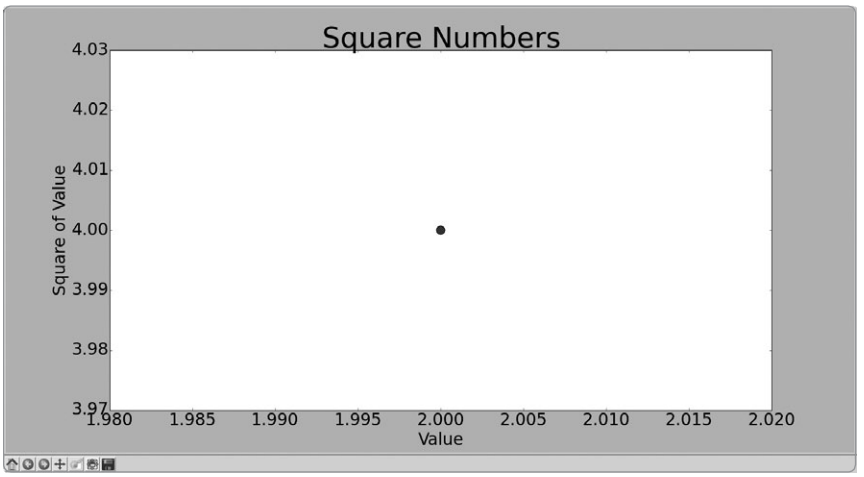


Figura 15-4: Trazado de un único punto

Trazado de una serie de puntos con scatter()

Para trazar una serie de puntos, podemos pasar a `scatter()` listas separadas de valores x e y, de esta forma:

dispersión_cuadrados.py

```
import matplotlib.pyplot as plt

valores_x = [1, 2, 3, 4, 5]
valores_y = [1, 4, 9, 16, 25]

plt.scatter(valores_x, valores_y,

s=100) # Establece el título del

gráfico y etiqueta los ejes.
--snip--
```

La lista `x_values` contiene los números que hay que elevar al cuadrado, e `y_values` contiene el cuadrado de cada número. Cuando estas listas se pasan a `scatter()`, matplotlib lee un valor de cada lista al trazar cada punto. Los puntos a representar son (1, 1), (2, 4), (3, 9), (4, 16), y (5, 25); el resultado se muestra a continuación en la Figura 15-5.

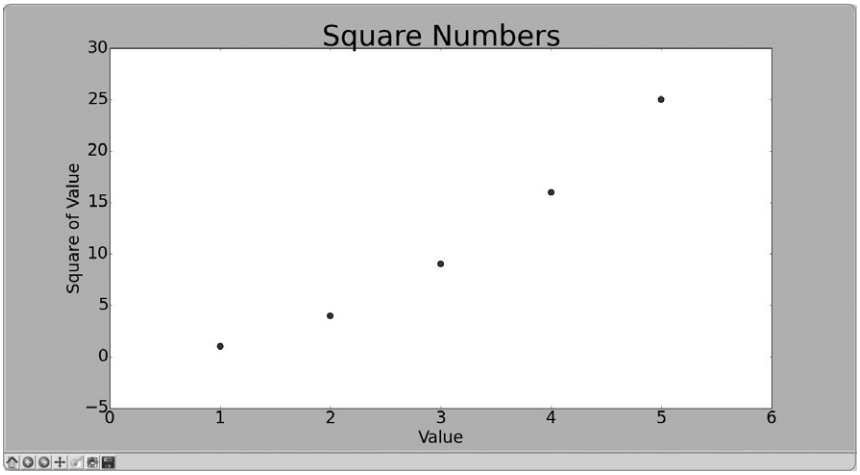


Figura 15-5: Diagrama de dispersión con múltiples puntos

Cálculo automático de datos

Escribir listas a mano puede ser ineficiente, especialmente cuando tenemos muchos puntos. En lugar de pasar nuestros puntos en una lista, vamos a utilizar un bucle en Python para hacer los cálculos por nosotros. Así se vería con 1000 puntos:

*dispersión_*

```
import matplotlib.pyplot as plt
```

cuadrados.py

```
❶ x_valores = list(range(1, 1001))
  y_valores = [x**2 for x in x_valores]

❷ plt.scatter(valores_x, valores_y, s=40)
```

```
    # Establece el título del gráfico y etiqueta los ejes.
    --snip--

# Establece el rango para cada eje.
  ❸ plt.axis([0, 1100, 0, 1100000])

    plt.show()
```

Empezamos con una lista de valores x que contiene los números del 1 al 1000 ❶. A continuación, una lista de comprensión genera los valores y mediante un bucle a través de los valores x (`para x en valores_x`), elevando al cuadrado cada número (`x**2`), y almacenando los resultados en `y_valores`. A continuación, pasamos las listas de entrada y salida a `scatter()` ❷.

Dado que se trata de un conjunto de datos grande, utilizamos un tamaño de punto más pequeño y usamos la función `axis()` para especificar el rango de cada eje ❸. La función `axis()` requiere cuatro valores: los valores mínimo y máximo para el eje x y e 1 eje y. Aquí, ejecutamos el eje x de 0 a 1100 y el eje y de 0 a 1.100.000. La Figura 15-6 muestra el resultado.

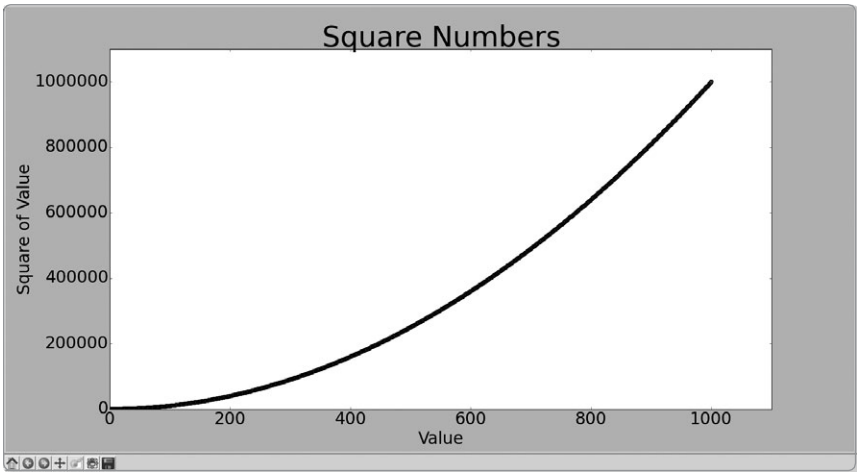


Figura 15-6: Python puede trazar 1000 puntos tan fácilmente como 5 puntos.

Eliminación de contornos de puntos de datos

matplotlib le permite colorear los puntos individualmente en un gráfico de dispersión. El valor predeterminado (puntos azules con un contorno negro) funciona bien para gráficos con pocos puntos. Pero cuando se trazan muchos puntos, los contornos negros pueden mezclarse. Para eliminar los contornos alrededor de los puntos, pase el argumento `edgecolor='none'` cuando llame a `scatter()`:

```
plt.scatter(valores_x, valores_y, color_borde='ninguno', s=40)
```

Ejecute *scatter_squares.py* utilizando esta llamada, y debería ver sólo puntos azules sólidos en su gráfico.

Definición de colores personalizados

Para cambiar el color de los puntos, pase `c` a `scatter()` con el nombre de un color a utilizar, como se muestra aquí:

```
plt.scatter(valores_x, valores_y, c='rojo', color_borde='ninguno', s=40)
```

También puede definir colores personalizados utilizando el modelo de color RGB. Para definir un color, pase al argumento `c` una tupla con tres valores decimales (uno para cada rojo, verde y azul), utilizando valores entre 0 y 1. Por ejemplo, la siguiente línea crearía un gráfico con puntos azul claro:

```
plt.scatter(valores_x, valores_y, c=(0, 0, 0.8), edgecolor='none', s=40)
```

Los valores más cercanos a 0 producen colores oscuros, y los valores más cercanos a 1 producen colores más claros.

Utilizar un mapa de colores

Un mapa de *colores* es una serie de colores en un gradiente que se desplaza desde un color inicial hasta un color final. Los mapas de colores se utilizan en visualizaciones para enfatizar un patrón en los datos. Por ejemplo, puede hacer que los valores bajos tengan un color claro y los valores altos un color más oscuro.

El módulo `pyplot` incluye un conjunto de mapas de color incorporados. Para utilizar uno de estos mapas de colores, es necesario especificar cómo `pyplot` debe asignar un color a cada punto del conjunto de datos. A continuación se muestra cómo asignar a cada punto un color basado en su valor y:

dispersión_cuadrados.py

```
import matplotlib.pyplot as plt

x_valores = list(range(1001))
y_valores = [x**2 for x in
x_valores]

plt.scatter(valores_x, valores_y, c=valores_y, cmap=plt.cm.Blues,
    edgecolor='none', s=40)

# Establece el título del gráfico y etiqueta los ejes.
--snip--
```

Pasamos la lista de valores de y a `c` y luego le decimos a `pyplot` qué mapa de colores utilizar a través del argumento `cmap`. Este código colorea los puntos con valores
azul claro y los puntos con valores y mayores azul oscuro. El gráfico resultante se muestra en la Figura 15-7.

NOTE *Puede ver todos los mapas de colores disponibles en pyplot en http://matplotlib.org/; vaya a*

***Ejemplos**, desplácese hasta Ejemplos de color y haga clic en **colormaps_reference**.*

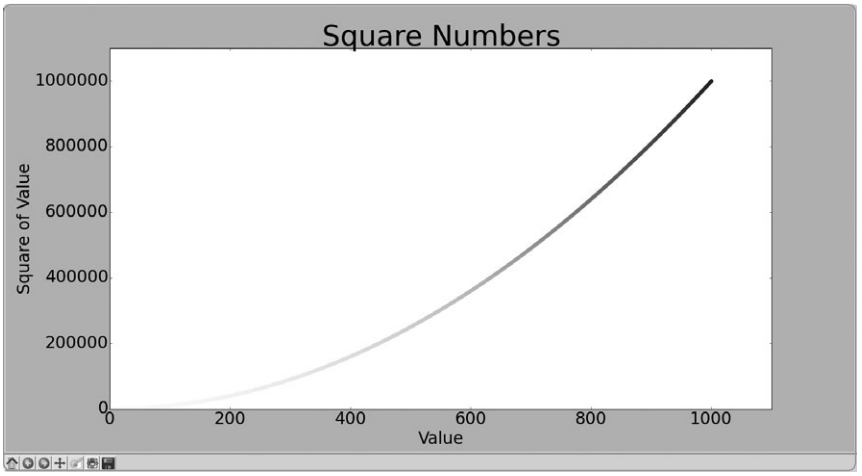


Figura 15-7: Gráfico con el mapa de colores Blues

Guardar las parcelas automáticamente

Si desea que su programa guarde automáticamente el gráfico en un archivo, puede sustituir la llamada a `plt.show()` por una llamada a `plt.savefig()`:

```
plt.savefig('squares_plot.png', bbox_inches='tight')
```

El primer argumento es un nombre de archivo para la imagen del gráfico, que se guardará en el mismo directorio que *scatter_squares.py*. El segundo argumento recorta los espacios en blanco del gráfico. Si desea el espacio en blanco extra alrededor del gráfico, puede omitir este argumento.

TRY IT yOURSELF

15-1. Cubos: Un número elevado a la tercera potencia es un *cubo*. Traza los cinco primeros números cúbicos y, a continuación, los 5000 primeros números cúbicos.

15-2. Cubos coloreados: Aplique un mapa de colores a su gráfico de cubos.

Paseos al azar

En esta sección usaremos Python para generar datos para un paseo aleatorio y luego usaremos matplotlib para crear una representación visualmente atractiva de los datos generados. Un *paseo aleatorio* es un camino que no tiene una dirección clara pero es

determinado por una serie de decisiones aleatorias, cada una de las cuales se deja enteramente al azar. Un paseo aleatorio es el camino que seguiría una hormiga si perdiera la cabeza y diera cada paso en una dirección aleatoria.

Los paseos aleatorios tienen aplicaciones prácticas en la naturaleza, la física, la biología, la química y la economía. Por ejemplo, un grano de polen flotando en una gota de agua se mueve por la superficie del agua porque las moléculas de agua lo empujan constantemente. El movimiento molecular en una gota de agua es aleatorio, por lo que la trayectoria que traza un grano de polen en la superficie es un paseo aleatorio. El código que vamos a escribir reproduce muchas situaciones del mundo real.

Creación de la clase RandomWalk()

Para crear un paseo aleatorio, crearemos una clase `RandomWalk`, que tomará decisiones aleatorias sobre la dirección que debe tomar el paseo. La clase necesita tres atributos: una variable para almacenar el número de puntos del paseo y dos listas para almacenar los valores de las coordenadas x e y de cada punto del paseo.

Utilizaremos sólo dos métodos para la clase `RandomWalk`: el método init () y `fill_walk()`, que calculará los puntos del paseo. Comencemos con `init ()` como se muestra aquí:

random_caminar.py

```
❶ from random import choice

clase RandomWalk():
"""Una clase para generar paseos aleatorios."""

❷     def init (self, num_points=5000):
          """Inicializar atributos de un
          paseo."""  self.num_points =
          num_points.

          # Todos los paseos comienzan en (0, 0).
❸         auto.valores_x =
          [0] auto.valores_y
          = [0]
```

Para tomar decisiones aleatorias, almacenaremos las posibles opciones en una lista y usaremos `choice()` para decidir qué opción usar cada vez que se tome una decisión ❶. A continuación, establecemos el número predeterminado de puntos en un paseo a 5000-suficientemente grande para generar algunos patrones interesantes, pero lo suficientemente pequeño como para generar paseos rápidamente ❷. A continuación, en ❸ hacemos dos listas para mantener los valores x e y, y empezamos cada paseo en el punto (0, 0).

Elegir la dirección

Usaremos `fill_walk()`, como se muestra aquí, para llenar nuestro paseo con puntos y determinar la dirección de cada paso. Añade este método a

random_walk.py:

aleatorio_paseo.py

```
    def llenar_camino(self):
        """Calcula todos los puntos del paseo."""

        # Sigue dando pasos hasta que el paseo alcance la longitud deseada.
(1)     while len(auto.valores_x) < auto.número_puntos:
```

```
            # Decide qué dirección tomar y hasta dónde llegar en esa dirección.
❷           x_dirección = elección([1, -1])
            x_distancia = elección([0, 1, 2, 3,
            4])
❸           paso_x = dirección_x * distancia_x

            y_dirección = elección([1, -1])
            y_distancia = elección([0, 1, 2, 3,
            4])
❹           paso_y = dirección_y * distancia_y

            # Rechaza los movimientos que no van a ninguna parte.
❺            si paso_x == 0 y paso_y == 0:
               continuar

        # Calcula los siguientes valores x e y.
❻             next_x = self.x_values[-1] +
          x_step next_y = self.y_values[-1]
                                  + y_step

            auto.valores_x.append(siguiente_x)
            auto.valores_y.append(siguiente_y)
```

En ❶ establecemos un bucle que se ejecuta hasta que el paseo se llena con el número correcto de puntos. La parte principal de este método le dice a Python cómo simular cuatro decisiones aleatorias: ¿El paseo irá a la derecha o a la izquierda? ¿Qué distancia recorrerá en esa dirección? ¿Subirá o bajará? ¿Qué distancia recorrerá en esa dirección?

Usamos `choice([1, -1])` para elegir un valor para `x_direction`, que devuelve 1 para movimiento a la derecha o -1 para movimiento a la izquierda ❷. A continuación, `choice([0, 1, 2, 3, 4])` le dice a Python hasta dónde moverse en esa dirección (`distancia_x`) seleccionando aleatoriamente un número entero entre 0 y 4. (La inclusión de un 0 nos permite dar pasos a lo largo del eje y, así como pasos que tengan movimiento a lo largo de ambos ejes).

En ❸ y ❹ determinamos la longitud de cada paso en las direcciones x e *y* multiplicando la dirección del movimiento por la distancia elegida. Un resultado positivo para `x_paso nos` desplaza a la derecha, un resultado negativo nos desplaza a la izquierda y 0 nos desplaza verticalmente. Un resultado positivo para `y_step` significa moverse hacia arriba, negativo significa moverse hacia abajo, y 0 significa moverse horizontalmente. Si el valor tanto de `x_step` como de `y_step` es 0, el paseo se detiene, pero continuamos el bucle para evitarlo ❺.

Para obtener el siguiente valor x para nuestro paseo, sumamos el valor en `x_step` al último valor almacenado en `x_values` ❻ y hacemos lo mismo para los valores y. Una vez que tenemos estos valores, los anexamos a `x_values` y `y_values`.

Trazado del paseo aleatorio

Aquí está el código para trazar todos los puntos en el paseo:

rw_visual.py

```
importar matplotlib.pyplot como plt

from random_walk import RandomWalk

# Haz un paseo aleatorio, y traza los puntos.
rw = RandomWalk()
rw.fill_walk()
```

```
❷ plt.scatter(rw.valores_x, rw.valores_y, s=15)
plt.show()
```

Comenzamos importando `pyplot` y `RandomWalk`. A continuación, creamos un recorrido aleatorio y lo almacenamos en `rw` ❶, asegurándonos de llamar a `fill_walk()`. En ❷ alimentamos los valores x- e y- del paseo a `scatter()` y elegimos un tamaño de punto apropiado. La Figura 15-8 muestra el gráfico resultante con 5000 puntos. (Las imágenes de esta sección omiten el visor de matplotlib, pero seguirá viéndolo cuando ejecute *rw_visual.py*).

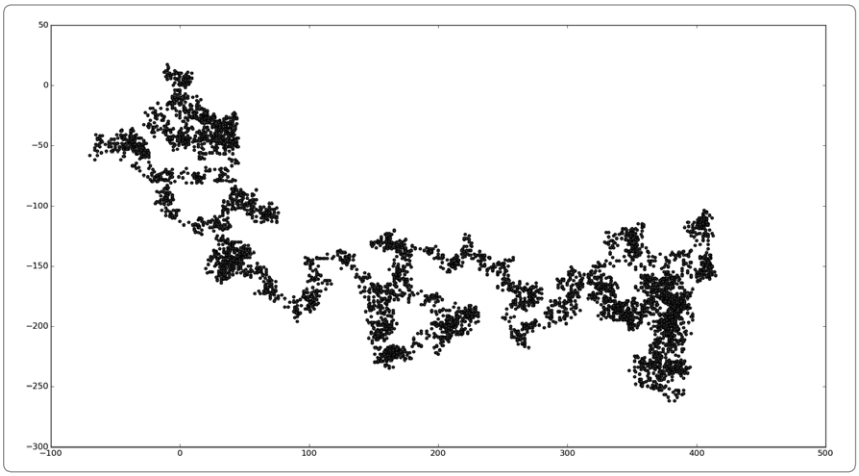


Figura 15-8: Un paseo aleatorio con 5000 puntos

Generación de múltiples recorridos aleatorios

Cada paseo aleatorio es diferente, y es divertido explorar los diversos patrones que se pueden generar. Una forma de utilizar el código anterior para hacer múltiples paseos sin tener que ejecutar el programa varias veces es envolverlo en un bucle `while`, así:

rw_visual.py

```
importar matplotlib.pyplot como plt

from random_walk import RandomWalk

# Sigue haciendo nuevos paseos, mientras el programa esté
activo. while True:
    # Realiza un paseo aleatorio, y traza los
    puntos. rw = RandomWalk()
    rw.fill_walk()
    plt.scatter(rw.valores_x, rw.valores_y,
    s=15) plt.show()

❶   keep_running = input("¿Dar otro paseo? (y/n): ")
    if keep_running == 'n':
        romper
```

Este código generará un paseo aleatorio, lo mostrará en el visor de matplotlib, y hará una pausa con el visor abierto. Cuando cierres el visor, se te preguntará si quieres generar otro paseo. Responda **y**, y usted debería ser capaz de generar paseos que permanecen cerca del punto de partida, que vagan en su mayoría en una dirección, que tienen secciones delgadas que conectan grandes grupos de puntos, y así sucesivamente. Cuando desee finalizar el programa, introduzca **n**.

NOTE *Si estás usando Python 2.7, recuerda usar* raw_input() *en lugar de* input() *en* ❶.

Estilizar el paseo

En esta sección personalizaremos nuestros gráficos para destacar las características importantes de cada paseo y restar importancia a los elementos que distraen. Para ello, identificaremos las características que queremos resaltar, como dónde empezó el paseo, dónde terminó y el camino recorrido. A continuación, identificamos las características que queremos restar importancia, como las marcas de graduación y las etiquetas. El resultado debe ser una representación visual sencilla que comunique claramente el camino recorrido en cada paseo aleatorio.

Colorear los puntos

Utilizaremos un mapa de colores para mostrar el orden de los puntos en el paseo y luego eliminaremos el contorno negro de cada punto para que el color de los puntos sea más claro. Para colorear los puntos según su posición en el paseo, pasamos al argumento `c` una lista que contiene la posición de cada punto. Como los puntos se trazan en orden, la lista sólo contiene los números del 1 al 5000, como se muestra aquí:

rw_visual.py

```
--snip--
mientras sea verdad:
    # Realiza un paseo aleatorio, y traza los
    puntos. rw = RandomWalk()
    rw.fill_walk()

❶   números_punto = list(rango(rw.número_puntos))
    plt.scatter(rw.valores_x, rw.valores_y, c=números_punto, cmap=plt.cm.Blues,
        edgecolor='none', s=15)
    plt.show()

    keep_running = input("¿Dar otro paseo? (y/n): ")
    --snip--
```

En ❶ usamos `range()` para generar una lista de números igual al número de puntos del paseo. Luego los almacenamos en la lista `números_punto`, que usaremos para establecer el color de cada punto del paseo. Pasamos `point_numbers` al argumento `c`, usamos el colormap `Blues`, y luego pasamos `edgecolor=none` para deshacernos del contorno negro alrededor de cada punto. El resultado es un trazado del paseo que varía de azul claro a azul oscuro a

lo largo de un gradiente, como se muestra en la Figura 15-9.

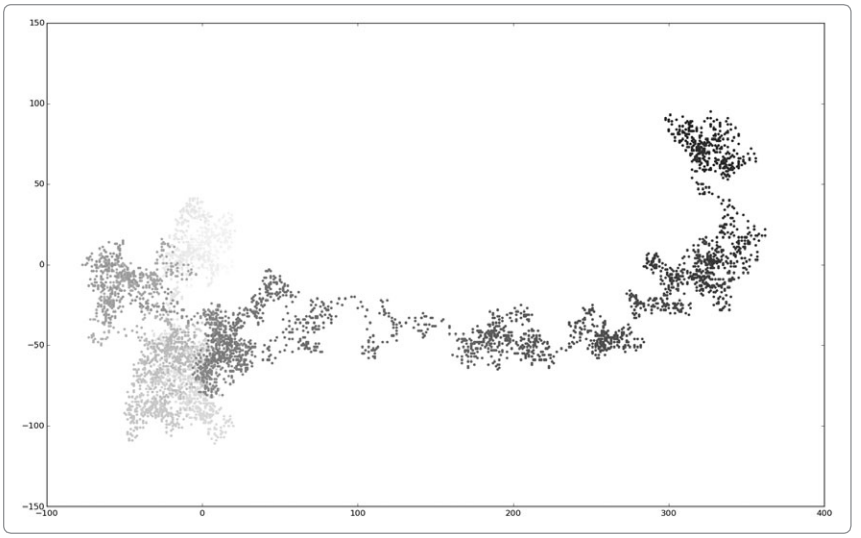


Figura 15-9: Un paseo aleatorio coloreado con el mapa de colores Blues

Trazado de los puntos inicial y final

Además de colorear los puntos para mostrar su posición a lo largo del recorrido, estaría bien ver dónde empieza y acaba cada recorrido. Para ello, podemos trazar los puntos primero y último individualmente una vez trazada la serie principal. Haremos los puntos finales más grandes y los colorearemos de forma diferente para que destaquen, como se muestra aquí:

rw_visual.py

```
--snip--
mientras sea verdad:
    --snip--
    plt.scatter(rw.valores_x, rw.valores_y, c=números_punto, cmap=plt.cm.Blues,
        edgecolor='none', s=15)

    # Enfatiza el primer y último punto.
    plt.scatter(0, 0, c='verde', edgecolors='ninguno', s=100)
    plt.scatter(rw.valores_x[-1], rw.valores_y[-1], c='rojo',
    edgecolors='ninguno',
        s=100)

    plt.show()
    --snip--
```

Para mostrar el punto inicial, trazamos el punto (0, 0) en verde en un tamaño mayor (`s=100`) que el resto de los puntos. Para marcar el punto final, trazamos el último
x- y y-valor en el paseo en rojo con un tamaño de 100. Asegúrate de insertar este código justo antes de la llamada a `plt.show()` para que los puntos inicial y final se dibujen encima de todos los demás puntos.

Cuando ejecutes este código, deberías ser capaz de distinguir

exactamente dónde empieza y termina cada paseo. (Si estos puntos finales no destacan claramente, ajusta su color y tamaño hasta que lo hagan).

Limpieza de los ejes

Quitemos los ejes de este gráfico para que no nos distraigan de la trayectoria de cada paseo. Para desactivar los ejes, utiliza este código:

rw_visual.py

```
--snip--
mientras sea verdad:
    --snip--
    plt.scatter(rw.valores_x[-1], rw.valores_y[-1], c='rojo',
        edgecolors='ninguno', s=100)

    # Elimina los ejes.
1   plt.axes().get_xaxis().set_visible(False)
    plt.axes().get_yaxis().set_visible(False)

    plt.show()
    --snip--
```

Para modificar los ejes, utilice la función `plt.axes()` ❶ para establecer la visibilidad de cada eje en `False`. A medida que continúe trabajando con visualizaciones, verá con frecuencia este encadenamiento de métodos.

Ejecute ahora *rw_visual.py*; debería ver una serie de gráficos sin ejes.

Añadir puntos de trazado

Vamos a aumentar el número de puntos para tener más datos con los que trabajar. Para ello, aumentamos el valor de `num_points` cuando hacemos una instancia de `RandomWalk` y ajustamos el tamaño de cada punto al dibujar el gráfico, como se muestra aquí:

rw_visual.py

```
--snip--
mientras sea verdad:
    # Realiza un paseo aleatorio, y traza los
    puntos. rw = RandomWalk(50000)
    rw.fill_walk()

    # Trazar los puntos, y mostrar el gráfico.
    point_numbers = list(range(rw.num_points))
    plt.scatter(rw.valores_x, rw.valores_y, c=números_punto, cmap=plt.cm.Blues,
        edgecolor='none', s=1)
    --snip--
```

Este ejemplo crea un paseo aleatorio con 50.000 puntos (para reflejar los datos del mundo real) y traza cada punto a un tamaño `s=1`. El paseo resultante es difuso y con aspecto de nube, como se muestra en la Figura 15-10. El paseo resultante es ondulado y como una nube, como se muestra en la Figura 15-10. Como puedes ver, hemos creado una obra de arte a partir de un simple gráfico de dispersión. Como puedes ver, ¡hemos creado una obra de arte a partir de un simple gráfico de dispersión!

Experimenta con este código para ver cuánto puedes aumentar el número de puntos en un paseo antes de que tu sistema empiece a ralentizarse significativamente o el gráfico pierda su atractivo visual.

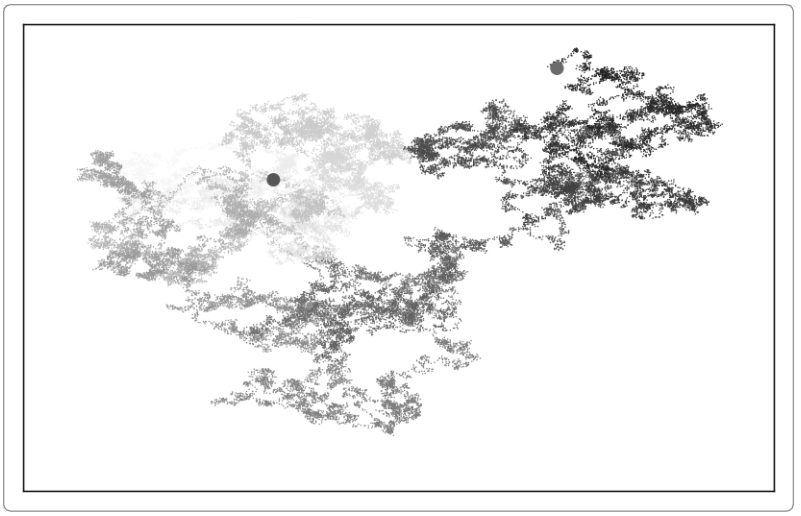


Figura 15-10: Un paseo con 50.000 puntos

Alterar el tamaño para llenar la pantalla

Una visualización es mucho más eficaz para comunicar patrones en los datos si se ajusta bien a la pantalla. Para que la ventana de trazado se ajuste mejor a su pantalla, ajuste el tamaño de la salida de matplotlib, así:

rw_visual.py

```
--snip--
mientras sea verdad:
    # Realiza un paseo aleatorio, y traza los
    puntos. rw = RandomWalk()
    rw.fill_walk()

    # Establecer el tamaño de la ventana de
    trazado. plt.figure(figsize=(10, 6))
    --snip--
```

La función `figure()` controla la anchura, la altura, la resolución y el color de fondo del gráfico. El parámetro `figsize` toma una tupla, que indica a matplotlib las dimensiones de la ventana de trazado en pulgadas.

Python asume que la resolución de su pantalla es de 80 píxeles por pulgada; si este código no le da un tamaño de gráfico exacto, ajuste los números según sea necesario. O, si conoce la resolución de su sistema, pase a `figure()` la resolución usando el parámetro `dpi` para establecer un tamaño de gráfico que haga un uso efectivo del espacio disponible en su pantalla, como se muestra aquí:

```
plt.figure(dpi=128, figsize=(10, 6))
```

TRY IT yOURSELF

15-3. Movimiento molecular: Modifique *rw_visual.py* sustituyendo `plt.scatter()` por `plt.plot()`. Para simular la trayectoria de un grano de polen en la superficie de una gota de agua, introduzca los valores `rw.x_values` y `rw.y_values`, e incluya un argumento de `anchura de línea`. Utilice 5000 en lugar de 50.000 puntos.

15-4. Caminatas aleatorias modificadas: En la clase `RandomWalk`, `x_paso` e `y_paso` se generan a partir del mismo conjunto de condiciones. La dirección se elige aleatoriamente de la lista `[1, -1]` y la distancia de la lista `[0, 1, 2, 3, 4]`. Modifica los valores de estas listas para ver qué ocurre con la forma general de tus paseos. Prueba con una lista más larga de opciones para la distancia, como de 0 a 8, o elimina el -1 de la lista de dirección *x* o *y*.

15-5. Refactorización: El método `fill_walk()` es largo. Cree un nuevo método llamado `get_step()` para determinar la dirección y distancia de cada paso, y luego calcule el paso. Debería terminar con dos llamadas a get_step() en `fill_walk()`:

```
x_paso = get_paso()
y_paso = get_paso()
```

Esta refactorización debería reducir el tamaño de `fill_walk()` y hacer que el método sea más fácil de leer y entender.

Tirar dados con Pygal

En esta sección utilizaremos el paquete de visualización Pygal de Python para producir archivos de gráficos vectoriales escalables. Estos son útiles en visualizaciones que se presentan en pantallas de diferentes tamaños porque se escalan automáticamente para adaptarse a la pantalla del espectador. Si planeas usar tus visualizaciones online, considera usar Pygal para que tu trabajo se vea bien en cualquier dispositivo que la gente use para ver tus visualizaciones.

En este proyecto analizaremos los resultados de lanzar dados. Si tiras un dado normal de seis caras, tienes las mismas probabilidades de sacar cualquiera de los números del 1 al 6. Sin embargo, si usas dos dados, tienes más probabilidades de sacar unos números que otros. Sin embargo, si utilizas dos dados, tienes más probabilidades de sacar ciertos números que otros. Intentaremos determinar qué números tienen más probabilidades de salir generando un conjunto de datos que represente el lanzamiento de dados. A continuación, representaremos gráficamente los resultados de un gran número de tiradas para determinar qué resultados son más probables que otros.

El estudio del lanzamiento de dados se utiliza a menudo en matemáticas para explicar diversos tipos de análisis de datos. Pero también tiene aplicaciones reales en los casinos y otros escenarios de juego, así como en la forma de jugar

a juegos como el Monopoly y muchos juegos de rol.

Instalación de Pygal

Instala Pygal con `pip`. (Si aún no ha utilizado `pip`, consulte "Instalación de paquetes Python con pip" en la página 237).

En Linux y OS X, esto debería ser algo como:

```
pip install --user pygal
```

En Windows, esto debería ser:

```
python -m pip install --user pygal
```

NOTE *Puede que tenga que utilizar el comando `pip3` en lugar de `pip`, y si el comando sigue sin funcionar puede que tenga que omitir la bandera `--user`.*

Galería Pygal

Para ver qué tipo de visualizaciones son posibles con Pygal, visita la galería de tipos de gráficos: ve a http://www.pygal.org/, haz clic en **Documentación** y, a continuación, en **Tipos de gráficos**. Cada ejemplo incluye el código fuente, para que puedas ver cómo se generan las visualizaciones.

Creación de la clase de troquel

He aquí una clase para simular la tirada de un dado:

morir.py

```
from random import randint

class Muere():
    """Una clase que representa a un único dado."""

❶   def init (self, num_sides=6):
        """Supongamos un dado de seis
        caras."""  self.num_sides =
        num_sides

    def roll(self):
        """Devuelve un valor aleatorio entre 1 y el número de lados."""
❷       return randint(1, self.num_sides)
```

El método `init ()` toma un argumento opcional. Con esta clase, cuando se crea una instancia de nuestro dado, el número de caras siempre será seis si no se incluye ningún argumento. Si *se incluye* un argumento, ese valor se utiliza para establecer el número de caras del dado ❶. (Los dados se nombran por su número de caras: un dado de seis caras es un D6, un dado de ocho caras es un D8, y así sucesivamente).

El método `roll()` utiliza la función `randint()` para devolver un número aleatorio entre 1 y el número de lados ❷. Esta función puede devolver el valor inicial (1), el valor final (`num_lados`) o cualquier número entero entre los dos.

Tirar el dado

Antes de crear una visualización basada en esta clase, vamos a tirar un D6, imprimir los resultados y comprobar que los resultados parecen razonables:

die_visual.py

```
from troquel import Troquel

# Crea un D6.
❶ die = Die()

# Haz algunas tiradas, y almacena los
resultados en una lista. results = []
❷ for roll_num in range(100):
    result = die.roll()
    results.append(result)

print(resultados)
```

En ❶ creamos una instancia de `Dado` con las seis caras por defecto. En ❷ lanzamos el dado 100 veces y almacenamos los resultados de cada tirada en la lista `resultados`. Aquí tenemos un conjunto de resultados de ejemplo:

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

Un examen rápido de estos resultados muestra que la clase `Die` parece funcionar. Vemos los valores 1 y 6, por lo que sabemos que se devuelven los valores más pequeños y más grandes posibles, y como no vemos 0 ni 7, sabemos que todos los resultados están en el rango apropiado. También vemos todos los números del 1 al 6, lo que indica que están representados todos los resultados posibles.

Análisis de los resultados

Analizamos los resultados de tirar un D6 contando cuántas veces tiramos cada número:

die_visual.py

```
--snip--
# Haz algunas tiradas, y almacena los
resultados en una lista. results = []
❶ for roll_num in
    range(1000): result =
    die.roll()
    results.append(result)

# Analizar los resultados.
frecuencias = []
❷ for value in range(1, die.num_sides+1):
❸     frecuencia = resultados.contar(valor)
❹     frequencies.append(frecuencia)
```

```
print(frecuencias)
```

Como estamos usando Pygal para analizar en lugar de imprimir los resultados, podemos aumentar el número de tiradas simuladas a 1000 ❶. Para analizar las tiradas, creamos las `frecuencias de` lista vacías para almacenar el número de veces que sale cada valor. Hacemos un bucle a través de los valores posibles (del 1 al 6 en este caso) en ❷, contamos cuántas veces aparece cada número en `los resultados` ❸, y luego añadimos este valor a la lista `frecuencias` ❹. A continuación, imprimimos esta lista antes de hacer una visualización:

```
[155, 167, 168, 170, 159, 181]
```

Estos resultados parecen razonables: vemos seis frecuencias, una para cada número posible al tirar un D6, y vemos que ninguna frecuencia es significativamente mayor que otra. Ahora visualicemos estos resultados.

Cómo hacer un histograma

Con una lista de frecuencias, podemos hacer un histograma de los resultados. Un *histograma* es un gráfico de barras que muestra la frecuencia con la que se producen determinados resultados. Aquí está el código para crear el histograma:

morir_visual.py

```
importar pygal
--snip--

# Analizar los resultados.
frecuencias = []
for value in range(1, die.num_sides+1):
    frequency = results.count(value)
    frequencies.append(frequency)

# Visualiza los resultados.
❶ hist = pygal.Bar()

hist.title = "Resultados de tirar un D6 1000 veces".
❷ hist.x_labels = ['1', '2', '3', '4', '5',
'6'] hist.x_title = "Resultado"
hist.y_title = "Frecuencia del resultado"

❸ hist.add('D6', frequencies)
hist.render_to_file('die_visual.svg')
```

Hacemos un gráfico de barras creando una instancia de `pygal.Bar()`, que almacenamos en `hist` ❶. A continuación, establecemos el atributo `title` de `hist` (sólo una cadena que usamos para etiquetar el histograma), usamos los posibles resultados de tirar un D6 como etiquetas para el eje x ❷, y añadimos un título para cada uno de los ejes. Usamos `add()` para añadir una serie de

valores al gráfico en ❸ (pasándole una etiqueta para el conjunto de valores a añadir y una lista de los valores que aparecerán en el gráfico). Finalmente, renderizamos el `grafico` a un archivo SVG, que espera un nombre de archivo con la extensión *.svg*.

La forma más sencilla de ver el histograma resultante es en un navegador web.

Abra una nueva pestaña en cualquier navegador web y, a continuación, abra el archivo *die_visual.svg*
(en la carpeta donde guardaste *die_visual.py*). Deberías ver un gráfico como el

el de la Figura 15-11. (He modificado ligeramente este gráfico para imprimirlo; por defecto Pygal genera gráficos con un fondo más oscuro que el que se ve aquí).

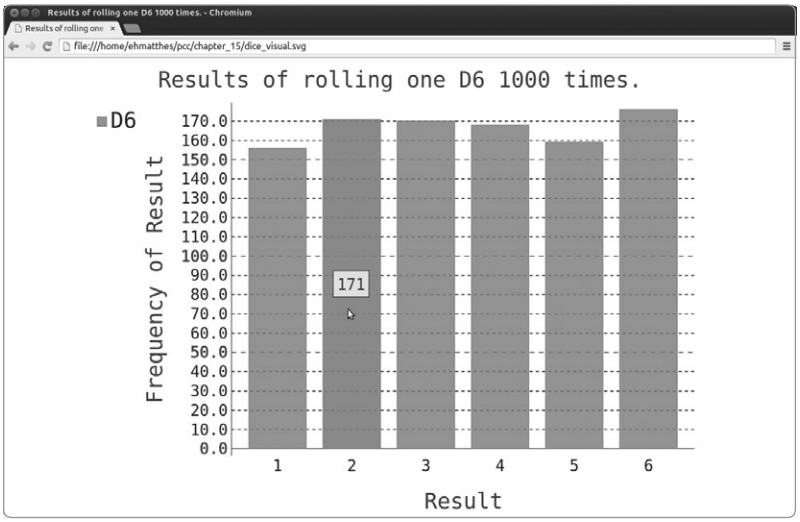


Figura 15-11: Un simple gráfico de barras creado con Pygal

Fíjate en que Pygal ha hecho el gráfico interactivo: pasa el cursor por encima de cualquier barra del gráfico y verás los datos asociados a ella. Esta característica es particularmente útil cuando se trazan múltiples conjuntos de datos en el mismo gráfico.

Tirar dos dados

Tirando dos dados se obtienen números más grandes y una distribución diferente de los resultados. Vamos a modificar nuestro código para crear dos dados D6 para simular la forma en que lanzamos un par de dados. Cada vez que lancemos el par, sumaremos los dos números (uno de cada dado) y almacenaremos la suma en `resultados`. Guarda una copia de *die_visual.py* como *dice_visual.py*, y haz los siguientes cambios:

dice_visual.py

```
importar pygal

from die import Die

# Crear dos dados D6.
die_1 = Die()
troquel_2 = Troquel()

# Haz algunas tiradas, y almacena los
resultados en una lista. results = []
para roll_num en range(1000):
❶    result = die_1.roll() + die_2.roll()
    results.append(result)

# Analizar los resultados.
```

```
    frecuencias = []
❷ max_result = die_1.num_sides + die_2.num_sides
```

```
❸ for value in range(2,
    max_result+1): frequency =
    results.count(value)
    frequencies.append(frequency)

# Visualizar los
resultados. hist =
pygal.Bar()

❹ hist.title = "Resultados de lanzar dos dados D6 1000 veces."
hist.x_labels = ['2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12']
hist.x_title = "Resultado"
hist.y_title = "Frecuencia del resultado"

hist.add('D6 + D6', frequencies)
hist.render_to_file('dice_visual.svg')
```

Después de crear dos instancias de `Die`, tiramos los dados y calculamos la suma de los dos dados de cada tirada ❶. El mayor resultado posible (12) es la suma del mayor número en ambos dados, que almacenamos en `max_result` ❷. El resultado más pequeño posible (2) es la suma del número más pequeño en ambos d a d o s . Cuando analizamos los resultados, contamos el número de resultados para cada valor entre 2 y `max_result` ❸. (Podríamos haber utilizado `range(2, 13)`, pero esto sólo funcionaría para dos dados D6). Cuando se modelan situaciones del mundo real, es mejor escribir código que pueda modelar fácilmente una variedad de situaciones. Este código nos permite simular el lanzamiento de un par de dados con cualquier número de caras).

Al crear el gráfico, actualizamos el título y las etiquetas para el eje x y las series de datos ❹. (Si la lista `x_etiquetas` fuera mucho más larga, tendría sentido escribir un bucle para generar esta lista automáticamente).

Después de ejecutar este código, actualice la pestaña de su navegador que muestra el gráfico; debería ver un gráfico como el de la Figura 15-12.

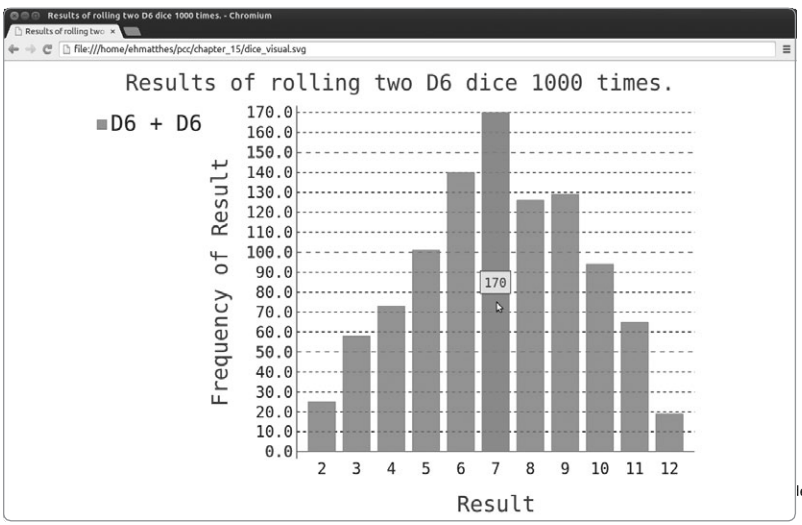


Figura 15-12: Resultados simulados de lanzar dos dados de seis caras 1000 veces

Este gráfico muestra los resultados aproximados que es probable que obtengas al tirar un par de dados D6. Como puedes ver, lo menos probable es que saques un 2 o un 12 y lo más probable es que saques un 7 porque hay seis formas de sacar un 7, a saber: 1 y 6, 2 y 5, 3 y 4, 4 y 3, 5 y 2, o 6 y 1.

Tirar dados de distintos tamaños

Creemos un dado de seis caras y otro de diez, y veamos qué ocurre cuando los lanzamos 50.000 veces:

diferente_dados.py

```
from die import Die

importar pygal

# Crea un D6 y un D10.
die_1 = Die()
❶ dado_2 = Dado(10)

# Haz algunas tiradas, y almacena los
resultados en una lista. results = []
para roll_num en range(50000):
    resultado = dado_1.tirar() +
    dado_2.tirar()
    resultados.append(resultado)

# Analiza los resultados.
--snip--

# Visualizar los
resultados. hist =
pygal.Bar()

❷ hist.title = "Resultados de tirar un D6 y un D10 50.000 veces"
hist.x_labels = ['2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12',
    '13', '14', '15', '16']
hist.x_title = "Resultado"
hist.y_title = "Frecuencia del resultado"

hist.add('D6 + D10', frequencies)
hist.render_to_file('dice_visual.svg')
```

Para hacer un D10, pasamos el argumento `10` al crear la segunda instancia del `Dado` ❶ y cambiamos el primer bucle para simular 50.000 tiradas en lugar de 1000. El resultado más bajo posible sigue siendo 2, pero el resultado más grande es ahora 16; así que ajustamos el título, las etiquetas del eje x y las etiquetas de las series de datos para reflejarlo ❷.

La Figura 15-13 muestra el gráfico resultante. En lugar de un resultado más probable, hay cinco. Esto sucede porque todavía hay sólo una manera de sacar el valor más pequeño (1 y 1) y el valor más grande (6 y 10), pero el dado más pequeño limita el número de maneras en que puede generar los números del medio: hay seis maneras de sacar un 7, 8, 9, 10 y 11. Por lo

tanto, estos son los resultados más comunes y es igualmente probable que saque cualquiera de estos números. Por lo tanto, estos son los resultados más comunes, y tienes las mismas probabilidades de sacar cualquiera de estos números.

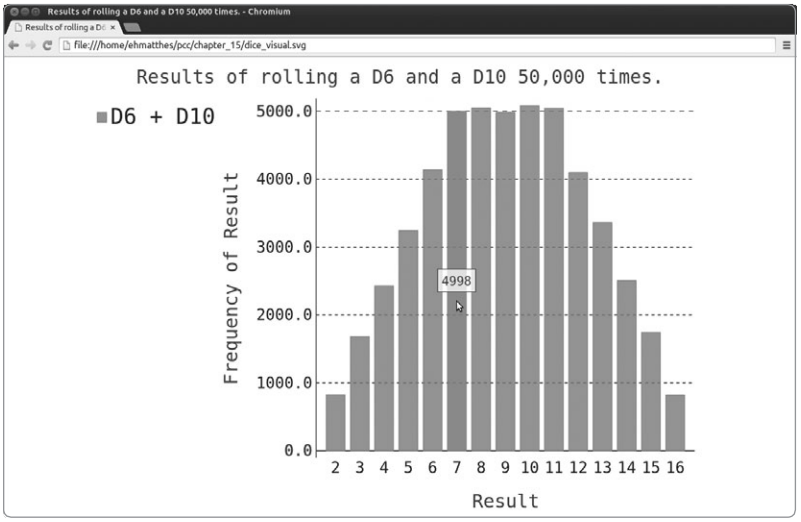


Figura 15-13: Resultados de lanzar un dado de seis caras y otro de diez caras 50.000 veces

Nuestra capacidad de utilizar Pygal para modelar la tirada de dados nos da una libertad considerable para explorar este fenómeno. En cuestión de minutos puedes simular un gran número de tiradas utilizando una gran variedad de dados.

TRY IT yOURSELF

15-6. Etiquetas automáticas: Modifique *die.py* y dice_visual.*py* reemplazando la lista que usamos para establecer el valor de `hist.x_labels` con un bucle para generar esta lista auto-máticamente. Si se siente cómodo con las comprensiones de listas, intente reemplazar los otros bucles `for` en *die_visual.py* y *dice_visual.py* con comprensiones también.

15-7. Dos D8: Crea una simulación que muestre lo que ocurre si lanzas dos dados de ocho caras 1000 veces. Aumenta el número de tiradas gradualmente hasta que empieces a ver los límites de las capacidades de tu sistema.

15-8. Tres dados: Si tiras tres dados D6, el número más pequeño que puedes sacar es 3 y el número más grande es 18. Crea una visualización que muestre lo que ocurre cuando tiras tres dados D6.

15-9. Multiplicación: Cuando tiras dos dados, normalmente sumas los dos números para obtener el resultado. Crea una visualización que muestre qué ocurre si multiplicas estos números.

15-10. Practicando con ambas bibliotecas: Intenta usar matplotlib para hacer una visualización de lanzamiento de dados, y usa Pygal para hacer la visualización de un paseo aleatorio.

resumen

En este capítulo aprendiste a generar conjuntos de datos y a crear visualizaciones de esos datos. Aprendiste a crear gráficos simples con matplotlib, y viste cómo usar un gráfico de dispersión para explorar caminatas aleatorias. Aprendiste a crear un histograma con Pygal y cómo usar un histograma para explorar los resultados de lanzar dados de diferentes tamaños.

Generar tus propios conjuntos de datos con código es una forma interesante y p o t e n t e de modelar y explorar una amplia variedad de situaciones del mundo real. A medida que continúes trabajando en los proyectos de visualización de datos que siguen, mantente atento a las situaciones que podrías modelar con código. Fíjate en las visualizaciones que ves en los medios de comunicación y comprueba si puedes identificar las que se generaron utilizando métodos similares a los que estás aprendiendo en estos proyectos.

En el capítulo 16 descargaremos datos de fuentes en línea y seguiremos utilizando matplotlib y Pygal para explorar esos datos.

16

DOWNLOADING DATA



En este capítulo descargarás conjuntos de datos de fuentes online y crearás visualizaciones funcionales de esos datos. Una increíble

En Internet se puede encontrar una gran variedad de datos, muchos de los cuales no se han examinado a fondo. La capacidad de analizar estos datos permite descubrir patrones y conexiones que nadie más ha encontrado.

Accederemos y visualizaremos datos almacenados en dos formatos de datos comunes, CSV y JSON. Utilizaremos el módulo `csv` de Python para procesar datos meteorológicos almacenados en formato CSV (valores separados por comas) y analizar las temperaturas altas y bajas a lo largo del tiempo en dos ubicaciones diferentes. A continuación, utilizaremos matplotlib para generar un gráfico basado en nuestros datos descargados para mostrar

variaciones de temperatura en dos entornos muy diferentes: Sitka, Alaska, y Death Valley, California. Más adelante, utilizaremos el módulo `json` para acceder a datos de población almacenados en formato JSON y utilizaremos Pygal para dibujar un mapa de población por países.

Al final de este capítulo, estarás preparado para trabajar con diferentes tipos y formatos de conjuntos de datos, y tendrás una comprensión más profunda de cómo construir visualizaciones complejas. La capacidad de acceder y visualizar datos en línea de diferentes tipos y formatos es esencial para trabajar con una amplia variedad de conjuntos de datos del mundo real.

El formato de archivo CSV

Una forma sencilla de almacenar datos en un archivo de texto es escribir los datos como una serie de valores separados por comas, denominados *valores separados por comas*. Los archivos resultantes se denominan archivos CSV. Por ejemplo, aquí tienes una línea de datos meteorológicos en formato CSV:

```
2014-1-5,61,44,26,18,7,-1,56,30,9,30.34,30.27,30.15,,,,10,4,,0.00,0,,195
```

Estos son los datos meteorológicos del 5 de enero de 2014 en Sitka, Alaska. Incluye las temperaturas máximas y mínimas del día, así como otras mediciones de ese día. Los archivos CSV pueden ser difíciles de leer para los humanos, pero son fáciles de procesar y extraer valores para los programas, lo que acelera el proceso de análisis de datos.

Comenzaremos con un pequeño conjunto de datos meteorológicos en formato CSV registrados en Sitka, que están disponibles en los recursos del libro a través de *https://www.nostarch.com/pythoncrashcourse/*. Copia el archivo *sitka_weather_07-2014.csv* a la carpeta donde estás escribiendo los programas de este capítulo. (Una vez que descargues los recursos del libro, tendrás todos los archivos que necesitas para este proyecto).

NOTE *Los datos meteorológicos de este proyecto se descargaron originalmente de* http://www.wunderground.com/historia/.

Análisis de las cabeceras del archivo CSV

El módulo `csv` de la biblioteca estándar de Python analiza las líneas de un archivo CSV y nos permite extraer rápidamente los valores que nos interesan. Empecemos examinando la primera línea del archivo, que contiene una serie de encabezados para los datos:

altas_bajas.py

```
importar csv

filename = 'sitka_weather_07-2014.csv'
❶ con open(nombrearchivo) como f:
❷     reader = csv.reader(f)
❸     fila_cabecera =
```

```
siguiente(lector)
print(fila_cabecera)
```

Después de importar el módulo csv, almacenamos el nombre del archivo con el que estamos trabajando en filename. A continuación, abrimos el archivo y almacenamos el objeto de archivo resultante en f ❶. A continuación, llamamos a csv.reader() y le pasamos el objeto archivo como un para crear un objeto lector asociado a ese fichero ❷. Almacenamos el objeto lector en reader.

El módulo csv contiene una función next(), que devuelve la siguiente línea del fichero cuando se le pasa el objeto reader. En el listado anterior llamamos a next() sólo una vez, de modo que obtenemos la primera línea del fichero, que contiene las cabeceras del fichero ❸. Almacenamos los datos devueltos en header_row. Como puede ver, header_row contiene encabezados significativos relacionados con el tiempo que nos dicen qué información contiene cada línea de datos:

```
['AKDT', 'TemperaturaF máxima', 'TemperaturaF media', 'TemperaturaF
mínima', 'Punto de rocíoF máximo', 'Punto de rocíoF medio', 'Punto de
rocíoF mínimo', 'HumedadF máxima', ' HumedadF media', ' HumedadF
mínima', ' PresiónIn máxima a nivel del mar',
Presión media a nivel del mar', 'Presión mínima a nivel del mar',
Max VisibilityMiles', ' Mean VisibilityMiles', ' Min VisibilityMiles', '
Max Wind SpeedMPH', ' Mean Wind SpeedMPH', ' Max Gust SpeedMPH',
'PrecipitationIn', ' CloudCover', ' Events', ' WindDirDegrees'].
```

reader procesa la primera línea de valores separados por comas del fichero y almacena cada uno de ellos como un elemento de una lista. La cabecera AKDT representa el horario de verano de Alaska. La posición de esta cabecera nos indica que el primer valor de cada línea será la fecha o la hora. La cabecera Max TemperatureF nos dice que el segundo valor de cada línea es la temperatura máxima para esa fecha, en grados Fahrenheit. Puede leer el resto de las cabeceras para determinar el tipo de información incluida en el fichero.

NOTE *Las cabeceras no siempre tienen el mismo formato: los espacios y las unidades están en lugares extraños. Esto es habitual en los archivos de datos sin procesar, pero no causará ningún problema.*

Impresión de las cabeceras y sus posiciones

Para facilitar la comprensión de los datos de cabecera del fichero, imprima cada cabecera y su posición en la lista:

altos_bajos.py

```
--snip--
with open(nombre_archivo) as
    f: reader = csv.reader(f)
    header_row = next(reader)

❶   for índice, encabezado_columna in
        enumerar(fila_cabecera): print(índice,
```

```
        encabezado_columna)
```

Utilizamos `enumerate()` ❶ en la lista para obtener el índice de cada elemento, así como el valor. (Tenga en cuenta que hemos eliminado la línea `print(header_row)` en favor de esta versión más detallada).

Aquí está la salida mostrando el índice de cada encabezado:

```
0 AKDT
1 Temperatura máximaF
2 Temperatura mediaF
3 Temperatura mínimaF
--snip--
20  CloudCover
21  Eventos
22  WindDirDegrees
```

Aquí vemos que las fechas y sus altas temperaturas se almacenan en las columnas 0 y 1. Para explorar estos datos, procesaremos cada fila de datos en *sitka_weather_07-2014.csv* y extraeremos los valores con los índices 0 y 1.

Extracción y lectura de datos

Ahora que sabemos qué columnas de datos necesitamos, vamos a leer algunos de esos datos. En primer lugar, leeremos la temperatura máxima de cada día:

altas_bajas.py

```
importar csv

# Obtener las altas temperaturas del
archivo. filename = 'sitka_weather_07-
2014.csv' with open(filename) as f:
    reader = csv.reader(f)
    header_row = next(reader)

❶   altas = []
❷   para fila en lector:
❸       highs.append(fila[1])

    print(máximos)
```

Hacemos una lista vacía llamada `altas` ❶ y luego recorremos en bucle las filas restantes del archivo ❷. El objeto lector continúa desde donde lo dejó en el archivo CSV y devuelve automáticamente cada línea siguiente a su posición actual. Como ya hemos leído la `fila de` cabecera, el bucle comenzará en la segunda línea, donde empiezan los datos reales. En cada pasada por el bucle, añadimos los datos del índice 1, la segunda columna, a `los altos` ❸.

El siguiente listado muestra los datos almacenados ahora en `altas`:

```
['64', '71', '64', '59', '69', '62', '61', '55', '57', '61', '57', '59', '57',
 '61', '64', '61', '59', '63', '60', '57', '69', '63', '62', '59', '57', '57',
 '61', '59', '61', '61', '66']
```

Hemos extraído la temperatura máxima de cada fecha y la hemos almacenado ordenadamente en una lista como cadenas.

A continuación, convierta estas cadenas en números con `int()` para que puedan ser leídas por matplotlib:

altos_bajos.py

```
--snip--
    altas = []
    para fila en lector:
        high =
(1)     int(fila[1])
        highs.append(high)

    print(máximos)
```

Convertimos las cadenas a enteros en (1) antes de añadir las temperaturas a la lista. El resultado es una lista de máximos diarios en formato numérico:

```
[64, 71, 64, 59, 69, 62, 61, 55, 57, 61, 57, 59, 57, 61, 64, 61, 59, 63, 60,
57,
 69, 63, 62, 59, 57, 57, 61, 59, 61, 61, 66]
```

Ahora vamos a crear una visualización de estos datos.

Trazado de datos en un gráfico de temperatura

Para visualizar los datos de temperatura que tenemos, primero crearemos un simple gráfico de las máximas diarias utilizando matplotlib, como se muestra aquí:

altas_bajas.py

```
importar csv

from matplotlib import pyplot as plt

# Obtener altas temperaturas del

fichero.
--snip--

# Plot data.
fig = plt.figure(dpi=128, figsize=(10, 6))
(1) plt.plot(altos, c='rojo')

# Format plot.
(2) plt.title("Temperaturas máximas diarias, julio de 2014", fontsize=24)
(3) plt.xlabel('', fontsize=16)
plt.ylabel("Temperatura (F)", fontsize=16)
plt.tick_params(axis='both', which='major', labelize=16)

plt.show()
```

Pasamos la lista de máximos a `plot()` (1) y pasamos `c='red'` para trazar los puntos en rojo. (Trazaremos los máximos en rojo y los mínimos en azul.) A continuación, especificamos algunos otros detalles de formato, como el

tamaño de la fuente y las etiquetas ❷, que deberías reconocer del capítulo 15. Porque aún tenemos que añadir las fechas, no etiquetaremos el eje x, pero `plt.xlabel()` modifica el tamaño de la fuente para hacer las etiquetas por defecto más legibles ❸. La Figura 16-1 muestra el gráfico resultante: un simple gráfico de líneas de las altas temperaturas de julio de 2014, en Sitka, Alaska.

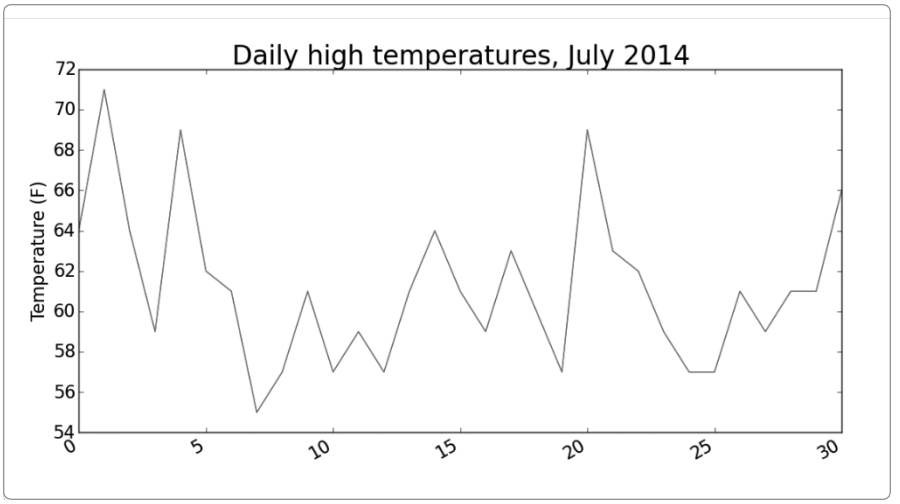


Figura 16-1: Gráfico lineal que muestra las temperaturas máximas diarias de julio de 2014 en Sitka, Alaska.

El módulo datetime

Añadamos fechas a nuestro gráfico para hacerlo más útil. La primera fecha del fichero de datos meteorológicos está en la segunda fila del fichero:

```
2014-7-1,64,56,50,53,51,48,96,83,58,30.19,--snip--
```

Los datos se leerán como una cadena, por lo que necesitamos una forma de convertir la cadena `'2014-7-1'` en un objeto que represente esta fecha. Podemos construir un objeto que represente el 1 de julio de 2014, utilizando el método strptime() del módulo `datetime`. Veamos cómo funciona strptime() en una sesión de terminal:

```
>>> from datetime import datetime
>>> first_date = datetime.strptime('2014-7-1', '%Y-%m-%d')
>>> print(primera_fecha)
2014-07-01 00:00:00
```

Primero importamos la clase `datetime` del módulo `datetime`. Luego llamamos al método `strptime()` con la cadena que contiene la fecha con la que queremos trabajar como primer argumento. El segundo argumento le dice a Python cómo se formatea la fecha. En este ejemplo, `'%Y-'` le dice a Python que interprete la parte de la cadena antes del primer guión como un año de cuatro dígitos; `'%m-'` le dice a Python que interprete la parte de la cadena antes del segundo guión como un número que representa el mes; y `'%d'` le dice a Python que interprete la última parte de la cadena como el día del mes, del 1 al 31.

El método `strptime()` puede tomar una variedad de argumentos para determinar cómo interpretar la fecha. La Tabla 16-1 muestra algunos de estos argumentos.

Tabla 16-1: Argumentos de formato de fecha y hora del módulo `datetime`

Argumento	Significado
`%A`	Nombre del día de la semana, como *lunes*
`%B`	Nombre del mes, por ejemplo *enero*
`%m`	Mes, como número (01 a 12)
`%d`	Día del mes, como número (01 a 31)
`%Y`	Año de cuatro dígitos, como 2015
`%y`	Año de dos dígitos, como 15
`%H`	Hora, en formato de 24 horas (00 a 23)
`%I`	Hora, en formato de 12 horas (01 a 12)
`%p`	AM o PM
`%M`	Minutos (00 a 59)
`%S`	Segundos (00 a 61)

Trazado de fechas

Sabiendo cómo procesar las fechas en nuestro archivo CSV, ahora podemos mejorar nuestro gráfico de los datos de temperatura extrayendo fechas para los máximos diarios y pasando las fechas y los máximos a `plot()`, como se muestra aquí:

altas_bajas.py

```
importar csv
from datetime import datetime

from matplotlib import pyplot as plt

# Obtener fechas y temperaturas altas del
archivo. filename = 'sitka_weather_07-
2014.csv'
with open(nombre_archivo) as
    f: reader = csv.reader(f)
    header_row = next(reader)

❶   dates, highs = [], []
    for row in reader:
❷       fecha_actual = datetime.strptime(fila[0], "%A-%m-
        %d") fechas.append(fecha_actual)

        alto = int(fila[1])
        altos.append(alto)

# Plot data.
fig = plt.figure(dpi=128, figsize=(10, 6))
❸ plt.plot(fechas, máximos, c='rojo')

# Format plot.
plt.title("Temperaturas máximas diarias, julio de 2014", fontsize=24)
```

```
plt.xlabel('', fontsize=16)
```

```
❹ fig.autofmt_xdate()
  plt.ylabel("Temperatura (F)", fontsize=16)
  plt.tick_params(axis='both', which='major', labelize=16)

  plt.show()
```

Creamos dos listas vacías para almacenar las fechas y las altas temperaturas del fichero ❶. A continuación, convertimos los datos que contienen la información de la fecha (`fila[0]`) en un objeto `datetime` ❷ y lo añadimos a las `fechas`. Pasamos las fechas y los valores de temperatura alta a `plot()` en ❸. La llamada a `fig.autofmt_xdate()` en ❹ dibuja las etiquetas de fecha en diagonal para evitar que se solapen.
La Figura 16-2 muestra el gráfico mejorado.

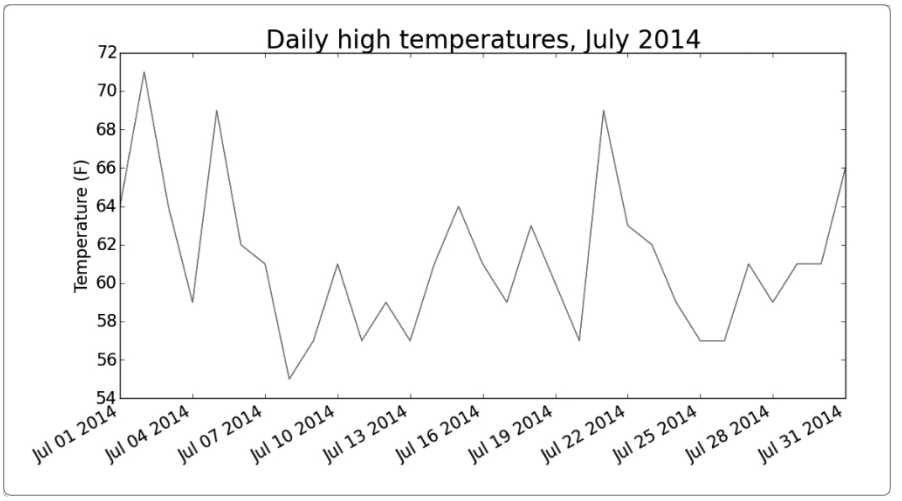


Figura 16-2: El gráfico tiene más sentido ahora que tiene fechas en el eje x.

Trazar un plazo más largo

Con nuestro gráfico configurado, vamos a añadir más datos para obtener una imagen más completa del tiempo en Sitka. Copia el archivo *sitka_weather_2014.csv*, que contiene un año completo de datos de Weather Underground para Sitka, a la carpeta donde guardas los programas de este capítulo.

Ahora podemos generar un gráfico para el tiempo de todo el año:

altos_bajos.py

```
  --snip--
  # Obtener fechas y temperaturas altas del archivo.
❶ filename = 'sitka_weather_2014.csv'
  with open(filename) as f:
  --snip--
  # Format plot.
❷ plt.title("Temperaturas máximas diarias - 2014",
  fontsize=24) plt.xlabel('', fontsize=16)
```

```
--snip--
```

Modificamos el nombre del archivo para utilizar el nuevo archivo de datos *sitka_weather_2014.csv* ❶, y actualizamos el título de nuestro gráfico para reflejar el cambio en su contenido ❷. La Figura 16-3 muestra el gráfico resultante.

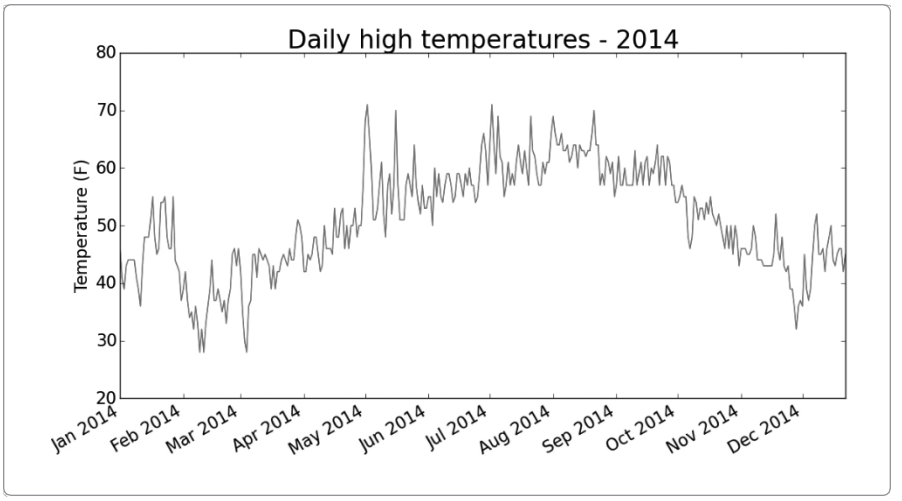


Figura 16-3: Datos de un año

Trazado de una segunda serie de datos

El gráfico reelaborado de la Figura 16-3 muestra una cantidad considerable de datos medios, pero podemos hacerlo aún más útil incluyendo las temperaturas bajas. Tenemos que extraer las temperaturas bajas del archivo de datos y luego añadirlas a nuestro gráfico, como se muestra aquí:

altos_bajos.py

```
--snip--
# Obtener fechas, temperaturas altas y bajas del
archivo. filename = 'sitka_weather_2014.csv'
with open(nombre_archivo) as
    f: reader = csv.reader(f)
    header_row = next(reader)

❶   fechas, máximos, mínimos = [],
    [], [] for fila in lector:
        fecha_actual = datetime.strptime(fila[0], "%A-%m-%d")
        fechas.append(fecha_actual)

        alto = int(fila[1])
        altos.append(alto)

❷       low =
        int(fila[3])
        lows.append(low)
```

```
    # Plot data.
    fig = plt.figure(dpi=128, figsize=(10, 6))
    plt.plot(fechas, máximos, c='rojo')
❸ plt.plot(fechas, mínimos, c='azul')

    # Format plot.
❹ plt.title("Temperaturas máximas y mínimas diarias - 2014", fontsize=24)
    --snip--
```

En ❶ añadimos la lista vacía `lows` para mantener las temperaturas bajas, y luego extraemos y almacenamos la temperatura baja para cada fecha, desde la cuarta posición en cada fila (`fila[3]`) ❷. En ❸ añadimos una llamada a `plot()` para las temperaturas bajas y coloreamos estos valores en azul. Finalmente, actualizamos el título ❹. La Figura 16-4 muestra el gráfico resultante.

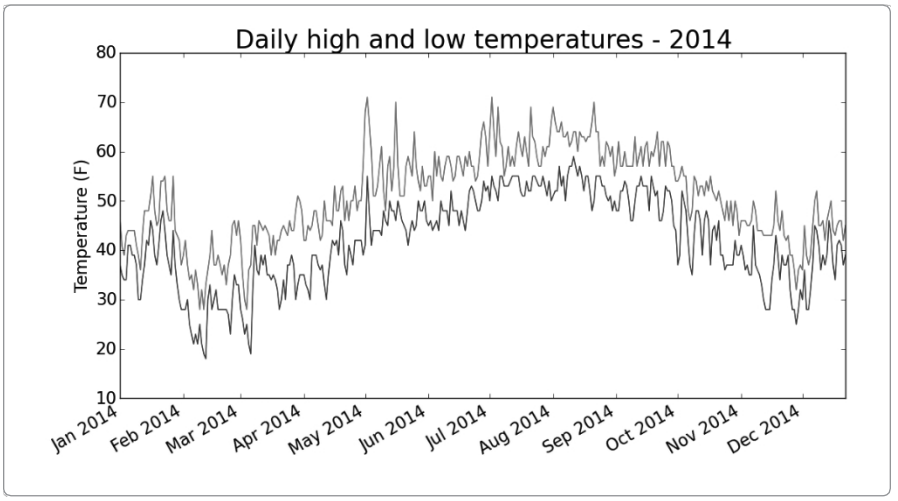


Figura 16-4: Dos series de datos en el mismo gráfico

Sombrear un área del gráfico

Una vez añadidas las dos series de datos, podemos examinar el rango de temperaturas de cada día. Vamos a dar un toque final al gráfico utilizando sombreado para mostrar el rango entre las temperaturas altas y bajas de cada día. Para ello, utilizaremos el método `fill_between()`, que toma una serie de valores x y dos series de valores y, y rellena el espacio entre las dos series de valores y:

altos_bajos.py

```
    --snip--
    # Plot data.
    fig = plt.figure(dpi=128, figsize=(10, 6))
❶ plt.plot(fechas, máximos, c='rojo', alpha=0.5)
    plt.plot(fechas, mínimos, c='azul', alpha=0.5)
```

```
❷ plt.fill_between(fechas, máximos, mínimos, facecolor='azul', alpha=0.1)
--snip--
```

El argumento `alfa` en ❶ controla la transparencia de un color. Un valor `alfa` de 0 es completamente transparente, y 1 (por defecto) es completamente opaco. Ajustando `alfa` a 0.5 hacemos que las líneas de trazado rojas y azules aparezcan más claras.

En ❷ pasamos a `fill_between()` las `fechas de` la lista para los valores x y luego las dos series de valores y `altos` y `bajos`. El argumento `facecolor` determina el color de la región sombreada, y le damos un valor `alfa` bajo de 0.1 para que la región rellena conecte las dos series de datos sin distraer de la información que representan. La Figura 16-5 muestra el gráfico con la región sombreada entre los máximos y los mínimos.

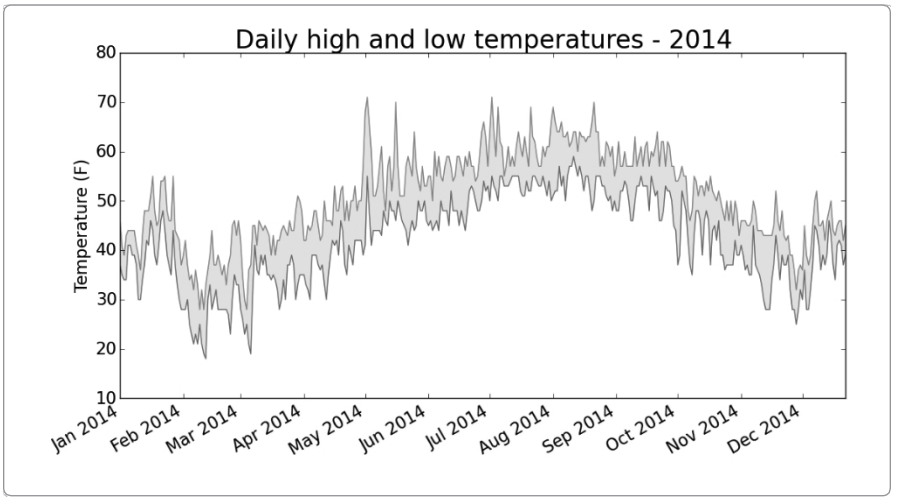


Figura 16-5: La región entre los dos conjuntos de datos está sombreada.

El sombreado ayuda a hacer inmediatamente aparente el rango entre los dos conjuntos de datos.

Comprobación de errores

Deberíamos poder ejecutar el código de *highs_lows.py* con los datos de cualquier lugar. Pero algunas estaciones meteorológicas funcionan mal de vez en cuando y no recogen algunos o todos los datos que se supone que deben recoger. La falta de datos puede dar lugar a excepciones que bloqueen nuestros programas si no las manejamos adecuadamente.

Por ejemplo, veamos qué ocurre cuando intentamos generar un gráfico de temperatura para Death Valley, California. Copia el archivo *death_valley_2014.csv* en la carpeta donde guardas los programas de este capítulo y, a continuación, modifica *highs_lows.py* para generar un gráfico para Death Valley:

altos_bajos.py

```
--snip--
# Obtener fechas, temperaturas altas y bajas del
archivo. filename = 'death_valley_2014.csv'
con open(nombrearchivo) como f:
--snip--
```

Cuando ejecutamos el programa obtenemos un error, como se muestra en la última línea de la siguiente salida:

```
Traceback (última llamada más reciente):
  File "altos_bajos.py", line 17, in <module>
    altos = int(fila[1])
ValueError: literal inválido para int() con base 10: ''
```

El rastreo nos dice que Python no puede procesar la temperatura alta para una de las fechas porque no puede convertir una cadena vacía (`''`) en un entero. Un vistazo a *death_valley_2014.csv* muestra el problema:

```
2014-2-16,,,,,,,,,,,,,,,,,,,,,0.00,,,-1
```

Parece que el 16 de febrero de 2014 no se registró ningún dato; la cadena correspondiente a la temperatura máxima está vacía. Para solucionar este problema, ejecutaremos un código de comprobación de errores cuando se lean los valores del archivo CSV para gestionar las excepciones que puedan surgir al analizar nuestros conjuntos de datos. Así es como funciona:

altos_bajos.py

```
--snip--
# Obtener fechas, temperaturas altas y bajas del
archivo. filename = 'death_valley_2014.csv'
with open(nombre_archivo) as
    f: reader = csv.reader(f)
    header_row = next(reader)

    fechas, máximos, mínimos = [],
    [], [] for fila en lector:
❶       probar:
            fecha_actual = datetime.strptime(fila[0], "%A-%m-%d")
            alta = int(fila[1])
            low = int(fila[3])
        except ValueError:
❷           print(fecha_actual, 'faltan datos')
        else:
❸           dates.append(fecha_actual)
            highs.append(high)
            lows.append(low)

# Plot data.
--snip--

# Format plot.
❹ title = "Temperaturas máximas y mínimas diarias - 2014\nDeath Valley, CA"
plt.title(title, fontsize=20)
--snip--
```

Cada vez que examinamos una fila, intentamos extraer la fecha y la temperatura alta y baja ❶. Si falta algún dato, Python lanzará un error `ValueError` y lo manejamos imprimiendo un mensaje de error que incluye la fecha del dato que falta ❷. Después de imprimir el error, el bucle continuará procesando la siguiente fila. Si todos los datos de una fecha se recuperan sin error, se ejecutará el bloque `else` y los datos se añadirán a las listas correspondientes ❸. Como estamos trazando información para una nueva localización, actualizamos el título para incluir la localización en el gráfico ❹.

Cuando ejecute ahora *highs_lows.py*, verá que sólo una fecha tenía datos erróneos:

```
2014-02-16 datos que faltan
```

La Figura 16-6 muestra el gráfico resultante.

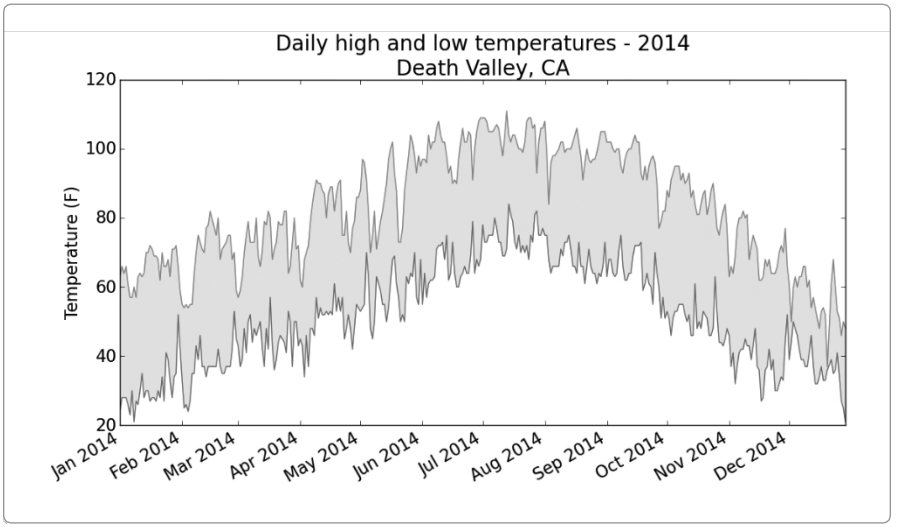


Figura 16-6: Temperaturas máximas y mínimas diarias en el Valle de la Muerte

Comparando este gráfico con el de Sitka, podemos ver que el Valle de la Muerte es más cálido en general que el sureste de Alaska, como cabría esperar, pero también que el rango de temperaturas de cada día es en realidad mayor en el desierto. La altura de la región sombreada lo pone de manifiesto.

Muchos de los conjuntos de datos con los que trabaje tendrán datos que faltan, datos con formato inadecuado o datos incorrectos. Utilice las herramientas que aprendió en la primera mitad de este libro para hacer frente a estas situaciones. Aquí usamos un bloque `try-except-else` para manejar los datos faltantes. A veces usará `continue` para saltarse algunos datos o usará `remove()` o `del` para eliminar algunos datos después de que hayan sido

extraídos. Puede utilizar cualquier método que funcione, siempre que el resultado sea una visualización significativa y precisa.

TRY IT yOURSELF

16-1. San Francisco: ¿Son las temperaturas de San Francisco más parecidas a las de Sitka o a las del Valle de la Muerte? Genera un gráfico de temperaturas altas y bajas para San Francisco y haz una comparación. (Puedes descargar datos meteorológicos de casi cualquier lugar en http://www.wunderground.com/history/.)
Introduzca una ubicación y un intervalo de fechas, desplácese hasta el final de la página y busque un enlace llamado *Archivo delimitado por comas*. Haga clic con el botón derecho en este enlace y guarde los datos como archivo CSV).

16-2. Comparación Sitka-Valle de la Muerte: Las escalas de temperatura en las gráficas de Sitka y del Valle de la Muerte reflejan los diferentes rangos de los datos. Para comparar correctamente el rango de temperatura en Sitka con el del Valle de la Muerte, necesita escalas idénticas en el eje y. Cambie los ajustes para el eje y en uno o ambos gráficos. Cambie los ajustes para el eje y en uno o ambos gráficos de las Figuras 16-5 y 16-6, y haga una comparación directa entre los rangos de temperatura en Sitka y el Valle de la Muerte (o cualquier otro lugar que desee comparar). También puede intentar trazar los dos conjuntos de datos en el mismo gráfico.

16-3. Precipitaciones: Elige cualquier lugar que te interese y crea una visualización que represente sus precipitaciones. Empieza por centrarte en los datos de un mes y, una vez que tu código funcione, ejecútalo para los datos de todo un año.

16-4. Explorar: Genere algunas visualizaciones más que examinen cualquier otro aspecto meteorológico que le interese para cualquier lugar que le resulte curioso.

Asignación de conjuntos de datos globales: Formato JSON

En esta sección, descargarás datos de países basados en su localización en formato JSON y trabajarás con ellos utilizando el módulo `json`. Utilizando la herramienta de mapeo de Pygal para principiantes, crearás visualizaciones de estos datos que exploran patrones globales relativos a la distribución de la población mundial en diferentes países.

Descargar datos sobre la población mundial

Copie el archivo *population_data.json*, que contiene datos de población de 1960 a 2010 para la mayoría de los países del mundo, en la carpeta donde almacena los programas de este capítulo. Estos datos proceden de uno de los muchos conjuntos de datos que la Open Knowledge Foundation *(http://data.okfn.org/)* pone a libre disposición.

Extracción de datos pertinentes

Echemos un vistazo a *population_data.json* para ver cómo podemos empezar a procesar los datos del archivo:

población_datos.json

```
[
  {
    "Country Name": "Mundo
    Árabe", "Código de País":
    "ARB", "Año": "1960",
    "Valor": "96388069"
  },
  {
    "Country Name": "Mundo
    Árabe", "Código de País":
    "ARB", "Año": "1961",
    "Valor": "98882541.4"
  },
  --snip--
]
```

El fichero es básicamente una larga lista de Python. Cada elemento es un diccionario con cuatro claves: un nombre de país, un código de país, un año y un valor que representa la población. Queremos examinar el nombre y la población de cada país sólo en 2010, así que empieza por escribir un programa que imprima sólo esa información:

mundo_población.py

```
importar json

# Cargar los datos en una lista.
filename = 'population_data.json'
with open(filename) as f:
❶    pop_data = json.load(f)

# Imprime la población de 2010 para cada país.
❷ for pop_dict in pop_data:
❸     if pop_dict['Año'] == '2010':
❹         nombre_país = pop_dict['Nombre del
          país'] población = pop_dict['Valor']
          print(nombre_país + ": " + población)
```

Primero importamos el módulo `json` para poder cargar los datos correctamente desde el archivo, y luego almacenamos los datos en `pop_data` en ❶. La función `json.load()` convierte los datos a un formato con el que Python pueda trabajar: en este caso, una lista. En ❷ recorremos en bucle cada elemento de pop_data. Cada elemento es un diccionario con cuatro pares clave-valor, y almacenamos cada diccionario en `pop_dict`.

En ❸ buscamos 2010 en la clave `'Año'` de cada diccionario. (Como los valores de *population_data.json* están todos entre comillas, hacemos una comparación de cadenas). Si el año es 2010, almacenamos el valor asociado a 'Nombre del `país'` en `nombre_país` y el valor asociado a `'Valor'` en

`población` en ❹. A continuación, imprimimos el nombre de cada país y su población.

El resultado es una serie de nombres de países y valores de población:

```
Mundo árabe: 357868000
Pequeños Estados del Caribe: 6880000
Asia Oriental y Pacífico (todos los niveles de renta): 2201536674
--snip--
Zimbabue: 12571000
```

No todos los datos capturados incluyen los nombres exactos de los países, pero es un buen comienzo. Ahora tenemos que convertir los datos a un formato con el que Pygal pueda trabajar.

Conversión de cadenas en valores numéricos

Cada clave y valor de *population_data.json* se almacena como una cadena. Para trabajar con los datos de población, tenemos que convertir las cadenas de población en valores numéricos. Para ello utilizamos la función `int()`:

mundo_población.py

```
--snip--
para pop_dict en pop_data:
    si pop_dict['Año'] == '2010':
        nombre_país = pop_dict['Nombre del país']
❶       población = int(pop_dict['Valor'])
❷       print(nombre_país + ": " + str(población))
```

Ahora hemos almacenado cada valor de población en formato numérico en ❶.

Cuando imprimimos el valor de la población, tenemos que convertirlo a cadena en ❷. Sin embargo, este cambio da lugar a un error para algunos valores, como se muestra aquí:

```
Mundo árabe: 357868000
Pequeños Estados del Caribe: 6880000
Asia Oriental y Pacífico (todos los niveles de renta): 2201536674
 --snip--
Traceback (última llamada más reciente):
  File "print_populations.py", line 12, in <module>
    population = int(pop_dict['Valor'])
❶ ValueError: literal no válido para int() con base 10: '1127437398.85751'
```

A menudo ocurre que los datos en bruto no tienen un formato consistente, por lo que nos encontramos con errores a menudo. Aquí el error se produce porque Python no puede convertir directamente una cadena que contiene un decimal, `'1127437398.85751'`, en un entero ❶. (Este valor decimal es probablemente el resultado de la interpolación de años en los que no se hizo un recuento específico de la población). Solucionamos este error convirtiendo la cadena a un flotante y luego convirtiendo ese flotante a un entero:

*mundo_*

```
--snip--
```

población.py

```
para pop_dict en pop_data:
    si pop_dict['Año'] == '2010':
        país = pop_dict['Nombre del país'] población
        = int(float(pop_dict['Valor'])) print(país
        + ": " + str(población))
```

La función `float()` convierte la cadena en un decimal, y la función `int()` elimina la parte decimal del número y devuelve un entero. Ahora podemos imprimir un conjunto completo de valores de población para el año 2010 sin errores:

```
Mundo árabe: 357868000
Pequeños Estados del Caribe: 6880000
Asia Oriental y Pacífico (todos los niveles de renta): 2201536674
--snip--
Zimbabue: 12571000
```

Cada cadena se ha convertido correctamente en un flotador y luego en un número entero. Ahora que estos valores de población están almacenados en un formato numérico, podemos utilizarlos para hacer un mapa de la población mundial.

Obtención de códigos de país de dos cifras

Antes de centrarnos en la cartografía, tenemos que abordar un último aspecto de los datos. La herramienta de mapeo de Pygal espera datos en un formato particular: los países deben proporcionarse como códigos de país y las poblaciones como valores. Los códigos incluidos en *population_data.json* son de tres letras, pero Pygal utiliza códigos de dos letras. Necesitamos una forma de encontrar el código de dos dígitos a partir del nombre del país.

Los códigos de país de Pygal se almacenan en un módulo llamado `i18n`, abreviatura de *internacionalización*. El diccionario `COUNTRIES` contiene los códigos de país de dos letras como claves y los nombres de los países como valores. Para ver estos códigos, importa el diccionario del módulo `i18n` e imprime sus claves y valores:

países.py

```
from pygal.i18n import PAÍSES

❶ for código_país in ordenados(COUNTRIES.keys()):
    print(código_país, COUNTRIES[código_país])
```

En el bucle `for` le decimos a Python que ordene las claves por orden alfabético ❶. A continuación, imprimimos cada código de país y el país al que está asociado:

```
ad Andorra
ae Emiratos Árabes
Unidos af Afganistán
--snip--
zw Zimbabue
```

Para extraer los datos del código de país, escribimos una función que busca en `COUNTRIES` y devuelve el código de país. Escribiremos esto en un módulo separado llamado `country_codes` para poder importarlo más tarde en un programa de visualización:

país_códigos.py

```
from pygal.i18n import PAÍSES

❶ def get_codigo_pais(nombre_pais):
    """Devuelve el código de país Pygal de 2 dígitos para el país dado."""
```

```
❷     para código, nombre en PAÍSES.elementos():
❸         si nombre ==
              nombre_país:
              devolver código
      # Si no se encuentra el país, devuelve None.
❹     devolver Ninguno

print(get_country_code('Andorra'))
print(get_country_code('Emiratos Árabes Unidos'))
print(get_country_code('Afganistán'))
```

Pasamos a `get_country_code()` el nombre del país y lo almacenamos en el parámetro `country_name` ❶. A continuación, recorremos en bucle los pares código-nombre en `COUNTRIES` ❷. Si se encuentra el nombre del país, se devuelve el código del país ❸. Añadimos una línea después del bucle para devolver `None` si no se ha encontrado el nombre del país ❹. Por último, pasamos tres nombres de países para comprobar que la función funciona. Como era de esperar, el programa devuelve tres códigos de país de dos letras:

```
ad
ae
af
```

Antes de utilizar esta función, elimine las sentencias `print` de *codigos_pais.py*.

A continuación importamos `get_country_code()` en *world_population.py*:

mundo_población.py

```
importar json

from codigos_paises import get_codigo_pais
--snip--

# Imprime la población de 2010 de cada país.
for pop_dict in pop_data:
    si pop_dict['Año'] == '2010':
        nombre_país = pop_dict['Nombre del país']
        población = int(float(pop_dict['Valor']))
❶       code = get_country_code(nombre_país)
        if code:
❷           print(código + ": "+ str(población))
❸       más:
            print('ERROR - ' + nombre_país)
```

Tras extraer el nombre del país y la población, almacenamos el código del país en `code` o `None` si no hay ningún código disponible ❶. Si se devuelve un código, se imprimen el código y la población del país ❷. Si el código no está disponible, mostramos un mensaje de error con el nombre

del país para el que no podemos encontrar un código ❸. Ejecuta este programa y verás algunos códigos de países con sus poblaciones y algunas líneas de error:

```
ERROR - Mundo árabe
ERROR - Pequeños Estados del Caribe
ERROR - Asia Oriental y Pacífico (todos los niveles de renta)
```

```
--snip--
af: 34385000
al: 3205000
dz: 35468000
--snip--
ERROR - Yemen, Rep.
zm: 12927000
zw: 12571000
```

Los errores proceden de dos fuentes. En primer lugar, no todas las clasificaciones del conjunto de datos son por países; algunas estadísticas de población corresponden a regiones (*Mundo Árabe*) y grupos económicos (*todos los niveles de renta*). En segundo lugar, algunas de las estadísticas utilizan un sistema diferente para los nombres completos de los países (*Yemen, Rep.* en lugar de *Yemen*). Por ahora, omitiremos los datos de países que causan errores y veremos qué aspecto tiene nuestro mapa con los datos que hemos recuperado correctamente.

Construir un mapa del mundo

Con los códigos de países que tenemos, es rápido y sencillo hacer un mapa del mundo. Pygal incluye un tipo de gráfico Worldmap para ayudar a mapear conjuntos de datos globales. Como ejemplo de cómo usar `Worldmap`, crearemos un mapa simple que ilumina Norteamérica, Centroamérica y Sudamérica:

americas.py

```
importar pygal

❶ wm = pygal.Worldmap()
  wm.title = "América del Norte, Central y del Sur

❷ wm.add('Norteamérica', ['ca', 'mx', 'us'])
  wm.add('América Central', ['bz', 'cr', 'gt', 'hn', 'ni', 'pa', 'sv'])
  wm.add('Sudamérica', ['ar', 'bo', 'br', 'cl', 'co', 'ec', 'gf',
      gy", "pe", "py", "sr", "uy", "ve"])

❸  wm.render_to_file('americas.svg')
```

En ❶ creamos una instancia de la clase `Worldmap` y establecemos el atributo `title` del mapa. En ❷ utilizamos el método `add()`, que recibe una etiqueta y una lista de códigos de los países en los que queremos centrarnos. Cada llamada a `add()` establece un nuevo color para el conjunto de países y añade ese color a una clave a la izquierda del gráfico con la etiqueta especificada aquí. Queremos que toda la región de Norteamérica esté representada en un color, así que colocamos `'ca'`, `'mx'` y `'us'` en la lista que pasamos a la primera llamada a `add()` para resaltar Canadá, México y Estados Unidos juntos. A continuación, hacemos lo mismo con los países de Centroamérica y Sudamérica.

El método `render_to_file()` en ❸ crea un archivo *.svg* que contiene el gráfico, el cual puede abrir en su navegador. El resultado es un mapa que resalta Norteamérica, Centroamérica y Sudamérica en diferentes colores,

como se muestra en la Figura 16-7.



Figura 16-7: Un ejemplo sencillo del tipo de gráfico Mapamundi

Ya sabemos cómo hacer un mapa con áreas coloreadas, una clave y etiquetas ordenadas. Vamos a añadir datos a nuestro mapa para mostrar información sobre un país.

Trazado de datos numéricos en un mapamundi

Para practicar el trazado de datos numéricos en un mapa, crea un mapa que muestre las poblaciones de los tres países de Norteamérica:

na_poblaciones.py

```
importar pygal

wm = pygal.MapaMundo()
wm.title = "Poblaciones de los países de América del Norte
❶ wm.add('Norteamérica', {'ca': 34126000, 'us': 309349000, 'mx': 113423000})

wm.render_to_file('na_populations.svg')
```

Primero crea una instancia de `Worldmap` y establece un título. A continuación, utilice el método `add()`, pero esta vez pase un diccionario como segundo argumento en lugar de una lista ❶. El diccionario tiene códigos de país de dos letras de Pygal como claves y números de población como sus valores. Pygal utiliza automáticamente estos números para sombrear los países de claro (menos poblado) a oscuro (más poblado). La Figura 16-8 muestra el mapa resultante.



Figura 16-8: Tamaño de la población de los países de América del Norte

Este mapa es interactivo: si pasas el ratón por encima de cada país, verás su población. Añadamos más datos a nuestro mapa.

Trazado de un mapa de población completo

Para trazar las cifras de población del resto de los países, tenemos que contrastar los datos de los países que hemos procesado antes en el formato de diccionario que Pygal espera, es decir, códigos de país de dos letras como claves y cifras de población como valores. Añade el siguiente código a *población_mundo.py*:

mundo_población.py

```
importar json

importar pygal

from codigos_pais import get_codigo_pais #

Cargar los datos en una lista.
--snip--

# Construir un diccionario de datos de población.
❶ cc_poblaciones = {}
para pop_dict en pop_data:
    si pop_dict['Año'] == '2010':
        country = pop_dict['Nombre del país']
        population = int(float(pop_dict['Valor']))
        code = get_country_code(country)
```

```
        si código:
❷           cc_poblaciones[código] = población

❸ wm = pygal.Worldmap()
  wm.title = "Población mundial en 2010, por países
❹ wm.add('2010', cc_populations)

  wm.render_to_file('población_mundo.svg')
```

Primero importamos `pygal`. En ❶ creamos un diccionario vacío para almacenar los códigos de país y las poblaciones en el formato que Pygal espera. En ❷ construimos el diccionario `cc_populations` utilizando el código de país como clave y la población como valor siempre que se devuelva un código. También eliminamos todas las sentencias `print`.

Creamos una instancia de `Worldmap` y establecemos su atributo `title` ❸. Cuando llamamos a `add()`, le pasamos el diccionario de códigos de país y valores de población ❹. La Figura 16-9 muestra el mapa generado.



Figura 16-9: La población mundial en 2010

No tenemos datos de algunos países, que están coloreados en negro, pero la mayoría de los países están coloreados según el tamaño de su población. Más adelante nos ocuparemos de los datos que faltan, pero antes modificaremos el sombreado para que refleje con mayor precisión la población de los países. Actualmente, nuestro mapa muestra muchos países ligeramente sombreados y dos países muy sombreados. El contraste entre la mayoría de los países no es suficiente para indicar la población de cada uno de ellos. Lo solucionaremos agrupando los países por niveles de población y sombreando cada grupo.

Agrupación de países por población

Como China e India están más poblados que otros países, el mapa muestra poco contraste. China e India tienen cada uno más de mil millones de habitantes, mientras que el siguiente país más poblado es Estados Unidos, con aproximadamente 300 millones de personas. En lugar de representar a todos los países como un solo grupo, dividámoslos en tres niveles de población: menos de 10 millones, entre 10 millones y 1.000 millones, y más de 1.000 millones:

mundo_poblacion.py

```
--snip--
# Construir un diccionario de datos de
población. cc_populations = {}
para pop_dict en pop_data:
    si pop_dict['Año'] == '2010':
        --snip--
        si
        código:
            cc_poblaciones[código] = población

# Agrupa los países en 3 niveles de población.
❶ cc_pops_1, cc_pops_2, cc_pops_3 = {}, {}, {}
❷ for cc, pop in cc_populations.items():
    if pop < 10000000:
        cc_pops_1[cc] = pop
    elif pop < 1000000000:
        cc_pops_2[cc] = pop
    else:
        cc_pops_3[cc] = pop

# Ver cuántos países hay en cada nivel.
❸ print(len(cc_pops_1), len(cc_pops_2), len(cc_pops_3))

wm = pygal.MapaMundo()
wm.title = "Población mundial en 2010, por países
❹ w m . a d d ('0-10m',
cc_pops_1) wm.add('10m-1bn',
cc_pops_2) wm.add('>1bn',
cc_pops_3)

wm.render_to_file('población_mundo.svg')
```

Para agrupar los países, creamos un diccionario vacío para cada categoría ❶. A continuación, hacemos un bucle en `cc_populations` para comprobar la población de cada país ❷. El bloque `if-elif-else` añade una entrada al diccionario apropiado (`cc_pops_1`, `cc_pops_2` o `cc_pops_3`) para cada par código de país-población.

En ❸ imprimimos la longitud de cada uno de estos diccionarios para averiguar los tamaños de los grupos. Cuando graficamos los datos ❹, nos aseguramos de agregar los tres grupos al `Worldmap`. Cuando ejecutes este programa, verás primero el tamaño de cada grupo:

```
85 69 2
```

Este resultado indica que hay 85 países con menos de 10 millones de habitantes, 69 países con entre 10 millones y 1.000 millones de habitantes y dos países atípicos con más de 1.000 millones. Parece una división lo suficientemente equilibrada para un mapa informativo. La Figura 16-10 muestra el mapa resultante.



Figura 16-10: La población mundial en tres grupos diferentes

Ahora, tres colores diferentes nos ayudan a distinguir los niveles de población. Dentro de cada uno de los tres niveles, los países se sombrean de claro a oscuro, de menor a mayor población.

Mapas del mundo con estilo en Pygal

Aunque los grupos de población de nuestro mapa son efectivos, la configuración de colores por defecto es bastante fea: por ejemplo, aquí Pygal ha elegido un chillón motivo rosa y verde. Utilizaremos las directivas de estilo de Pygal para rectificar los colores.

Indiquemos a Pygal que vuelva a utilizar un color base, pero esta vez elegiremos el color y aplicaremos un sombreado más diferenciado para los tres grupos de población:

mundo_población.py

```
importar json

importar pygal
❶ from pygal.style import RotateStyle
--snip--
# Agrupar los países en 3 niveles de población.
cc_pops_1, cc_pops_2, cc_pops_3 = {}, {}, {}
```

```
para cc, pop en
    cc_poblaciones.elementos(): si pop
    < 10000000:
        --snip--

❷ wm_style = RotateStyle('#336699')
❸ wm = pygal.Worldmap(style=wm_style)
wm.title = "Población mundial en 2010, por países
--snip--
```

Los estilos Pygal se almacenan en el módulo `style` del que importamos el estilo `RotateStyle` ❶. Esta clase toma un argumento, un color RGB en formato hexadecimal ❷. Pygal elige entonces los colores para cada uno de los grupos basándose en el color proporcionado. El *formato hexadecimal* es una cadena con una almohadilla (`#`) seguida de seis caracteres: los dos primeros representan el componente rojo del color, los dos siguientes el componente verde y los dos últimos el componente azul. Los valores de los componentes pueden ir de `00` (nada de ese color) a `FF` (cantidad máxima de ese color). Si buscas en Internet *un selector hexadecimal de colores*, encontrarás una herramienta que te permitirá experimentar con los colores y te dará los valores RGB. El color utilizado aquí (#336699) mezcla un poco de rojo (33), un poco más de verde (66), y aún más azul (99) para dar a `RotateStyle` un color base azul claro a partir del cual trabajar.

`RotateStyle` devuelve un objeto de estilo, que almacenamos en `wm_style`. Para utilizar este objeto de estilo, páselo como argumento de palabra clave cuando haga una instancia de `Worldmap` ❸. La Figura 16-11 muestra el gráfico actualizado.



Figura 16-11: Los tres grupos de población en un tema de color unificado

Este estilo da al mapa un aspecto unificado y permite distinguir fácilmente los grupos.

Aligerar el tema del color

Pygal tiende a usar temas oscuros por defecto. Para fines de impresión, he aligerado el estilo de mis gráficos usando `LightColorizedStyle`. Esta clase cambia el tema general del gráfico, incluyendo el fondo y las etiquetas, así como los colores individuales de cada país. Para utilizarla, primero hay que importar el estilo:

```
from pygal.style import EstiloColorLuz
```

A continuación, puede utilizar `LightColorizedStyle` por sí solo, como tal:

```
wm_style = EstiloColor Claro
```

Pero esta clase no te da control directo sobre el color usado, así que Pygal elegirá un color base por defecto. Para establecer un color, utilice `LightColorizedStyle` como base para `RotateStyle`. Importa tanto `LightColorizedStyle` como `RotateStyle`:

```
from pygal.style import LightColorizedStyle, RotateStyle
```

A continuación, cree un estilo utilizando `RotateStyle`, pero pásele un `base_style` adicional
argumento:

```
wm_style = RotateStyle('#336699', base_style=LightColorizedStyle)
```

Esto le da un tema general claro pero basa los colores de los países en el color que usted pasa como argumento. Con este estilo verá que sus gráficos coinciden un poco más con las capturas de pantalla.

Mientras experimenta para encontrar directivas de estilo que funcionen bien para diferentes visualizaciones, puede ser útil utilizar alias en sus declaraciones de `importación`:

```
from pygal.style import LightColorizedStyle as LCS, RotateStyle as RS
```

De este modo, las definiciones de estilo serán más breves:

```
wm_style = RS('#336699', base_style=LCS)
```

El uso de este pequeño conjunto de directivas de estilo te da un control significativo sobre la apariencia de los gráficos y mapas en Pygal.

TRY IT yOURSELF

16-5. Todos los países: En los mapas de población que hicimos en esta sección, nuestro programa no pudo encontrar automáticamente códigos de dos letras para unos 12 países. Averigua qué países carecen de códigos y busca los códigos en el diccionario `COUNTRIES`. Añade un bloque `if-elif` a `get_country_code()` para que devuelva los valores de código de país correctos para estos países específicos:

```
if nombre_país == 'Yemen, Rep.'
    return 'ye'
elif --snip--
```

Coloque este código después del bucle `COUNTRIES` pero antes del estado `return None`. Cuando haya terminado, debería ver un mapa más completo.

16-6. Producto Interior Bruto: La Open Knowledge Foundation mantiene un conjunto de datos que contiene el producto interior bruto (PIB) de cada país del mundo, que puedes encontrar en http://data.okfn.org/data/core/gdp/. Descargue la versión JSON de este conjunto de datos y trace el PIB de cada país del mundo para el año más reciente del conjunto de datos.

16-7. Elija sus propios datos: El Banco Mundial mantiene muchos conjuntos de datos que se desglosan para obtener información sobre cada país del mundo. Vaya a *http:// data.worldbank.org/indicator/* y encuentre un conjunto de datos que le parezca interesante. Haz clic en el conjunto de datos, pulsa el enlace **Descargar datos** y selecciona **CSV**. Recibirás tres archivos CSV, dos de los cuales están etiquetados como *Metadatos*; utiliza el tercer archivo CSV. Escribe un programa que genere un diccionario con los códigos de país de dos letras de Pygal como claves y los datos elegidos del archivo como valores. Traza los datos en un `mapa del mundo` y dale el estilo que quieras.

16-8. Probando el Módulo country_codes: Cuando escribimos el módulo `country_codes`, usamos sentencias `print` para comprobar si la función `get_country_code()` funcionaba. Escriba una prueba apropiada para esta función usando lo que aprendió en el Capítulo 11.

Resumen

En este capítulo ha aprendido a trabajar con conjuntos de datos en línea. Has aprendido a procesar archivos CSV y JSON, y a extraer los datos en los que quieres centrarte. Usando datos meteorológicos históricos, aprendiste más sobre cómo trabajar con matplotlib, incluyendo cómo usar el módulo `datetime` y cómo trazar múltiples series de datos en un gráfico. Aprendiste a trazar datos de países en un mapa del mundo en Pygal y a dar estilo a los mapas y gráficos de Pygal.

A medida que adquiera experiencia con los archivos CSV y JSON, podrá procesar casi cualquier dato que desee analizar. La mayoría de los conjuntos de datos en línea pueden descargarse en uno de estos formatos o en ambos. Al trabajar con estos formatos, podrás aprender también otros formatos de datos.

En el próximo c a p í t u l o , escribirás programas que recopilan automáticamente sus propios datos de fuentes en línea, y luego crearás visualizaciones de esos datos. Estas son habilidades divertidas de tener si quieres programar como hobby y habilidades críticas si estás interesado en programar profesionalmente.

17

TRABAJAR CON UN P I s



En este capítulo aprenderás a escribir un programa autónomo para generar un visualización basada en los datos que recupera. Su programa utilizará una *interfaz de programación de aplicaciones* web *(API)* para solicitar automáticamente información específica de un sitio web en lugar de páginas enteras. A continuación, utilizará esa información para generar una visualización. Dado que los programas escritos de este modo siempre utilizarán datos actuales para generar una v i s u a l i z a c i ó n , incluso cuando esos datos puedan estar cambiando rápidamente, siempre estarán actualizados.

Uso de una API web

Una API web es una parte de un sitio web diseñada para interactuar con programas que utilizan URL muy específicas para solicitar determinada información. Este tipo de solicitud se denomina *llamada API*. Los datos solicitados se devuelven en un formato fácil de procesar, como JSON o CSV. La mayoría de las aplicaciones que dependen de datos externos como las aplicaciones que se integran con las redes sociales, se basan en llamadas a la API.

Git y GitHub

Basaremos nuestra visualización en información de GitHub, un sitio que permite a los programadores colaborar en proyectos. Utilizaremos la API de GitHub para solicitar información sobre proyectos de Python en el sitio y luego generaremos una visualización interactiva de la popularidad relativa de estos proyectos en Pygal.

GitHub (https://github.com/) toma su nombre de Git, un sistema de control de versiones distribuido que permite a equipos de programadores colaborar en proyectos. Git ayuda a las personas a gestionar su trabajo individual en un proyecto, de modo que los cambios realizados por una persona no interfieran con los cambios que otras personas están haciendo. Cuando implementas una nueva funcionalidad en un proyecto, Git rastrea los cambios que haces en cada archivo. Cuando tu nuevo código funciona, *confirmas* los cambios que has hecho, y Git registra el nuevo estado de tu proyecto. Si cometes un error y quieres revertir tus cambios, puedes volver fácilmente a cualquier estado anterior. (Para saber más sobre el control de versiones con Git, consulta el Apéndice D.) Los proyectos en GitHub se almacenan en *repositorios*, que contienen todo lo relacionado con el proyecto: su código, información sobre sus colaboradores, cualquier incidencia o informe de errores, etcétera.

Cuando a los usuarios de GitHub les gusta un proyecto, pueden ponerle una "estrella" para mostrar su apoyo y hacer un seguimiento de los proyectos que podrían querer utilizar. En este capítulo escribiremos un programa para descargar automáticamente información sobre los proyectos Python con más estrellas en GitHub, y luego crearemos una visualización informativa de estos proyectos.

Solicitud de datos mediante una llamada a la API

La API de GitHub te permite solicitar una amplia gama de información a través de llamadas a la API. Para ver cómo es una llamada a la API, introduce lo siguiente en la barra de direcciones de tu navegador y pulsa ENTER:

```
https://api.github.com/search/repositories?q=language:python&sort=stars
```

Esta llamada devuelve el número de proyectos Python alojados actualmente en GitHub, así como información sobre los repositorios Python más populares. Examinemos la llamada. La primera parte,

https://api.github.com/, dirige la solicitud a la parte del sitio web de GitHub que responde a las llamadas a la API. La siguiente parte, `search/repositories`, indica a la API que realice una búsqueda en todos los repositorios de GitHub.

El signo de interrogación después de `repositorios` indica que vamos a pasar un argumento. La `q` significa consulta, y el signo igual nos permite comenzar

especificando una consulta (q=). Al utilizar language:python, indicamos que sólo queremos información sobre repositorios que tengan Python como lenguaje principal. La parte final, &sort=stars, ordena los proyectos por el número de estrellas que han recibido.

El siguiente fragmento muestra las primeras líneas de la respuesta. Puede ver en la respuesta que esta URL no está pensada para ser introducida por humanos.

```
{
  "total_count": 713062,
  "incomplete_results": false,
  "items": [
    {
      "id": 3544424,
      "name": "httpie",
      "full_name": "jkbrzt/httpie",
      --snip--
```

Como puedes ver en la segunda línea de salida, GitHub encontró un total de 713.062 proyectos Python en el momento de escribir esto. Dado que el valor de "incomplete_results" es falso, sabemos que la solicitud se ha realizado correctamente (no está incompleta). Si GitHub no hubiera podido procesar completamente l a solicitud de la API, habría devuelto aquí true. Los "elementos" devueltos son
que se muestran en la lista siguiente, que contiene detalles sobre los proyectos de Python más populares en GitHub.

Instalación de solicitudes

El paquete requests permite a un programa Python solicitar fácilmente información de un sitio web y examinar la respuesta que se devuelve. Para instalar requests, ejecute un comando como el siguiente:

```
$ pip install --user solicitudes
```

Si aún no ha utilizado pip, consulte "Instalar paquetes Python con pip" en la página 237. (Puede que necesite utilizar una versión ligeramente diferente de este com- mand dependiendo de la configuración de su sistema. (Puede que necesite utilizar una versión ligeramente diferente de este commandante, dependiendo de la configuración de su sistema).

Procesamiento de una respuesta API

Ahora empezaremos a escribir un programa para realizar una llamada a la API y procesar los resultados identificando los proyectos Python con más estrellas en GitHub:

python_repos.py

```
❶ importar peticiones

# Realiza una llamada a la API y almacena la respuesta.
❷ url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
❸ r = requests.get(url)
```

```
❹ print("Código de estado:", r.código_estado)
```

```
# Almacena la respuesta de la API en una variable.
❺ response_dict = r.json()

# Procesa los resultados.
print(response_dict.keys())
```

En ❶ importamos el módulo `requests`. En ❷ almacenamos la URL de la llamada a la API, y luego usamos `requests` para hacer la llamada ❸. Llamamos a `get()` y le pasamos la URL, y almacenamos el objeto respuesta en la variable `r`. El objeto respuesta tiene un atributo llamado `status_code`, que nos dice si la petición ha tenido éxito. (Un código de estado 200 indica una respuesta correcta.) En ❹ imprimimos el valor de `status_code` para asegurarnos de que la llamada se ha realizado correctamente.

La API devuelve la información en formato JSON, así que usamos el método `json()` ❺ para convertir la información a un diccionario Python. Almacenamos el diccionario resultante en `response_dict`.

Finalmente, imprimimos las claves de `response_dict` y vemos esto:

```
Código de estado: 200
dict_keys(['items', 'total_count', 'incomplete_results'])
```

Como el código de estado es 200, sabemos que la solicitud se ha realizado correctamente. El diccionario de respuesta sólo contiene tres claves: `'items'`, `'total_count'` e `'incomplete_results'`.

NOTE *Las llamadas sencillas como ésta deberían devolver un conjunto completo de resultados, por lo que es bastante seguro ignorar el valor asociado a 'incomplete_results'. Pero cuando realice llamadas a la API más complejas, su programa debería comprobar este valor.*

Trabajar con el diccionario de respuestas

Ahora que tenemos la información de la llamada a la API almacenada como un diccionario, podemos trabajar con los datos almacenados allí. Vamos a generar una salida que resuma la información. Esta es una buena forma de asegurarnos de que hemos recibido la información que esperábamos y de empezar a examinar la información que nos interesa:

python_repos.py

```
importar peticiones

# Realiza una llamada a la API y almacena la respuesta.
url = 'https://api.github.com/search/repositories?q=language:python&sort=stars'
r = requests.get(url)
print("Código de estado:", r.código_estado)

# Almacena la respuesta de la API en
una variable. response_dict =
r.json()
❶ print("Total repositorios:", response_dict['total_count'])
```

```
    # Explorar información sobre los repositorios.
❷   repo_dicts = response_dict['items']
    print("Repositorios devueltos:", len(repo_dicts))
```

```
# Examinar el primer repositorio.
❸ repo_dict = repo_dicts[0]
❹ print("\nClaves:", len(repo_dict))
❺ for key in sorted(repo_dict.keys()):
    print(key)
```

En ❶ imprimimos el valor asociado a `'total_count'`, que representa el número total de repositorios de Python en GitHub.

El valor asociado a `'items'` es una lista que contiene un número de diccionarios, cada uno de los cuales contiene datos sobre un repositorio Python individual. En ❷ almacenamos esta lista de diccionarios en `repo_dicts`. Luego imprimimos la longitud de `repo_dicts` para ver de cuántos repositorios tenemos información.

Para echar un vistazo más de cerca a la información devuelta sobre cada repositorio, extraemos el primer elemento de `repo_dicts` y lo almacenamos en `repo_dict` ❸. A continuación, imprimimos el número de claves del diccionario para ver cuánta información tenemos ❹. En ❺ imprimimos todas las claves del diccionario para ver qué tipo de información incluye.

Los resultados empiezan a darnos una idea más clara de los datos reales:

```
Código de estado: 200
Total de repositorios: 713062
Repositorios devueltos: 30

❶ Claves: 68
archive_url
assignees_url
blobs_url
--snip--
vigilant
es de
url
recuento_de_vigilantes
```

La API de GitHub devuelve mucha información sobre cada repositorio: hay 68 claves en `repo_dict` ❶. Cuando busques entre estas claves, obtendrás una idea del tipo de información que se puede extraer sobre un proyecto. (La única forma de saber qué información está disponible a través de una API es leer la documentación o examinar la información a través del código, como estamos haciendo aquí).

Saquemos los valores de algunas de las claves de `repo_dict`:

python_repos.py

```
--snip--
# Explorar información sobre los repositorios.
repo_dicts = response_dict['items']
print("Repositorios devueltos:",
len(repo_dicts))

# Examinar el primer repositorio.
```

```
repo_dict = repo_dicts[0]

print("\nInformación seleccionada sobre el primer repositorio:")
print('Nombre:', repo_dict['nombre'])
print('Propietario:', repo_dict['propietario']['login'])
```

```
❸ print('Stars:', repo_dict['stargazers_count'])
  print('Repository:', repo_dict['html_url'])
❹ print('Creado:', repo_dict['created_at'])
❺ print('Actualizado:', repo_dict['updated_at'])
  print('Descripción:', repo_dict['description'])
```

Aquí imprimimos los valores de una serie de claves del diccionario del primer repositorio. En ❶ imprimimos el nombre del proyecto. Un diccionario entero representa al dueño del proyecto, así que en ❷ usamos la clave `owner` para acceder al diccionario que representa al dueño y luego usamos la clave `login` para obtener el nombre de usuario del dueño. En ❸ imprimimos cuántas estrellas ha ganado el proyecto y la URL del repositorio GitHub del proyecto. A continuación, mostramos cuándo se creó ❹ y cuándo se actualizó por última vez ❺. Por último, imprimimos la descripción del repositorio, y el resultado debería ser algo parecido a esto:

```
Código de estado: 200
Total de repositorios: 713065
Repositorios devueltos: 30

Información seleccionada sobre el primer depósito:
Nombre: httpie
Propietario:
jkbrzt
Estrellas: 16101
Repositorio: https://github.com/jkbrzt/httpie
Creado: 2012-02-25T12:39:13Z
Actualizado: 2015-07-13T14:56:41Z
Descripción: Cliente HTTP CLI; reemplazo de cURL fácil de usar que cuenta
con una interfaz de usuario intuitiva, soporte JSON, resaltado de
sintaxis, descargas similares a wget, extensiones, etc.
```

Podemos ver que el proyecto Python con más estrellas en GitHub en el momento de escribir esto es HTTPie, su propietario es el usuario *jkbrzt*, y ha sido destacado por más de 16.000 usuarios de GitHub. Podemos ver la URL del repositorio del proyecto, su fecha de creación en febrero de 2012, y que ha sido actualizado recientemente. Por último, la descripción nos dice que HTTPie ayuda a hacer llamadas HTTP desde un terminal (*CLI* es la abreviatura de *interfaz de línea de comandos*).

Resumen de los principales repositorios

Cuando hagamos una visualización de estos datos, querremos incluir más de un repositorio. Escribamos un bucle para imprimir información seleccionada sobre cada uno de los repositorios devueltos por la llamada a la API, de forma que podamos incluirlos todos en la visualización:

python_repos.py

```
--snip--
# Explorar información sobre los repositorios.
repo_dicts = response_dict['items']
print("Repositorios devueltos:",
len(repo_dicts))
```

```
❶ print("❶Información seleccionada sobre cada repositorio:")
❷ for repo_dict in repo_dicts:
      print('Nombre:', repo_dict['nombre'])
      print('Owner:', repo_dict['owner']['login'])
      print('Stars:', repo_dict['stargazers_count'])
      print('Repository:', repo_dict['html_url'])
      print('Description:', repo_dict['description'])
```

Imprimimos un mensaje introductorio en ❶. En ❷ recorremos todos los diccionarios de `repo_dicts`. Dentro del bucle imprimimos el nombre de cada proyecto, su propietario, cuántas estrellas tiene, su URL en GitHub y la descripción del proyecto:

```
Código de estado: 200
Total de repositorios: 713067
Repositorios devueltos: 30

Información seleccionada sobre cada repositorio:

Nombre: httpie
Propietario:
jkbrzt
Estrellas: 16101
Repositorio: https://github.com/jkbrzt/httpie
Descripción: Cliente HTTP CLI; reemplazo de cURL fácil de usar que ofrece
una interfaz de usuario intuitiva, soporte JSON, resaltado de sintaxis,
descargas similares a wget, extensiones, etc.

Nombre: django
Propietario:
django
Estrellas: 15028
Repositorio: https://github.com/django/django
Descripción: El marco Web para perfeccionistas con plazos de entrega.
--snip--

Nombre: powerline
Propietario:
powerline
Estrellas: 4315
Repositorio: https://github.com/powerline/powerline
Descripción: Powerline es un plugin de statusline para vim, y proporciona
statuslines y avisos para varias otras aplicaciones, incluyendo zsh, bash,
tmux, IPython, Awesome y Qtile.
```

En estos resultados aparecen algunos proyectos interesantes, y puede que merezca la pena echar un vistazo a unos cuantos. Pero no te entretengas demasiado, porque estamos a punto de crear una visualización que facilitará mucho la lectura de los resultados.

Supervisión de los límites de la tasa API

La mayoría de las API tienen una tasa limitada, lo que significa que hay

un límite en el número de solicitudes que puede realizar en un determinado periodo de tiempo. Para saber si

acercándote a los límites de GitHub, introduce *https://api.github.com/rate_limit* en un navegador web. Deberías ver una respuesta como esta:

```
{
  "recursos": {
    "core": {
      "límite": 60,
      "restante": 58,
      "reset": 1426082320
    },
❶   "buscar": {
❷     "límite": 10,
❸     "restante": 8,
❹     "reset": 1426078803
    }
  },
  "tasa": {
    "límite": 60,
    "restante": 58,
    "reset": 1426082320
  }
}
```

La información que nos interesa es el límite de tarifa para la búsqueda API ❶. Vemos en ❷ que el límite es de 10 peticiones por minuto y que nos quedan 8 peticiones para el minuto actual ❸. El valor de reinicio representa la hora en *Unix* o *tiempo de época* (el número de segundos desde la medianoche del 1 de enero de 1970) en que se reiniciará nuestra cuota ❹. Si alcanza su cuota, recibirá una breve respuesta que le informará de que ha alcanzado el límite de la API. Si alcanza el límite, espere a que se restablezca la cuota.

NOTE *Muchas API requieren que se registre y obtenga una clave API para poder realizar llamadas a la API.*
En el momento de escribir este artículo, GitHub no tiene este requisito, pero si obtienes una clave API, tus límites serán mucho mayores.

Visualización de repositorios con Pygal

Ahora que tenemos algunos datos interesantes, vamos a hacer una visualización que muestre la popularidad relativa de los proyectos Python en GitHub. Haremos un gráfico de barras interactivas: la altura de cada barra representará el número de estrellas que ha adquirido el proyecto. Si haces clic en una barra, irás a la página de inicio de ese proyecto en GitHub. He aquí un primer intento:

python_repos.py

```
importar peticiones
importar pygal
from pygal.style import LightColorizedStyle as LCS, LightenStyle as LS
```

```
# Realiza una llamada a la API y almacena la respuesta.
URL = 'https://api.github.com/search/repositories?q=language:python&sort=star'
r = requests.get(URL)
print("Código de estado:", r.código_estado)

# Almacena la respuesta de la API en una
variable. response_dict = r.json()
print("Total repositorios:", response_dict['total_count'])

# Explorar información sobre los repositorios.
repo_dicts = response_dict['items']

❶ nombres, estrellas = [], []
para repo_dict en repo_dicts:
❷     names.append(repo_dict['name'])
      stars.append(repo_dict['stargazers_count'])

# Hacer visualización.
❸ mi_estilo = LS('#333366', base_style=LCS)
❹ chart = pygal.Bar(style=my_style, x_label_rotation=45, show_legend=False)
chart.title = 'Proyectos Python con más estrellas en GitHub'
chart.x_labels = nombres

❺ chart.add('', stars)
chart.render_to_file('python_repos.svg')
```

Empezamos importando `pygal` y los estilos de Pygal que necesitaremos para el gráfico. Seguimos imprimiendo el estado de la respuesta de la llamada a la API y el número total de repositorios encontrados, para saber si hubo algún problema con la llamada a la API. Ya no imprimimos información sobre los proyectos específicos que se devuelven, porque esa información se incluirá en la visualización.

En ❶ creamos dos listas vacías para almacenar los datos que incluiremos en el gráfico. Necesitaremos el nombre de cada proyecto para etiquetar las barras, y necesitaremos el número de estrellas para determinar la altura de las barras. En el bucle, añadimos a estas listas el nombre de cada proyecto y el número de estrellas que tiene ❷.

A continuación definimos un estilo utilizando la clase `LightenStyle` (alias `LS`) y lo basamos en un tono oscuro de azul ❸. También pasamos el argumento `base_style` para utilizar la clase `LightColorizedStyle` (alias `LCS`). A continuación, utilizamos `Bar()` para hacer un gráfico de barras simple y le pasamos `mi_estilo` ❹. Pasamos otros dos argumentos de estilo: establecemos la rotación de las etiquetas a lo largo del eje x a 45 grados (`x_label_rotation=45`), y ocultamos la leyenda, porque estamos trazando sólo una serie en el gráfico (`show_legend=False`). A continuación, damos un título al gráfico y establecemos el atributo `x_labels` en los `nombres` de las listas.

Como no necesitamos que esta serie de datos esté etiquetada, pasamos una cadena vacía para la etiqueta cuando añadimos los datos en ❺. El gráfico resultante se muestra en la Figura 17-1. Podemos ver que los

primeros proyectos son significativamente más populares que el resto, pero todos ellos son proyectos importantes en el ecosistema Python.

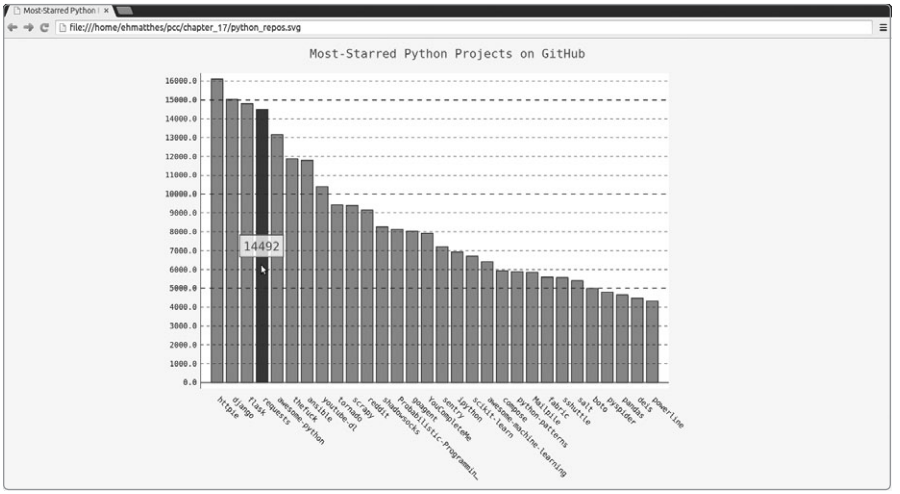


Figura 17-1: Proyectos Python con más estrellas en GitHub

Perfeccionamiento de los gráficos Pygal

Vamos a refinar el estilo de nuestro gráfico. Vamos a hacer algunas personalizaciones diferentes, por lo que primero reestructurar el código ligeramente mediante la creación de un objeto de configuración que contiene todas nuestras personalizaciones para pasar a `Bar()`:

python_repos.py

```
--snip--

# Hacer visualización.
mi_estilo = LS('#333366', base_style=LCS)

❶ mi_config = pygal.Config()
❷ my_config.x_label_rotation = 45
  my_config.show_legend = False
❸ my_config.title_font_size = 24
  my_config.label_font_size = 14
  my_config.major_label_font_size = 18
❹ my_config.truncate_label = 15
❺ my_config.show_y_guides = False
❻ mi_config.anchura = 1000

❼  c hart = pygal.Bar(my_config, style=my_style)
  chart.title = 'Proyectos Python con más estrellas en
  GitHub' chart.x_labels = names

  chart.add('', stars)
  chart.render_to_file('python_repos.svg')
```

En ❶ hacemos una instancia de la clase `Config` de Pygal, llamada `mi_config`; modificando los atributos de mi_config personalizaremos la apariencia del gráfico.

Establecemos los dos atributos `x_label_rotation` y `show_legend` ❷, originalmente pasados como argumentos de palabra clave cuando hicimos una instancia de `Bar`.

En ❸

establecemos el tamaño de la fuente para el título del gráfico, las etiquetas secundarias y las etiquetas principales. Las etiquetas menores en este gráfico son los nombres de los proyectos en el eje x y la mayoría de los números en el eje y. Las etiquetas mayores son sólo las etiquetas en el eje y que marcan los incrementos de 5000 estrellas. Las etiquetas principales son sólo las etiquetas en el eje y que marcan incrementos de 5000 estrellas. Estas etiquetas serán más grandes, por eso las diferenciamos. En ❹ utilizamos `truncate_label` para acortar los nombres de proyecto más largos a 15 caracteres. (Cuando pases el ratón por encima de un nombre de proyecto truncado en tu pantalla, aparecerá el nombre completo). A continuación, ocultamos las líneas horizontales del gráfico estableciendo `show_y_guides` en `False` ❺. Por último, en ❻ establecemos un ancho personalizado para que el gráfico utilice más espacio disponible en el navegador.

Ahora, cuando hacemos una instancia de `Bar` en ❼, pasamos `my_config` como primer argumento, y envía todos nuestros ajustes de configuración en un solo argumento.
Podemos hacer tantos cambios de estilo y configuración como queramos a través de `my_config`, y la línea de ❼ no cambiará. La Figura 17-2 muestra el gráfico rediseñado.

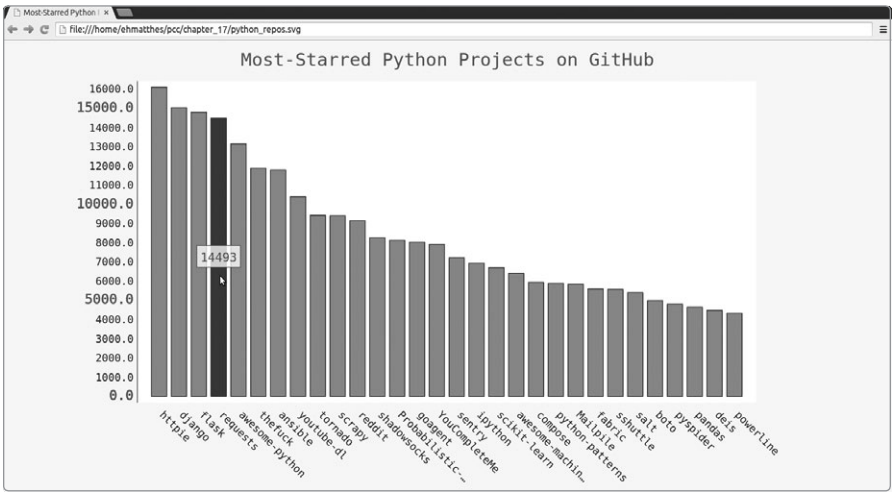


Figura 17-2: Se ha perfeccionado el estilo del gráfico.

Añadir información sobre herramientas personalizada

En Pygal, al pasar el cursor sobre una barra individual se muestra la información que representa la barra. Esto se llama comúnmente *tooltip*, y en este caso
actualmente muestra el número de estrellas que tiene un proyecto. Vamos a crear un tooltip personalizado para mostrar también la descripción de cada proyecto.

Veamos un breve ejemplo utilizando los tres primeros proyectos trazados individualmente con etiquetas personalizadas para cada barra.

Para ello, pasaremos una lista de diccionarios a `add()` en lugar de una lista de valores:

bar_descripciones.py

```
importar pygal
from pygal.style import LightColorizedStyle as LCS, LightenStyle as LS

mi_estilo = LS('#333366', base_style=LCS)
chart = pygal.Bar(style=my_style, x_label_rotation=45, show_legend=False)
```

```
chart.title = 'Proyectos Python'
chart.x_labels = ['httpie', 'django', 'flask']

❶ plot_dicts = [
❷     {'value': 16101, 'label': 'Descripción de httpie.'},
      {'value': 15028, 'label': 'Descripción de django.'},
      {'value': 14798, 'label': 'Descripción del frasco.'},
      ]

❸ chart.add('', plot_dicts)
  chart.render_to_file('bar_descriptions.svg')
```

En ❶ definimos una lista llamada `plot_dicts` que contiene tres diccionarios: uno para el proyecto HTTPie, otro para el proyecto Django y otro para Flask. Cada diccionario tiene dos claves: `'value'` y `'label'`. Pygal utiliza el número asociado a `'value'` para calcular la altura de cada barra, y utiliza la cadena asociada a `'label'` para crear el tooltip de cada barra. Por ejemplo, el primer diccionario de ❷ creará una barra que representa un proyecto con 16.101 estrellas, y su tooltip dirá *Descripción de httpie*.

El método `add()` necesita una cadena y una lista. Cuando llamamos a `add()`, pasamos la lista de diccionarios que representan las barras (`plot_dicts`)❸.

La Figura 17-3 muestra uno de los tooltips. Pygal incluye el número de estrellas como tooltip por defecto además del tooltip personalizado que le pasamos.

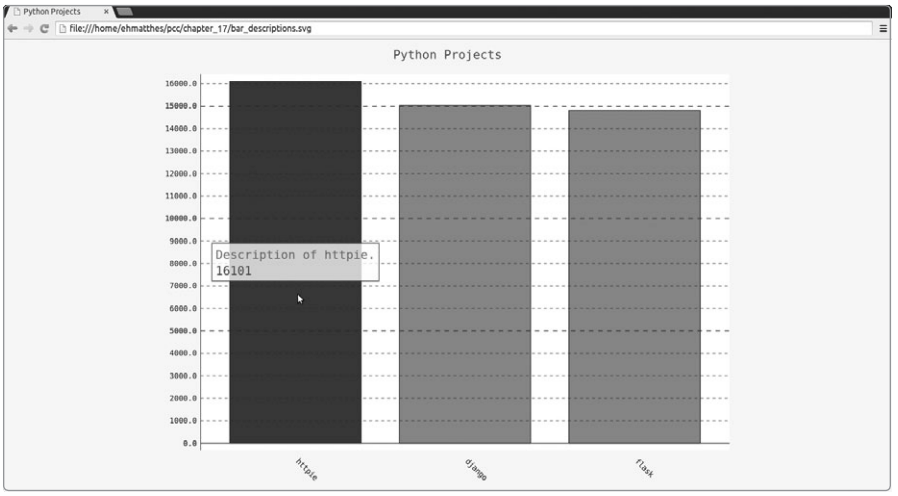


Figura 17-3: Cada barra tiene una etiqueta personalizada.

Trazado de los datos

Para trazar nuestros datos, generaremos `plot_dicts` automáticamente para los 30 proyectos devueltos por la llamada a la API.

Aquí está el código para hacer esto:

python_repos.py

```
--snip--
# Explorar información sobre los
repositorios. repo_dicts =
response_dict['items'] print("Número de
items:", len(repo_dicts))

❶ n ames, plot_dicts = [], []
for repo_dict in repo_dicts:
    names.append(repo_dict['nombre'])

❷     plot_dict = {
          'value': repo_dict['stargazers_count'],
          'label': repo_dict['description'],
          }
❸     plot_dicts.append(plot_dict)

# Hacer visualización.
mi_estilo = LS('#333366', base_style=LCS)
--snip--

❹ chart.add('', plot_dicts)
chart.render_to_file('python_repos.svg')
```

En ❶ hacemos una lista vacía para `nombres` y una lista vacía para `plot_dicts`. Seguimos necesitando la lista de `nombres` para generar las etiquetas del eje x.

Dentro del bucle creamos el diccionario `plot_dict` para cada proyecto ❷. Almacenamos el número de estrellas con la clave `'value'` y la descripción del proyecto con la clave `'label'` en cada `plot_dict`. A continuación, añadimos el `plot_dict` de cada proyecto a `plot_dicts` ❸. En ❹ pasamos la lista `plot_dicts` a `add()`. La Figura 17-4 muestra el gráfico resultante.

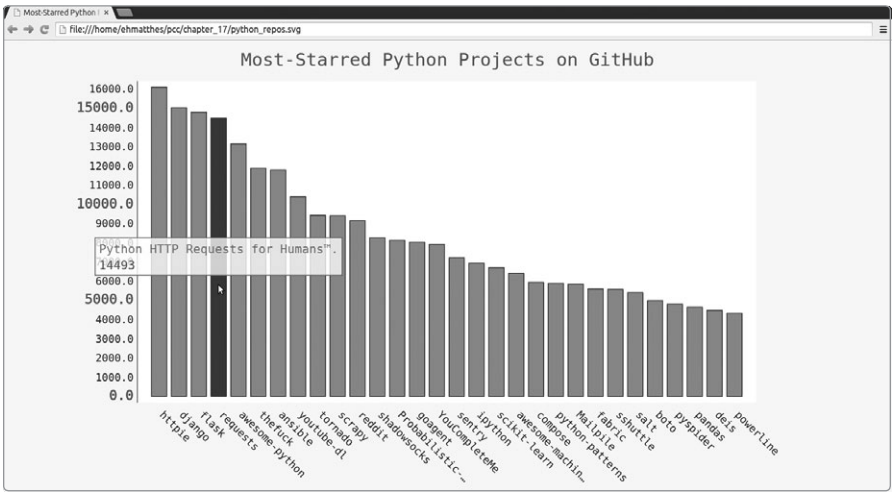


Figura 17-4: Al pasar el ratón por encima de una barra se muestra la descripción del proyecto.

Añadir enlaces clicables a nuestro gráfico

Pygal también permite utilizar cada barra del gráfico como enlace a una página web. Para añadir esta capacidad, sólo tenemos que añadir una línea a nuestro código, aprovechando el d i c c i o n a r i o que hemos configurado para cada proyecto. Añadimos un nuevo par clave-valor al `plot_dict` de cada proyecto utilizando la clave `'xlink'`:

python_repos.py

```
--snip--
names, plot_dicts = [], []
for repo_dict in repo_dicts:
    names.append(repo_dict['nombre'])

    plot_dict = {
        'value': repo_dict['stargazers_count'],
        'label': repo_dict['description'],
        'xlink': repo_dict['html_url'],
        }
    plot_dicts.append(plot_dict)
--snip--
```

Pygal utiliza la URL asociada a `'xlink'` para convertir cada barra en un enlace activo. Puedes hacer clic en cualquiera de las barras del gráfico y la página de GitHub de ese proyecto se abrirá automáticamente en una nueva pestaña de tu navegador. Ya tienes una visualización interactiva e informativa de los datos obtenidos a través de una API.

La API de Hacker News

Para saber cómo utilizar las llamadas a la API en otros sitios, nos fijaremos en Hacker News *(*http://news.ycombinator.com/). En Hacker News la gente comparte artículos sobre programación y tecnología, y participa en animados debates sobre esos artículos. La API de Hacker News proporciona acceso a datos sobre todos los envíos y comentarios del sitio, a los que se puede acceder sin tener que r e g i s t r a r s e para obtener una clave.

La siguiente llamada devuelve información sobre el artículo principal actual en el momento de escribir este artículo:

```
https://hacker-news.firebaseio.com/v0/item/9884165.json
```

La respuesta es un diccionario de información sobre el artículo con el ID 9884165:

```
{
❶    'url': 'http://www.bbc.co.uk/news/science-environment-33524589',
     'type': 'story',
❷    'título': 'New Horizons: La nave espacial de la Nasa pasa a toda velocidad
junto a Plutón',
❸    'descendientes': 141,
     puntuación: 230,
     'tiempo': 1436875181,
```

```
        'texto': '',
```

```
    'by': 'nns',
    'id': 9884165,
❹   'kids': [9884723, 9885099, 9884789, 9885604, 9885844]
}
```

El diccionario contiene una serie de claves con las que podemos trabajar, como `'url'` ❶ y `'title'` ❷. La clave `'descendientes'` contiene el número de comentarios que ha recibido un artículo ❸. La clave `'hijos'` proporciona los ID de todos los comentarios realizados directamente en respuesta a este envío ❹. Cada uno de estos comentarios puede tener sus propios hijos, por lo que el número de descendientes de un envío puede ser mayor que su número de hijos.

Hagamos una llamada a la API que devuelva los ID de los principales artículos actuales de Hacker News y, a continuación, examinemos cada uno de los artículos principales:

hn_presentaciones.py

```
importar solicitudes

from operador import itemgetter

# Realiza una llamada a la API y almacena la respuesta.
❶ url = 'https://hacker-news.firebaseio.com/v0/topstories.json'
r = requests.get(url)
print("Código de estado:", r.código_estado)

# Procesar información sobre cada envío.
❷ submission_ids = r.json()
❸ sumision_dicts = []
para submission_id en submission_ids[:30]:
    # Realiza una llamada a la API distinta para cada envío.
❹   url = ('https://hacker-news.firebaseio.com/v0/item/' +
            str(submission_id) + '.json')
    submission_r = requests.get(url)
    print(submission_r.status_code)
    response_dict = submission_r.json()

❺   submission_dict = {
        'title': response_dict['title'],
        'link': 'http://news.ycombinator.com/item?id=' + str(submission_id),
❻       'comentarios': response_dict.get('descendientes', 0)
        }
    submission_dicts.append(submission_dict)

❼ submission_dicts = sorted(submission_dicts, key=itemgetter('comments'),
                            reverse=True)

❽ para diccionario_de_envío en diccionario_de_envío:
    print("Título:", diccionario_de_envío['título'])
    print("Enlace de discusión:",
    diccionario_de_envío['enlace'])
    print("Comentarios:",
    diccionario_de_envío['comentarios'])
```

En primer lugar, realizamos la llamada a la API e imprimimos el estado de la respuesta ❶. Esta llamada a la API devuelve una lista con los ID de los 500 artículos más populares de Hacker News en el momento en que se realiza la llamada. A continuación, convertimos la respuesta

texto a una lista Python en ❷, que almacenamos en `submission_ids`. Utilizaremos estos ID para construir un conjunto de diccionarios que almacenen cada uno información sobre uno de los envíos actuales.

Creamos una lista vacía llamada `submission_dicts` en ❸ para almacenar estos diccionarios. A continuación, hacemos un bucle con los ID de los 30 envíos principales. Hacemos una nueva llamada a la API para cada envío generando una URL que incluye el valor actual de `submission_id` ❹. Imprimimos el estado de cada solicitud para saber si se ha realizado correctamente.

En ❺ creamos un diccionario para el envío que se está procesando en ese momento, donde almacenamos el título del envío y un enlace a la página de discusión de ese elemento. En ❻ almacenamos el número de comentarios en el diccionario. Si un artículo aún no tiene comentarios, la clave `'descendientes'` no estará presente. Cuando no estés seguro de si una clave existe en un diccionario, utiliza el método `dict.get()`, que devuelve el valor asociado a la clave dada si existe o el valor que proporciones si la clave no existe (0 en este ejemplo). Finalmente, añadimos cada `submission_dict` a la lista `submission_dicts`.

Los envíos a Hacker News se clasifican de acuerdo con una puntuación general, basada en una serie de factores que incluyen cuántas veces se ha votado, cuántos comentarios se han recibido y lo reciente que es el envío. Queremos ordenar la lista de diccionarios por el número de comentarios. Para ello, utilizamos una función llamada `itemgetter()` ❼, que proviene del módulo `operador`. Pasamos a esta función la clave `"comentarios"` y extrae el valor asociado a esa clave de cada diccionario de la lista. La función `sorted()` utiliza este valor como base para ordenar la lista. Ordenamos la lista en orden inverso para colocar primero las historias más comentadas.

Una vez ordenada la lista, la recorremos en ❽ e imprimimos tres datos sobre cada uno de los envíos más destacados: el título, un enlace a la página de debate y el número de comentarios que tiene el envío en ese momento:

```
Código de estado: 200
200
200
200
--snip--

Título: Firefox desactiva Flash por defecto
Enlace de debate: http://news.ycombinator.com/item?id=9883246
Comentarios: 231

Título: New Horizons: La nave espacial de la NASA pasa a toda
velocidad por Plutón Enlace de discusión:
http://news.ycombinator.com/item?id=9884165 Comentarios: 142

Título: Iran Nuclear Deal Is Reached With World Powers
Discussion link: http://news.ycombinator.com/item?id=9884005
```

```
Comentarios: 141
```

```
Título: Match Group compra PlentyOfFish por 575 millones de dólares
Enlace de debate: http://news.ycombinator.com/item?id=9884417
Comentarios: 75

Título: Nuestros dispositivos Nexus 4 están a punto de explotar
Enlace de debate: http://news.ycombinator.com/item?id=9885625
Comentarios: 14

--snip--
```

Utilizarías un proceso similar para acceder a la información y analizarla con cualquier API. Con estos datos, podrías hacer una visualización que mostrara qué sub-misiones han inspirado los debates recientes más activos.

TRY IT yOURSELF

17-1. Otros idiomas: Modifica la llamada a la API en *python_repos.py* para que genere un gráfico mostrando los proyectos más populares en otros lenguajes. Pruebe lenguajes como *JavaScript*, *Ruby*, *C*, *Java*, *Perl*, *Haskell* y *Go*.

17-2. Discusiones activas: Usando los datos de *hn_submissions.py*, haz un gráfico de barras que muestre las discusiones más activas que se están produciendo actualmente en Hacker News. La altura de cada barra debe corresponder al número de comentarios que tiene cada envío. La etiqueta de cada barra debe incluir el título del envío, y cada barra debe actuar como un enlace a la página de discusión para ese envío.

17-3. Probando `python_repos.py`: En *python_repos.py*, imprimimos el valor de `status_code` para asegurarnos de que la llamada a la API tuvo éxito. Escribe un programa llamado *test_python_repos.py*, que use `unittest` para afirmar que el valor de `status_code` es 200. Piensa en otras afirmaciones que puedas hacer, por ejemplo, que el número de elementos devueltos sea el esperado y que el número total de repositorios sea superior a una cantidad determinada.

resumen

En este capítulo has aprendido a utilizar las API para escribir programas autónomos que recopilan automáticamente los datos que necesitan y los utilizan para crear una visualización. Usamos la API de GitHub para explorar los proyectos Python con más estrellas en GitHub, y también vimos brevemente la API de Hacker News. Aprendiste cómo utilizar el paquete requests para realizar automáticamente una llamada a la API de GitHub y cómo procesar los resultados de esa llamada. También hemos introducido algunos ajustes de Pygal que personalizan aún más el aspecto de los gráficos que generas.

En el proyecto final utilizaremos Django para construir una aplicación web.

PROYECTO 3

W E B A PP L IC A C I O N E S

18

GETTING STARTED DJANGO



Entre bastidores, los sitios web actuales son en realidad aplicaciones ricas que actúan como aplicaciones de escritorio totalmente desarrolladas. Python tiene un gran conjunto de herramientas para construir aplicaciones web. En este capítulo aprenderás a usar Django (http://djangoproject.com/) para construir un proyecto llamado Learning Log, un sistema de diario en línea que te permite llevar un registro de la información que has aprendido sobre temas particulares.

Escribiremos una especificación para este proyecto y luego definiremos modelos para los datos con los que trabajará la aplicación. Utilizaremos el sistema de administración de Django para introducir algunos datos iniciales y luego aprenderemos a escribir vistas y plantillas para que Django pueda construir las páginas de nuestro sitio.

Django es un *framework web, un* conjunto de herramientas diseñadas para ayudarte a construir sitios web interactivos. Django puede

responder a peticiones de páginas y hacer que

más fácil de leer y escribir en una base de datos, gestionar usuarios, y mucho más. En los Capítulos 19 y 20 refinaremos el proyecto Learning Log y luego lo desplegaremos en un servidor en vivo para que tú (y tus amigos) puedan usarlo.

Crear un proyecto

Al iniciar un proyecto, primero hay que describirlo en una *especificación*. A continuación, configure un entorno virtual para construir el proyecto.

Escribir una especificación

Una especificación completa detalla los objetivos del proyecto, describe su funcionalidad y analiza su aspecto e interfaz de usuario. Como cualquier buen proyecto o plan de negocio, las especificaciones deben mantenerte centrado y ayudar a que el proyecto siga su curso. No vamos a escribir aquí una especificación completa del proyecto, pero vamos a establecer unos cuantos objetivos claros para mantener centrado nuestro proceso de desarrollo. Estas son las especificaciones que utilizaremos:

> Escribiremos una aplicación web llamada Learning Log que permita a los usuarios registrar los temas que les interesan y hacer anotaciones en el diario a medida que aprenden sobre cada tema. La página de inicio de Learning Log debe describir el sitio e invitar a los usuarios a registrarse o a iniciar sesión. Una vez conectado, el usuario podrá crear nuevos temas, añadir nuevas entradas y leer y editar las ya existentes.

Cuando aprendes sobre un tema nuevo, llevar un diario de lo que has aprendido puede ser útil para hacer un seguimiento y volver a consultar la información. Una buena aplicación hace que este proceso sea eficiente.

Creación de un entorno virtual

Para trabajar con Django, primero configuraremos un entorno virtual en el que trabajar. Un *entorno virtual* es un lugar en tu sistema donde puedes instalar paquetes y aislarlos de todos los demás paquetes de Python. Separar las librerías de un proyecto de otros proyectos es beneficioso y será necesario cuando despleguemos Learning Log en un servidor en el Capítulo 20.

Cree un nuevo directorio para su proyecto llamado *learning_log*, cambie a ese directorio en un terminal y cree un entorno virtual. Si estás usando Python 3, deberías ser capaz de crear un entorno virtual con el siguiente comando:

```
learning_log$ python -m venv ll_env
learning_log$
```

Aquí estamos ejecutando el módulo `venv` y usándolo para crear un entorno virtual llamado *ll_env*. Si esto funciona, pasa a "Activación del entorno virtual" en la página 399. Si no funciona, lee la siguiente sección, "Instalando

virtualenv".

Instalación de virtualenv

Si estás utilizando una versión anterior de Python o si tu sistema no está configurado para utilizar correctamente el módulo venv, puedes instalar el paquete virtualenv. Para instalar virtualenv, introduzca lo siguiente:

```
$ pip install --user virtualenv
```

Tenga en cuenta que puede necesitar usar una versión ligeramente diferente de este comando. (Si aún no ha utilizado pip, consulte "Instalación de paquetes Python con pip" en la página 237.)

NOTE *Si utilizas Linux y esto sigue sin funcionar, puedes instalar virtualenv a través del gestor de paquetes de tu sistema. En Ubuntu, por ejemplo, el comando* `sudo apt-get install` *python-virtualenv instalará virtualenv.*

Vaya al directorio *learning_log* en un terminal y cree un entorno virtual como el siguiente:

```
learning_log$ virtualenv ll_env
Nuevo ejecutable python en ll_env/bin/python
Instalando setuptools, pip...hecho.
learning_log$
```

NOTE *Si tiene más de una versión de Python instalada en su sistema, debe especificar la versión que debe utilizar virtualenv. Por ejemplo, el comando* `virtualenv ll_env --python=python3` *creará un entorno virtual que utiliza Python 3.*

Activar el entorno virtual

Ahora que tenemos un entorno virtual configurado, necesitamos activarlo con el siguiente comando:

```
learning_log$ source ll_env/bin/activate
❶ (ll_env)learning_log$
```

Este comando ejecuta el script *activate* en *ll_env/bin*. Cuando el entorno esté activo, verá el nombre del entorno entre paréntesis, como se muestra en ❶; entonces podrá instalar paquetes en el entorno y utilizar paquetes que ya hayan sido instalados. Los paquetes que instales en *ll_env* sólo estarán disponibles mientras el entorno esté activo.

NOTE *Si utiliza Windows, utilice el comando* `ll_env\Scripts\activate` *(sin la palabra* `source`*) para activar el entorno virtual.*

Para dejar de utilizar un entorno virtual, introduzca **deactivate**:

```
(ll_env)learning_log$ deactivate
learning_log$
```

El entorno también quedará inactivo si cierras el terminal en el que se está ejecutando.

Instalación de Django

Una vez que hayas creado tu entorno virtual y lo hayas activado, instala Django:

```
(ll_env)learning_log$ pip install Django
Instalación de los paquetes recopilados:
Django Instalado con éxito Django
Limpiando...
(ll_env)learning_log$
```

Como estamos trabajando en un entorno virtual, este comando es el mismo en todos los sistemas. No hay necesidad de utilizar la bandera `--user`, y no hay necesidad de utilizar comandos más largos como `python -m pip install` *nombre_paquete*.

Ten en cuenta que Django sólo estará disponible cuando el entorno esté activo.

Creación de un proyecto en Django

Sin salir del entorno virtual activo (recuerde buscar *ll_env* entre paréntesis), introduzca los siguientes comandos para crear un nuevo proyecto:

```
❶ (ll_env)learning_log$ django-admin.py startproject learning_log .
❷ (ll_env)learning_log$ ls
learning_log ll_env manage.py
❸ (ll_env)learning_log$ ls learning_log
 init.py settings.py urls.py wsgi.py
```

El comando ❶ le dice a Django que cree un nuevo proyecto llamado *learning_log*. El punto al final del comando crea el nuevo proyecto con una estructura de directorios que facilitará el despliegue de la aplicación en un servidor cuando hayamos terminado de desarrollarla.

NOTE *No olvides este punto, o puedes encontrarte con algunos problemas de configuración cuando despleguemos la aplicación. Si olvidas el punto, borra los archivos y carpetas que se crearon (excepto* ll_env*), y ejecuta el comando de nuevo.*

Ejecutando el comando `ls` (`dir` en Windows) ❷ muestra que Django ha creado un nuevo directorio llamado *learning_log*. También ha creado un archivo llamado *manage.py*, que es un pequeño programa que recibe comandos y los envía a la parte relevante de Django para ejecutarlos. Usaremos estos comandos para gestionar tareas como trabajar con bases de datos y ejecutar servidores.

El directorio *learning_log* contiene cuatro archivos ❸, los más importantes de los cuales son *settings.py*, *urls.py*, y *wsgi.py*. El archivo

settings.py controla cómo Django interactúa con tu sistema y gestiona tu proyecto. Modificaremos algunos de estos ajustes y añadiremos algunos ajustes propios a medida que el proyecto

evoluciona. El archivo *urls.py* le dice a Django qué páginas construir en respuesta a las peticiones del navegador. El archivo *wsgi.py* ayuda a Django a servir los archivos que crea. El nombre del archivo es un acrónimo de *web server gateway interface*.

Creación de la base de datos

Debido a que Django almacena la mayor parte de la información relacionada con un proyecto en una base de datos, necesitamos crear una base de datos con la que Django pueda trabajar. Para crear la base de datos para el proyecto Learning Log, introduce el siguiente comando (aún en un entorno activo):

```
(ll_env)learning_log$ python manage.py migrate
❶ Operaciones a realizar:
    Sincronizar aplicaciones no migradas: mensajes, archivos estáticos
    Aplicar todas las migraciones: contenttypes, sessions, auth, admin
    --snip--
    Aplicando sessions.0001_initial... OK
❷ (ll_env)learning_log$ ls
db.sqlite3 learning_log ll_env manage.py
```

> Cada vez que modificamos una base de datos, decimos que estamos *migrando* la base de datos.

Ejecutar el comando `migrate` por primera vez le dice a Django que se asegure de que la base de datos coincide con el estado actual del proyecto. La primera vez que ejecutamos este comando en un nuevo proyecto que utiliza SQLite (más sobre SQLite en un momento), Django creará una nueva base de datos para nosotros. En ❶ Django informa que hará las tablas de la base de datos necesarias para almacenar la información que usaremos en este proyecto (*Sincronizar apps no migradas*), y luego se asegurará de que la estructura de la base de datos coincide con el código actual (*Aplicar todas las migraciones*).

Ejecutar el comando `ls` muestra que Django creó otro archivo llamado *db.sqlite3* ❷. SQLite es una base de datos que se ejecuta desde un único archivo; es ideal para escribir aplicaciones sencillas porque no tendrás que prestar mucha atención a la gestión de la base de datos.

Visualización del proyecto

Asegurémonos de que Django ha configurado el proyecto correctamente. Introduce el archivo `runserver` de la siguiente manera:

```
(ll_env)learning_log$ python manage.py runserver
Realizar comprobaciones del sistema...

❶ La comprobación del sistema no identificó problemas (0 silenciados).
15 de julio de 2015 - 06:23:51
❷ Django versión 1.8.4, usando ajustes 'learning_log.settings'
❸ Iniciando el servidor de desarrollo en
http://127.0.0.1:8000/ Salga del servidor con CONTROL-C.
```

Django inicia un servidor para que puedas ver el proyecto en tu sistema y comprobar su funcionamiento. Cuando solicitas una página introduciendo una URL en un navegador, el servidor Django responde a esa solicitud construyendo la página apropiada y enviando esa página al navegador.

En ❶ Django comprueba que el proyecto está configurado correctamente; en ❷ informa de la versión de Django en uso y el nombre del archivo de configuración que se está utilizando; y en ❸ informa de la URL donde se está sirviendo el proyecto. La URL *http://127.0.0.1:8000/* indica que el proyecto está escuchando peticiones en el puerto 8000 de tu ordenador-llamado localhost. El término *localhost* se refiere a un servidor que sólo procesa peticiones en tu sistema; no permite que nadie más vea las páginas que estás desarrollando.

Ahora abre un navegador web e introduce la URL http://localhost:8000/, o *http://127.0.0.1:8000/* si la primera no funciona. Deberías ver algo como la Figura 18-1, una página que Django crea para hacerte saber que todo está funcionando correctamente hasta ahora. Mantén el servidor funcionando por ahora, pero cuando quieras parar el servidor puedes hacerlo pulsando CTRL-C.

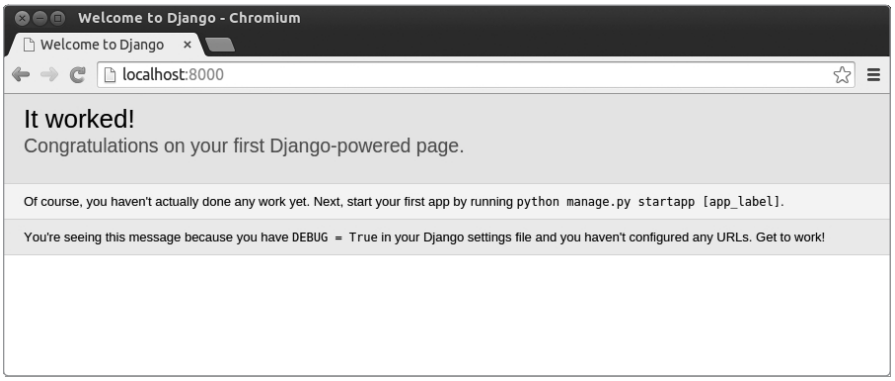


Figura 18-1: Hasta ahora todo funciona.

NOTE *Si recibes el mensaje de error* Ese puerto ya está en uso*, dile a Django que use un puerto diferente introduciendo* `python manage.py runserver 8001` *y haz un ciclo a través de números más altos hasta que encuentres un puerto abierto.*

TRY IT yOURSELF

18-1. Nuevos proyectos: Para tener una mejor idea de lo que hace Django, construye un par de proyectos vacíos y mira lo que crea. Crea una nueva carpeta con un nombre simple, como *InstaBook* o *FaceGram* (fuera de tu directorio *learning_log*), navega a esa carpeta en un terminal y crea un entorno virtual. Instala Django, y ejecuta el comando `django-admin.py startproject instabook .` (asegúrate de incluir el punto al final del comando).

Mira los archivos y carpetas que este comando crea, y compáralos con el Registro de Aprendizaje. Haz esto unas cuantas veces hasta que estés familiarizado con lo que Django crea al iniciar un nuevo proyecto. Luego borra los directorios del proyecto si lo deseas.

Iniciar una aplicación

Un *proyecto* Django está organizado como un grupo de *aplicaciones* individuales que trabajan juntas para hacer que el proyecto funcione como un todo. Por ahora, crearemos sólo una aplicación para hacer la mayor parte del trabajo de nuestro proyecto. Añadiremos otra aplicación para gestionar las cuentas de usuario en el Capítulo 19.

Deberías seguir ejecutando `runserver` en la ventana de terminal que abriste antes. Abre una nueva ventana de terminal (o pestaña) y navega hasta el directorio que contiene *manage.py*. Activa el entorno virtual y ejecuta el comando `startapp`:

```
learning_log$ source ll_env/bin/activate
(ll_env)learning_log$ python manage.py startapp learning_logs
❶ (ll_env)learning_log$ ls
  db.sqlite3 learning_log learning_logs ll_env manage.py
❷ (ll_env)learning_log$ ls learning_logs/
  admin.py init .py migraciones models.py tests.py views.py
```

El comando `startapp` *appname* le dice a Django que cree la infraestructura necesaria para construir una app. Si miras ahora en el directorio del proyecto, verás una nueva carpeta llamada *learning_logs* ❶. Abre esa carpeta para ver lo que Django ha creado ❷. Los archivos más importantes son *models.py*, *admin.py*, y *views.py*. Usaremos *models.py* para definir los datos que queremos gestionar en nuestra app. Llegaremos a *admin.py* y views*.py* un poco más tarde.

Definición de modelos

Pensemos un momento en nuestros datos. Cada usuario deberá crear una serie de temas en su registro de aprendizaje. Cada entrada estará vinculada a un tema y se mostrará como texto. También necesitaremos almacenar la fecha y hora de cada entrada para poder mostrar a los usuarios cuándo hicieron cada entrada.

Abre el archivo *models.py*, y mira su contenido existente:

modelos.py

```
from django.db import modelos

# Crea tus modelos aquí.
```

Un módulo llamado `models` se importa para nosotros, y se nos invita a crear nuestros propios modelos. Un modelo le dice a Django cómo trabajar con los datos que se almacenarán en la aplicación. Desde el punto de vista del código, un modelo es sólo una clase; tiene atributos y métodos, como todas las clases que hemos discutido. Este es el modelo para los temas que almacenarán los usuarios:

```
from django.db import models

class Tema(modelos.Modelo):
```

```
    """Un tema sobre el que el usuario está aprendiendo"""
        ❶ text = models.CharField(max_length=200)
        ❷ date_added = models.DateTimeField(auto_now_add=True)
```

```
❸    def str (self):
         """Devuelve una representación de cadena del
         modelo."""  return self.text
```

Hemos creado una clase llamada `Topic`, que hereda de `Model-una` clase padre incluida en Django que define la funcionalidad básica de un modelo. Sólo hay dos atributos en la clase `Topic`: `text` y `date_added`.

El atributo `text` es un `CharField-un` elemento de datos formado por caracteres, o texto ❶. Usas `CharField` cuando quieres almacenar una pequeña cantidad de texto, como un nombre, un título o una ciudad. Cuando definimos un atributo `CharField`, tenemos que decirle a Django cuánto espacio debe reservar en la base de datos. Aquí le damos un `max_length` de 200 caracteres, que debería ser suficiente para contener la mayoría de los nombres de temas.

El atributo `date_added` es un `DateTimeField-un` dato que registrará una fecha y hora ❷. Pasamos el argumento `auto_add_now=True`, que indica a Django que establezca automáticamente este atributo con la fecha y hora actuales siempre que el usuario cree un nuevo tema.

NOTE *Para ver los diferentes tipos de campos que puedes utilizar en un modelo, consulta la* Referencia de campos de modelo de Django *en* https://docs.djangoproject.com/en/1.8/ref/models/ fields/. *No necesitarás toda la información ahora mismo, pero te será extremadamente útil cuando estés desarrollando tus propias aplicaciones.*

Necesitamos decirle a Django qué atributo usar por defecto cuando muestra información sobre un tema. Django llama a un método `str ()` para mostrar una representación simple de un modelo. Aquí hemos escrito un método `str ()` que devuelve la cadena almacenada en el atributo de `texto` ❸.

NOTE *Si estás usando Python 2.7, deberías llamar al método* `str ()` `unicode ()` *en su lugar. El cuerpo del método es idéntico.*

Activación de modelos

Para utilizar nuestros modelos, tenemos que decirle a Django que incluya nuestra app en el proyecto general. Abre *settings.py* (en el directorio *learning_log/learning_log*), y verás una sección que le dice a Django qué apps están instaladas en el proyecto:

settings.py

```
--snip--
APLICACIONES_INSTALADAS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
```

```
    'django.contrib.staticfiles',
)
--snip--
```

Esto es sólo una tupla, diciéndole a Django qué aplicaciones trabajan juntas para formar el proyecto. Añade nuestra aplicación a esta tupla modificando `INSTALLED_APPS` para que se vea así:

```
--snip--
INSTALLED_APPS = (
    --snip--
    'django.contrib.staticfiles',

    # Mis aplicaciones
    'learning_logs',
)
--snip--
```

Agrupar las aplicaciones en un proyecto ayuda a realizar un seguimiento de las mismas a medida que el proyecto crece e incluye más aplicaciones. Aquí empezamos una sección llamada *Mis aplicaciones*, que por ahora sólo incluye `learning_logs`.

A continuación, tenemos que decirle a Django que modifique la base de datos para que pueda almacenar información relacionada con el modelo `Topic`. Desde el terminal, ejecuta el siguiente comando:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migraciones para 'learning_logs':
  0001_initial.py:
    - Crear modelo Tema
(ll_env)learning_log$
```

El comando `makemigrations` le dice a Django que averigüe cómo modificar la base de datos para que pueda almacenar los datos asociados con los nuevos modelos que hemos definido. La salida aquí muestra que Django ha creado un archivo de migración llamado *0001_initial.py*. Esta migración creará una tabla para el modelo `Topic` en la base de datos.

Ahora aplicaremos esta migración y haremos que Django modifique la base de datos por nosotros:

```
(ll_env)learning_log$ python manage.py migrate
--snip--
Ejecución de migraciones:
  Renderizado de los estados del modelo... HECHO
❶ Aplicando learning_logs.0001_initial... OK
```

La mayor parte de la salida de este comando es idéntica a la salida de la primera vez que ejecutamos el comando `migrate`. La línea que necesitamos comprobar aparece en ❶, donde Django confirma que todo funcionó bien cuando aplicó la migración para `learning_logs`.

Siempre que queramos modificar los datos que gestiona Learning Log, seguiremos estos tres pasos: modificar *models.py*, llamar a `makemigrations` sobre `learning_logs`, y decirle a Django que migre el proyecto.

El sitio de administración de Django

Cuando defines modelos para una aplicación, Django te facilita trabajar con tus modelos a través del sitio de *administración*. Los administradores de un sitio usan el sitio de administración, no los usuarios en general. En esta sección, configuraremos el sitio de administración y lo utilizaremos para añadir algunos temas a través del modelo `Topic`.

Configurar un superusuario

Django permite crear un usuario que tiene todos los privilegios disponibles en el sitio, llamado *superusuario*. Un *privilegio* controla las acciones que un usuario puede realizar. La configuración de privilegios más restrictiva permite a un usuario leer únicamente la información pública del sitio. Los usuarios registrados suelen tener el privilegio de leer sus propios datos privados y alguna información seleccionada que sólo está disponible para los miembros. Para administrar eficazmente una aplicación web, el propietario del sitio suele necesitar acceso a toda la información almacenada en el sitio. Un buen administrador es cuidadoso con la información sensible de sus usuarios, porque los usuarios depositan mucha confianza en las aplicaciones a las que acceden.

Para crear un superusuario en Django, introduce el siguiente comando y responde a las indicaciones:

```
(ll_env)learning_log$ python manage.py createsuperuser
❶ Nombre de usuario (dejar en blanco para usar 'ehmatthes'): ll_admin
❷ Dirección de correo electrónico:
❸ Contraseña:
Contraseña (otra
vez):
Superusuario creado con éxito.
(ll_env)learning_log$
```

Cuando ejecutas el comando `createsuperuser`, Django te pide que introduzcas un nombre de usuario para el superusuario ❶. Aquí estamos usando *ll_admin*, pero puedes introducir cualquier nombre de usuario que desees. Puedes introducir una dirección de correo electrónico si quieres o simplemente dejar este campo en blanco ❷. Tendrás que introducir tu contraseña dos veces ❸.

NOTE *Cierta información sensible puede ocultarse a los administradores de un sitio. Por ejemplo, Django no almacena realmente la contraseña que introduces; en su lugar, almacena una cadena derivada de la contraseña, llamada* hash*. Cada vez que introduces tu contraseña, Django hace un hash de tu entrada y la compara con el hash almacenado. Si los dos hashes coinciden, estás autenticado. Al requerir que los hashes coincidan, si un atacante obtiene acceso a la base de datos de un sitio, podrá leer los hashes almacenados pero no las contraseñas. Cuando un sitio está configurado correctamente, es casi imposible obtener las contraseñas originales a partir de los hashes.*

Registro de un modelo en el sitio de administración

Django incluye algunos modelos en el sitio de administración de forma automática, como `Usuario` y `Group`, pero los modelos que creamos deben registrarse manualmente.

Cuando iniciamos la aplicación `learning_logs`, Django creó un archivo llamado *admin.py* en el mismo directorio que *models.py*:

admin.py

```
from django.contrib import admin

# Registra tus modelos aquí.
```

Para registrar `Topic` en el sitio de administración, introduzca:

```
from django.contrib import admin

❶ from learning_logs.models import Tema.

❷ admin.site.register(Tema)
```

Este código importa el modelo que queremos registrar, `Topic` ❶, y luego utiliza `admin.site.register()` ❷ para decirle a Django que gestione nuestro modelo a través del sitio de administración.

Ahora usa la cuenta de superusuario para acceder al sitio admin. Ve a *http:// localhost:8000/admin/*, introduce el nombre de usuario y la contraseña para el superusuario que acabas de crear, y deberías ver una pantalla como la de la Figura 18-2. Esta página te permite añadir nuevos usuarios y grupos y cambiar los existentes. También podemos trabajar con datos relacionados con el modelo de `temas` que acabamos de definir.



Figura 18-2: El sitio de administración con el tema incluido

NOTE *Si ves un mensaje en tu navegador indicando que la página web no está disponible, asegúrate de que todavía tienes el servidor Django ejecutándose en una ventana de terminal. Si no es así, activa un entorno virtual y vuelve a ejecutar el comando python manage.py runserver.*

Añadir temas

Ahora que el `tema` ha sido registrado en el sitio de administración, vamos

a añadir nuestro primer tema. Haga clic en **Temas** para ir a la página Temas, que está casi vacía, porque todavía no tenemos temas que gestionar. Haga clic en **Añadir**, y verá un formulario para añadir

un nuevo tema. Introduce **Ajedrez** en la primera casilla y haz clic en **Guardar**. Volverás a la página de administración de Temas y verás el tema que acabas de crear.

Vamos a crear un segundo tema para tener más datos con los que trabajar. Haz clic en **Añadir** de nuevo, y crea un segundo tema, **Escalada**. Cuando hagas clic en **Guardar,** volverás a la página principal de Temas, y verás tanto Ajedrez como Escalada en la lista.

Definir el modelo de entrada

Para registrar lo que hemos estado aprendiendo sobre ajedrez y escalada, necesitamos definir un modelo para los tipos de entradas que los usuarios pueden hacer en sus registros de aprendizaje. Cada entrada debe estar asociada a un tema concreto. Esta relación se denomina *relación de muchos a uno*, lo que significa que muchas entradas pueden asociarse a un tema.

Este es el código del modelo de `entrada`:

modelos.py

```
from django.db import modelos

clase Topic(models.Model):
    --snip--

❶ clase Entrada(modelos.Modelo):
    """Algo específico aprendido sobre un tema"""
❷   topic = models.ForeignKey(Topic)
❸   text = models.TextField()
    date_added = models.DateTimeField(auto_now_add=True)

❹   clase Meta:
        verbose_name_plural = 'entradas'

     def str (self):
 """Devuelve una representación en forma de cadena del modelo."""
❺       return self.text[:50] + "..."
```

La clase `Entry` hereda de la clase base `Model` de Django, al igual que `Topic` ❶. El primer atributo, `topic`, es una instancia `ForeignKey` ❷. Una clave *foránea* es un término de base de datos; es una referencia a otro registro en la base de datos. Es la clave que conecta cada entrada con un tema específico. A cada tema se le asigna una clave, o ID, cuando se crea. Cuando Django necesita establecer una conexión entre dos piezas de datos, utiliza la clave asociada a cada pieza de información. En breve utilizaremos estas conexiones para recuperar todas las entradas asociadas a un determinado tema.

A continuación hay un atributo llamado `text`, que es una instancia de `TextField` ❸.

Este tipo de campo no necesita un límite de tamaño, porque no queremos limitar el tamaño de las entradas individuales. El atributo `date_added` nos permite presentar las entradas en el orden en que fueron creadas y colocar una marca de tiempo junto a cada entrada.

En ❹ anidamos la clase `Meta` dentro de nuestra clase `Entry`. `Meta` contiene

información extra para gestionar un modelo; aquí nos permite establecer un atributo especial que le dice a Django que use *Entries* cuando necesite referirse a más de una entrada.

(Sin esto, Django se referiría a múltiples entradas como *Entradas*.) Finalmente, el método `str()` le dice a Django qué información mostrar cuando se refiere a entradas individuales. Debido a que una entrada puede ser un cuerpo largo de texto, le decimos a Django que muestre sólo los primeros 50 caracteres de `texto` ❺. También añadimos una elipsis para aclarar que no siempre estamos mostrando la entrada completa.

Migración del modelo de entrada

Como hemos añadido un nuevo modelo, tenemos que volver a migrar la base de datos. Este proceso te resultará bastante familiar: modificas *models.py*, ejecutas el com- mando `python manage.py makemigrations` *`app_name`* y, a continuación, ejecutas el comando `python manage.py migrate`.

Migre la base de datos y compruebe el resultado:

```
(ll_env)learning_log$ python manage.py makemigrations learning_logs
Migraciones para 'learning_logs':
❶  0002_entry.py:
     - Crear modelo Entrada
(ll_env)learning_log$ python manage.py migrate
Operaciones a realizar:
  --snip--
❷  Aplicando learning_logs.0002_entry... OK
```

Se genera una nueva migración llamada *0002_entry.py*, que indica a Django cómo modificar la base de datos para almacenar información relacionada con el modelo `Entry` ❶. Cuando emitimos el comando `migrate`, vemos que Django aplicó esta migración, y todo estaba bien ❷.

Registro de entrada en el sitio de administración

También necesitamos registrar el modelo `Entry`. Este es el aspecto que debería tener *admin.py* ahora:

admin.py

```
from django.contrib import admin

from learning_logs.models import Tema, Entrada

admin.site.register(Tema)
admin.site.register(Entrada)
```

Vuelva a *http://localhost/admin/* y verá *Entradas* en *learning_logs*. Haga clic en el enlace **Añadir** entradas o en **Entradas** y, a continuación, seleccione **Añadir entrada**. Deberías ver una lista desplegable para seleccionar el tema para el que estás creando una entrada y un cuadro de texto para añadir una entrada. Selecciona **Ajedrez** en la lista desplegable y añade una entrada. Aquí está la primera entrada que hice:

> La apertura es la primera parte de la partida, aproximadamente las diez primeras jugadas. En la apertura, es buena idea hacer

tres cosas: sacar los alfiles y los caballos, intentar controlar el centro del tablero y enrocar el rey.

> Por supuesto, se trata sólo de directrices. Será importante aprender cuándo seguir estas directrices y cuándo hacer caso omiso de estas sugerencias.

Cuando haga clic en **Guardar,** volverá a la página principal de administración de entradas. Aquí verá la ventaja de utilizar `text[:50]` como cadena de representación de cada entrada; es mucho más fácil trabajar con varias entradas en la interfaz de administración si sólo ve la primera parte de una entrada en lugar del texto completo de cada entrada.

Haz una segunda entrada para Ajedrez y una entrada para Escalada para que tengamos algunos datos iniciales. Aquí tienes una segunda entrada para Ajedrez:

> En la fase inicial de la partida, es importante sacar a relucir sus alfiles y caballos. Estas piezas son lo suficientemente poderosas y maniobrables como para desempeñar un papel importante en las primeras jugadas de una partida.

Y aquí está la primera entrada para Escalada en Roca:

> Uno de los conceptos más importantes en la escalada es mantener el peso sobre los pies tanto como sea posible. Existe el mito de que los escaladores pueden colgarse todo el día de los brazos. En realidad, los buenos escaladores han practicado formas específicas de mantener su peso sobre los pies siempre que sea posible.

Estas tres entradas nos darán algo con lo que trabajar mientras seguimos desarrollando Learning Log.

La cáscara de Django

Ahora que hemos introducido algunos datos, podemos examinarlos programáticamente a través de una sesión de terminal interactiva. Este entorno interactivo se llama el *shell de* Django, y es un gran entorno para probar y solucionar problemas de tu proyecto. Aquí hay un ejemplo de una sesión de shell interactiva:

```
(ll_env)learning_log$ python manage.py shell
❶ >>> from learning_logs.models import Tema
>>> Topic.objects.all()
[<Tema: Ajedrez>, <Tema: Escalada>]
```

El comando `python manage.py shell` (ejecutado en un entorno virtual activo) lanza un intérprete de Python que puede utilizar para explorar los datos almacenados en la base de datos de su proyecto. Aquí importamos el modelo `Topic` del módulo `learning_logs.models` ❶. A continuación, utilizamos el método `Topic.objects.all()` para obtener todas las instancias del modelo `Topic`; la lista que se devuelve se denomina *queryset*.

Podemos hacer un bucle sobre un conjunto de consultas del mismo modo que lo haríamos sobre una lista. Así es como puedes ver el ID que se ha asignado a cada objeto de tema:

```
>>> topics = Topic.objects.all()
>>> para tema en temas:
```

```
... print(id.tema, tema)
...
1 Ajedrez
2 Escalada
```

Almacenamos el conjunto de consultas en `temas` y, a continuación, imprimimos el atributo `id` de cada tema y la representación de cadena de cada tema. Podemos ver que Ajedrez tiene un ID de 1, y Escalada tiene un ID de 2.

Si conoce el ID de un objeto en particular, puede obtener ese objeto y examinar cualquier atributo que tenga el objeto. Veamos los valores `text` y `date_added` de Chess:

```
>>> t = Topic.objects.get(id=1)
>>> t.text
Ajedrez
>>> t.fecha_agregada
datetime.datetime(2015, 5, 28, 4, 39, 11, 989446, tzinfo=<UTC>)
```

También podemos ver las entradas relacionadas con un tema determinado. Anteriormente definimos el atributo `topic` para el modelo `Entry`. Se trata de una `ForeignKey`, una conexión entre cada entrada y un tema. Django puede utilizar esta conexión para obtener todas las entradas relacionadas con un tema determinado, así:

```
❶ >>> t.entry_set.all()
[<Entrada: La apertura es la primera parte del juego, aproximadamente...>,
<Entrada: En la fase de apertura del juego, es importante ...>]
```

Para obtener datos a través de una relación de clave foránea, se utiliza el nombre en minúsculas del modelo relacionado seguido de un guión bajo y la palabra `conjunto` ❶. Por ejemplo, digamos que tienes los modelos `Pizza` y `Topping`, y `Topping` está relacionado con `Pizza` a través de una clave foránea. Si tu objeto se llama `mi_pizza`, representando una única pizza, puedes obtener todos los toppings de la `pizza` utilizando el código `mi_pizza.topping_set.all()`.

Utilizaremos este tipo de sintaxis cuando empecemos a codificar las páginas que los usuarios pueden solicitar. El shell es muy útil para asegurarte de que tu código recupera los datos que deseas. Si tu código funciona como esperas en el shell, puedes esperar que funcione correctamente en los archivos que escribas dentro de tu proyecto. Si tu código genera errores o no recupera los datos que esperas, es mucho más fácil solucionar los problemas de tu código en el sencillo entorno del shell que dentro de los archivos que generan las páginas web. No nos referiremos mucho al shell, pero deberías seguir usándolo para practicar el trabajo con la sintaxis de Django para acceder a los datos almacenados en el proyecto.

NOTA *Cada vez que modifiques tus modelos, necesitarás reiniciar el shell para ver los efectos de esos cambios. Para salir de una sesión de shell, introduce* cTRL-D*; en Windows introduce cTRL-Z y luego pulsa* ENTER*.*

TRY IT yOURSELF

18-2. Entradas cortas: El método `str()` en el modelo `Entry` actualmente añade una elipsis a cada instancia de `Entry` cuando Django la muestra en el sitio admin
o el shell. Añade una sentencia `if` al método `str()` que añada una elipsis sólo si la entrada tiene más de 50 caracteres. Utiliza el sitio de administración para añadir una entrada que tenga menos de 50 caracteres y comprueba que no tiene elipsis cuando se visualiza.

18-3. La API de Django: Cuando escribes código para acceder a los datos de tu proyecto, estás escribiendo una *consulta*. Hojea la documentación para consultar tus datos en *https://docs.djangoproject.com/en/1.8/topics/db/queries/*. Mucho de lo que veas te parecerá nuevo, pero te será bastante útil cuando empieces a trabajar en tus propios proyectos.

18-4. Pizzería: Comience un nuevo proyecto llamado `pizzería` con una aplicación llamada `pizzas`. Defina un modelo `Pizza` con un campo llamado `nombre`, que contendrá valores de nombre como `Hawaiian` y `Meat Lovers`. Defina un modelo llamado `Topping` con campos llamados `pizza` y `name`. El campo `pizza` debe ser una clave externa a `Pizza`, y `nombre` debe poder contener valores como `piña`, `bacon canadiense` y `salchicha`.

Registra ambos modelos en el sitio de administración, y utiliza el sitio para introducir algunos nombres de pizza y coberturas. Utiliza el shell para explorar los datos introducidos.

Haciendo páginas: Página de inicio del Diario de aprendizaje

Normalmente, hacer páginas web con Django consiste en tres etapas: definir URLs, escribir vistas y escribir plantillas. Primero, debes definir patrones para las URLs. Un patrón de URL describe la forma en que la URL está dispuesta y le dice a Django qué buscar cuando hace coincidir una solicitud del navegador con la URL de un sitio para saber qué página devolver.

Cada URL se asigna a una *vista concreta: la función de* vista recupera y procesa los datos necesarios para esa página. La función de vista a menudo llama a una *plantilla*, que construye una página que un navegador puede leer. Para ver cómo funciona esto, vamos a crear la página de inicio de Learning Log. Definiremos la URL de la página de inicio, escribiremos su función de vista y crearemos una plantilla sencilla.

Como lo único que estamos haciendo es asegurarnos de que Learning Log funcione como es debido, por ahora mantendremos la página simple. Una aplicación web que funciona es divertida cuando está completa; una aplicación que tiene buen aspecto pero no funciona bien
es inútil. Por ahora, la página de inicio sólo mostrará un título y una breve descripción.

Asignación de una URL

Los usuarios solicitan las páginas introduciendo las URL en un navegador y haciendo clic en los enlaces, por lo que tendremos que decidir qué URL son necesarias en nuestro proyecto. La página de inicio La URL de la página es lo primero: es la URL base que la gente utiliza para acceder al proyecto. Por el momento, la URL base, http://localhost:8000/, devuelve el sitio Django por defecto que nos permite saber que el proyecto se ha configurado correctamente. Cambiaremos esto mapeando la URL base a la página de inicio de Learning Log.

En la carpeta principal del proyecto *learning_log*, abra el archivo *urls.py*. Este es el código que verá:

urls.py

```
❶ from django.conf.urls import include, url
  from django.contrib import admin

❷ urlpatterns = [
❸     url(r'^admin/', include(admin.site.urls)),
  ]
```

Las dos primeras líneas importan las funciones y módulos que gestionan URLs para el proyecto y el sitio de administración ❶. El cuerpo del archivo define la variable `urlpatterns` ❷. En este archivo *urls.py,* que representa el proyecto en su conjunto, la variable `urlpatterns` incluye conjuntos de URLs de las apps del proyecto. El código en ❸ incluye el módulo `admin.site.urls`, que define todas las URLs que se pueden solicitar desde el sitio de administración.

Debemos incluir las direcciones URL de `learning_logs`:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
❶     url(r'', include('learning_logs.urls', namespace='learning_logs')),
]
```

Hemos añadido una línea para incluir el módulo `learning_logs.urls` en ❶. Esta línea incluye un argumento `namespace`, que nos permite distinguir las URLs de learning_logs de otras URLs que puedan aparecer en el proyecto, lo que puede ser muy útil cuando tu proyecto empiece a crecer.

El *archivo urls.py* predeterminado se encuentra en la *carpeta learning_log*; ahora debemos crear un segundo archivo *urls.py* en la carpeta *learning_logs*:

urls.py

```
❶ """Define patrones de URL para learning_logs."""

❷ from django.conf.urls import url

❸ from . import vistas
```

```
❹ urlpatterns = [
# Página de inicio
❺     url(r'^$', views.index, name='index'),
  ]
```

Para dejar claro en qué *urls.py estamos* trabajando, añadimos un docstring a l principio del archivo ❶. A continuación importamos la función `url`, necesaria para mapear URLs a vistas ❷. También importamos las `vistas` mod- ule ❸; el punto le dice a Python que importe vistas del mismo directorio que el módulo *urls.py* actual. La variable `urlpatterns` en este módulo es una lista de páginas individuales que se pueden solicitar desde la app `learning_logs` ❹.

El patrón URL real es una llamada a la función `url()`, que toma tres argumentos ❺. El primero es una expresión regular. Django buscará una expresión regular en `urlpatterns` que coincida con la cadena URL solicitada. Por lo tanto, una expresión regular definirá el patrón que Django puede buscar.

Veamos la expresión regular `r'^$'`. La `r` le dice a Python que inter- prete la siguiente cadena como una cadena sin procesar, y las comillas le dicen a Python dónde empieza y termina la expresión regular. El signo de intercalación (^) le dice a Python que busque el principio de la c a d e n a , y el signo de dólar le dice a Python que busque el final de la cadena. En su totalidad, esta expresión le dice a Python que busque una URL sin nada entre el principio y el final de la URL. Python ignora la URL base del proyecto *(*http://localhost:8000/), por lo que una expresión regular vacía coincide con la URL base. Cualquier otra URL no coincidirá con esta expresión, y Django devolverá una página de error si la URL solicitada no coincide con ningún patrón de URL existente.

El segundo argumento de `url()` en ❺ especifica qué función de vista a llamar. Cuando una URL solicitada coincida con la expresión regular, Django llamará a `views.index` (escribiremos esta función de vista en la siguiente sección). El tercer argumento proporciona el nombre `index` para este patrón de URL para que podamos referirnos a él en otras secciones del código. Siempre que queramos proporcionar un enlace a la página de inicio, utilizaremos este nombre en lugar de escribir una URL.

NOTE *Las expresiones regulares, a menudo llamadas* regexes*, se utilizan en casi todos los lenguajes de programación. Son increíblemente útiles, pero se necesita algo de práctica para acostumbrarse a ellas. Si no has seguido todo esto, no te preocupes; verás muchos ejemplos a medida que trabajes en este proyecto.*

Escribir una vista

Una función de vista recibe información de una solicitud, prepara los datos necesarios para generar una página y, a continuación, envía los datos de vuelta al navegador, a menudo utilizando una plantilla que define el aspecto que tendrá la página.

El archivo views*.py* en *learning_logs* se generó automáticamente cuando ejecutamos el comando `python manage.py startapp`. Esto es lo que hay en *views.py* ahora mismo:

vistas.py

```
from django.shortcuts import render

# Crea tus vistas aquí.
```

Actualmente, este archivo sólo importa la función `render()`, que renderiza la respuesta basándose en los datos proporcionados por las vistas. El siguiente código es cómo debe escribirse la vista para la página de inicio:

```
from django.shortcuts import render

def index(request):
    """La página de inicio de Learning Log"""
    return render(request, 'learning_logs/index.html')
```

Cuando una petición URL coincide con el patrón que acabamos de definir, Django buscará una función llamada `index()` en el archivo *views.py*. Django entonces pasa el objeto de `la petición` a esta función de vista. En este caso, no necesitamos procesar ningún dato para la página, por lo que el único código en la función es una llamada a render(). La función `render()` utiliza aquí dos argumentos: el objeto de `petición` original y una plantilla que puede utilizar para construir la página. Escribamos esta plantilla.

Escribir una plantilla

Una plantilla establece la estructura de una página web. La plantilla define el aspecto que debe tener la página, y Django rellena los datos relevantes cada vez que se solicita la página. Una plantilla te permite acceder a cualquier dato proporcionado por la vista. Debido a que nuestra vista para la página de inicio no proporciona datos, esta plantilla es bastante simple.

Dentro de la carpeta *learning_logs*, cree una nueva carpeta llamada *templates*.

Dentro de la carpeta *templates*, crea otra carpeta llamada *learning_logs*. Esto puede parecer un poco redundante (tenemos una carpeta llamada *learning_logs* dentro de una carpeta llamada *templates* dentro de una carpeta llamada *learning_logs*), pero establece
de una estructura que Django pueda interpretar sin ambigüedades, incluso en el contexto de un gran proyecto que contenga muchas aplicaciones individuales. Dentro de la carpeta *learning_logs*, crea un nuevo archivo llamado *index.html*. Escribe lo siguiente en ese archivo:

index.html

```
<p>Registro de aprendizaje</p>

<p>Learning Log te ayuda a llevar un registro de tu aprendizaje, para cualquier
tema sobre el que estés aprendiendo.</p> <p>Learning Log te ayuda a llevar un
registro de tu aprendizaje, para cualquier tema sobre el que estés aprendiendo.
```

Se trata de un archivo muy sencillo. Si no estás familiarizado con HTML, las etiquetas `<p></p>` significan párrafos. La etiqueta `<p>` abre un párrafo y la etiqueta `</p>` lo cierra. Tenemos dos párrafos: el primero sirve de título y el segundo describe lo que los usuarios pueden hacer con Learning Log.

Ahora cuando solicitemos la URL base del proyecto, http://localhost:8000/, veremos la página que acabamos de construir en

lugar de la página por defecto de Django. Django tomará la URL solicitada, y esa URL coincidirá con el patrón `r'^$'`; entonces Django llamará a la función `views.index()`, y esto renderizará la página usando la plantilla contenida en *index.html*. La página resultante se muestra en la Figura 18-3.

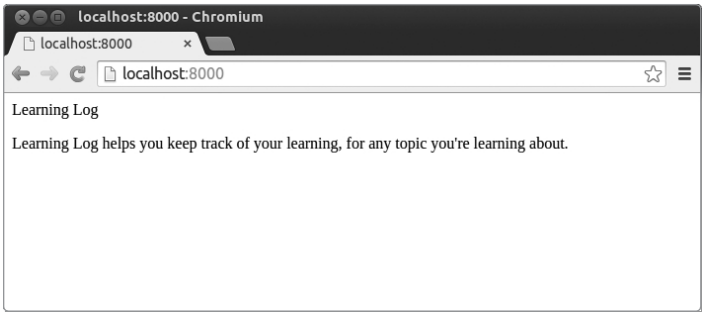


Figura 18-3: Página de inicio de Learning Log

Aunque pueda parecer un proceso complicado para crear una página, esta separación entre URL, vistas y plantillas funciona realmente bien. Permite pensar en cada aspecto de un proyecto por separado y, en proyectos más grandes, permite a las personas centrarse en las áreas en las que son más fuertes. Por ejemplo, un especialista en bases de datos puede centrarse en los modelos, un programador en el código de las vistas y un diseñador web en las plantillas.

TRY IT yOURSELF

18-5. Planificador de comidas: Piensa en una aplicación que ayude a la gente a planificar sus comidas a lo largo de la semana. Crea una nueva carpeta llamada *meal_planner*, y comienza un nuevo proyecto Django dentro de esta carpeta. A continuación, crea una nueva aplicación llamada `meal_plans`. Haz una página de inicio simple para este proyecto.

18-6. Página de Inicio de la Pizzería: Añade una página de inicio al proyecto *Pizzería* que iniciaste en el Ejercicio 18-4 (página 412).

Creación de páginas adicionales

Ahora que hemos establecido una rutina para construir una página, podemos empezar a construir el proyecto Learning Log. Construiremos dos páginas que muestran datos: una página que enumera todos los temas y una página que muestra todas las entradas de un tema en particular. Para cada una de estas páginas, especificaremos un patrón URL, escribiremos una función de vista y escribiremos una plantilla. Pero antes de hacer esto, crearemos una plantilla base de la que podrán heredar todas las plantillas del proyecto.

Herencia de plantillas

Al construir un sitio web, casi siempre necesitará que algunos elementos se repitan en cada página. En lugar de escribir estos elementos directamente en cada página, puede escribir una plantilla base que

contenga los elementos repetidos

y luego hacer que cada página herede de la plantilla. Este enfoque le permite centrarse en el desarrollo de los aspectos únicos de cada página y hace que sea mucho más fácil cambiar el aspecto general del proyecto.

La plantilla de padres

Empezaremos creando una plantilla llamada base.*html* en el mismo directorio que *index.html*. Este archivo contendrá elementos comunes a todas las páginas; todas las demás plantillas heredarán de base.*html*. El único elemento que queremos repetir en cada página ahora mismo es el título en la parte superior. Como incluiremos esta plantilla en todas las páginas, hagamos que el título sea un enlace a la página de inicio*:*

base.html

```
<p>
❶  <a href="{% url 'learning_logs:index' %}">Registro de aprendizaje</a>
</p>

❷ {% block content %}{% endblock content %}
```

La primera parte de este archivo crea un párrafo que contiene el nombre del proyecto, que también actúa como enlace a la página de inicio. Para generar un enlace, utilizamos una *etiqueta de plantilla*, indicada por llaves y signos de porcentaje `{% %}`. Una etiqueta de plantilla es un fragmento de código que genera información que se mostrará en una página.
En este ejemplo, la etiqueta de plantilla `{% url 'learning_logs:index' %}` genera una URL que coincide con el patrón URL definido en *learning_logs/urls.py* con el nombre `'index'` ❶. En este ejemplo, `learning_logs` es el *espacio de nombres* e `index` es un patrón URL con nombre único en ese espacio de nombres.

En una página HTML sencilla, un enlace está rodeado por la etiqueta *anchor*:

```
<a href="link_url">texto del enlace</a>
```

Hacer que la etiqueta de plantilla genere la URL por nosotros hace que sea mucho más fácil mantener nuestros enlaces actualizados. Para cambiar una URL en nuestro proyecto, sólo tenemos que cambiar el patrón de URL en *urls.py*, y Django insertará automáticamente la URL actualizada la próxima vez que se solicite la página. Cada página de nuestro proyecto heredará de *base.html*, así que a partir de ahora cada página tendrá un enlace a la página de inicio.

En ❷ insertamos un par de etiquetas de `bloque`. Este bloque, llamado `contenido`, es un soporte; la plantilla hija definirá el tipo de información que va en el bloque de `contenido`.

Una plantilla hija no tiene por qué definir todos los bloques de su padre, por lo que puede reservar espacio en las plantillas padre para tantos bloques como desee, y la plantilla hija sólo utilizará los que necesite.

NOTA *En código Python, casi siempre sangramos cuatro espacios. Los archivos de*

plantilla tienden a tener más niveles de anidamiento que los archivos Python, por lo que es común usar sólo dos espacios para cada nivel de sangría.

Plantilla infantil

Ahora tenemos que reescribir index.*html* para que herede de *base.html*. Aquí está index.*html*:

index.html

```
❶ {% extends "learning_logs/base.html" %}

❷ {% block content %}
    <p>Learning Log te ayuda a llevar un registro de tu aprendizaje, para
    cualquier tema sobre el que estés aprendiendo.</p> <p>Learning Log te ayuda a
    llevar un registro de tu aprendizaje, para cualquier tema sobre el que estés
    aprendiendo.
❸ {% endblock content %}
```

Si comparas esto con el *index.html original*, puedes ver que hemos reemplazado el título Learning Log con el código para heredar de una plantilla padre ❶. Una plantilla hija debe tener una etiqueta `{% extends %}` en la primera línea para indicarle a Django de qué plantilla padre debe heredar. El archivo *base.html* es parte de `learning_logs`, así que incluimos *learning_logs* en la ruta a la plantilla par- ticular. Esta línea extrae todo lo que contiene la plantilla base.*html* y permite que *index.html* defina lo que va en el espacio reservado por el bloque de `contenido`.

Definimos el bloque de contenido en ❷ insertando una etiqueta `{% block %}` con el nombre `content`. Todo lo que no estamos heredando de la plantilla padre va dentro de un bloque de `contenido`. Aquí, ese es el párrafo que describe el proyecto Learning Log. En ❸ indicamos que hemos terminado de definir el contenido usando una etiqueta `{% endblock content %}`.

Puedes empezar a ver el beneficio de la herencia de plantillas: en una plantilla hija sólo necesitamos incluir contenido que sea único para esa página. Esto no sólo simplifica cada plantilla, sino que también hace que sea mucho más fácil modificar el sitio. Para modificar un elemento común a varias páginas, basta con modificar el elemento en la plantilla padre. Los cambios se aplicarán a todas las páginas que hereden de esa plantilla. En un proyecto que incluye decenas o cientos de páginas, esta estructura puede hacer que sea mucho más fácil y más rápido para mejorar su sitio.

NOTE *En un proyecto grande, es común tener una plantilla padre llamada base.*html *para todo el sitio y plantillas padre para cada sección principal del sitio. Todas las plantillas de sección heredan de base.*html*, y cada página del sitio hereda de una sección*
plantilla. De este modo, puede modificar fácilmente el aspecto del sitio en su conjunto, de cualquier sección del sitio o de cualquier página individual. Esta configuración proporciona una forma muy eficaz de trabajar y le anima a actualizar su sitio de forma constante a lo largo del tiempo.

Página de temas

Ahora que tenemos un enfoque eficiente para construir páginas, podemos centrarnos en nuestras dos próximas páginas: la página general de temas y la página para mostrar las entradas de un único tema. La página de temas mostrará todos los temas que los usuarios han creado, y es la primera página que implicará trabajar con datos.

El patrón de URL de los temas

En primer lugar, definimos la URL de la página de temas. Es habitual elegir un fragmento de URL sencillo que refleje el tipo de información que se presenta en la página. Utilizaremos la palabra *temas*, por lo que la URL *http://localhost:8000/topics/* devolverá esta página. Así es como modificamos *learning_logs/urls.py*:

urls.py

```
"""Define patrones de URL para learning_logs."""
--snip--
urlpatterns = [
    # Página de inicio
    url(r'^$', views.index, name='index'),

    # Mostrar todos los temas.
❶   url(r'^topics/$', views.topics, name='topics'),
]
```

Simplemente hemos añadido `topics/` en el argumento de expresión regular utilizado para la URL de la página de inicio ❶. Cuando Django examina una URL solicitada, este patrón coincidirá con cualquier URL que tenga la URL base seguida de *topics*. Puedes incluir u omitir una barra diagonal al final, pero no puede haber nada más después de la palabra *topics*, o el patrón no coincidirá. Cualquier petición con una URL que coincida con este patrón será pasada a la función `topics()` en *views.py*.

La vista de temas

La función `topics()` necesita obtener algunos datos de la base de datos y enviarlos a la plantilla. Esto es lo que tenemos que añadir a *views.py*:

vistas.py

```
from django.shortcuts import render

❶ from .models import Tema

def index(solicitud):
    --snip--

❷ def temas(solicitud):
    """Mostrar todos los temas."""
❸   topics = Topic.objects.order_by('fecha_añadido')
❹   contexto = {'temas': temas}
❺   return render(request, 'learning_logs/topics.html', context)
```

Primero importamos el modelo asociado a los datos que necesitamos ❶. La función `topics()` necesita un parámetro: el objeto de petición que Django recibió del servidor ❷. En ❸ consultamos la base de datos pidiendo los objetos `Topic`, ordenados por el atributo `date_added`. Almacenamos el queryset resultante en `topics`.

En ❹ definimos un contexto que enviaremos a la plantilla. Un *contexto* es un diccionario en el que las claves son nombres que usaremos en la plantilla para acceder a los datos y los valores son los datos que necesitamos enviar a la plantilla. En este caso, hay un par clave-valor, que contiene el conjunto de temas que mostraremos en la página. Cuando construimos una página que utiliza datos, pasamos la variante `context` a `render()` así como el objeto `request` y la ruta a la plantilla ❺.

Plantilla de temas

La plantilla de la página de temas recibe el diccionario de `contexto` para que la plantilla pueda utilizar los datos que proporciona `topics()`. Crea un archivo llamado *topics.html* en el mismo directorio que *index.html*. Así es como podemos mostrar los temas en la plantilla:

temas.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}

  <p>Temas</p>

❶ <ul>
❷   {% for tema in temas %}
❸     <li>{{tema }}</li>
❹   {% empty %}
      <li>Aún no se ha añadido ningún tema.</li>.
❺   {% endfor %}
❻ </ul>

{% endblock content %}
```

Empezamos usando la etiqueta `{% extends %}` para heredar de *base.html*, tal y como hace la plantilla index, y luego abrimos un bloque de `contenido`. El cuerpo de esta página contiene una lista con viñetas de los temas que se han introducido. En HTML estándar, una lista con viñetas se denomina *lista desordenada*, indicada por las etiquetas `<ul></ul>`. Comenzamos la lista de temas con viñetas en ❶.

En ❷ tenemos otra etiqueta de plantilla equivalente a un bucle `for`, que recorre la lista de `temas` del diccionario de `contexto`. El código utilizado en las plantillas difiere de Python en algunos aspectos importantes. Python utiliza sangría para indicar qué líneas de una sentencia `for` forman parte de un bucle. En una plantilla, cada bucle `for` necesita una etiqueta `{% endfor %}` explícita que indique dónde se produce el final del bucle. Así que en una plantilla, verá bucles escritos así:

```
{% for item in list %}
    hacer algo con cada elemento
{% endfor %}
```

Dentro del bucle, queremos convertir cada tema en un elemento de la lista con viñetas. Para imprimir una variable en una plantilla, envuelva el

nombre de la variable en doble

llaves. El código `{{ topic }}` en ❸ será reemplazado por el valor de `topic` en cada pasada por el bucle. Los corchetes no aparecerán en la página; sólo indican a Django que estamos utilizando una variable de plantilla. La etiqueta HTML `<li></li>` indica un elemento de lista. Cualquier cosa entre estas etiquetas, dentro de un par de `<ul></ul>`, aparecerá como una viñeta en la lista.

En ❹ usamos la etiqueta de plantilla `{% empty %}`, que le dice a Django qué hacer si no hay temas en la lista. En este caso, imprimimos un mensaje informando al usuario de que aún no se ha añadido ningún tema. Las dos últimas líneas cierran el bucle `for` ❺ y luego cierran la lista con viñetas ❻.

Ahora tenemos que modificar la plantilla base para incluir un enlace a la página de temas:

base.html

```
<p>
❶  <a href="{% url 'learning_logs:index' %}">Registro de aprendizaje</a> -
❷  <a href="{% url 'learning_logs:topics' %}">Temas</a>
</p>

{% block content %}{% endblock content %}
```

Añadimos un guión después del enlace a la página de inicio ❶, y luego añadimos un enlace a la página de temas, utilizando de nuevo la etiqueta de plantilla URL ❷. Esta línea

indica a Django que genere un enlace que coincida con el patrón de URL con el nombre `topics'` en *learning_logs/urls.py*.

Ahora, cuando actualice la página de inicio en su navegador, verá un enlace *Temas*. Cuando haga clic en el enlace, verá una página similar a la de la Figura 18-4.

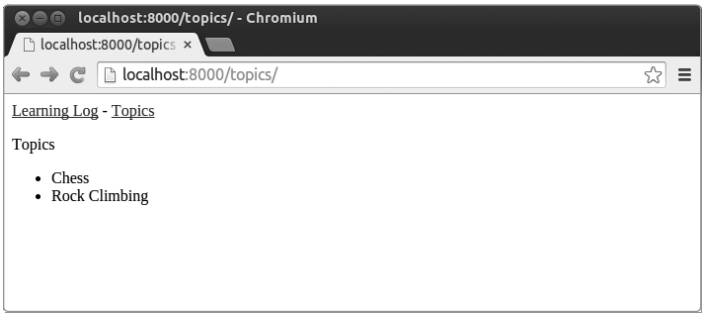


Figura 18-4: Página de temas

Páginas temáticas individuales

A continuación, necesitamos crear una página que pueda centrarse en un único tema, mostrando el nombre del tema y todas las entradas de ese tema. Definiremos de nuevo un nuevo patrón URL, escribiremos una vista y

crearemos una plantilla. También modificaremos la página de temas para que cada elemento de la lista con viñetas enlace a su correspondiente página de temas.

El patrón URL del tema

El patrón de URL para la página de temas es un poco diferente de los otros patrones de URL que hemos visto hasta ahora porque utilizará el atributo `id del tema` para indicar qué tema se ha solicitado. Por ejemplo, si el usuario quiere ver la página de detalles del tema Ajedrez, cuyo `id` es 1, la URL será *http:// localhost:8000/topics/1/. A* continuación se muestra un patrón para que coincida con esta URL, que va en *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Página detallada para un único tema
    url(r'^topics/(?P<topic_id>\d+)/$', views.topic, name='topic'),
]
```

Examinemos la expresión regular en este patrón de URL, `r'^topics/(?P<topic_id>\d+)/$'`. La `r` le dice a Django que interprete la cadena como una cadena sin procesar, y la expresión está contenida entre comillas. La segunda parte de la expresión, `/(?P<topic_id>\d+)/`, coincide con un número entero entre dos barras diagonales y almacena el valor entero en un argumento llamado `topic_id`. Los paréntesis que rodean esta parte de la expresión capturan el valor almacenado en la URL; la parte `?P<topic_id>` almacena el valor coincidente en `topic_id`; y la expresión `\d+` coincide con cualquier número de dígitos que aparezcan entre las barras inclinadas.

Cuando Django encuentra una URL que coincide con este patrón, llama a la función `topic()` de la vista con el valor almacenado en `topic_id` como argumento. Usaremos el valor de `topic_id` para obtener el topic correcto dentro de la función.

La vista de temas

La función `topic()` necesita obtener el tema y todas las entradas asociadas de la base de datos, como se muestra aquí:

views.py

```
--snip--
❶ def tema(solicitud, tema_id):
    """Mostrar un único tema y todas sus entradas."""
❷   topic = Topic.objects.get(id=topic_id)
❸   entradas = topic.entry_set.order_by('-fecha_agregada')
❹   context = {'tema': tema, 'entradas': entradas}
❺   return render(request, 'learning_logs/topic.html', context)
```

Esta es la primera función de vista que requiere un parámetro distinto del objeto de petición. La función acepta el valor capturado por la expresión `(?P<topic_id>\d+)` y lo almacena en `topic_id` ❶. En ❷ usamos `get()` para recuperar el topic, tal y como hicimos en el shell de Django. En ❸ obtenemos las entradas asociadas a este tema, y las ordenamos según `date_added`: el signo menos delante de `date_added` ordena los resultados en orden inverso, lo que mostrará primero las entradas más recientes. Almacenamos el tema y las entradas en el diccionario de contexto ❹ y enviamos el `contexto` a la plantilla

topic.html ❺.

NOTE *Las frases de código en ❷ y ❸ se llaman* consultas*, porque consultan la base de datos para obtener información específica. Cuando escribas consultas como estas en tus propios proyectos, es muy útil probarlas primero en el shell de Django. Obtendrás una respuesta mucho más rápida en el shell que escribiendo una vista y una plantilla y luego comprobando los resultados en un navegador.*

Plantilla de temas

La plantilla debe mostrar el nombre del tema y las entradas. También es necesario informar al usuario de si aún no se han realizado entradas para este tema:

topic.html

```
{% extends 'learning_logs/base.html' %}

{% block content %}

❶ <p>Tema: {{ topic }}</p>

  <p>Entradas:</p>
❷ <ul>
❸ {% for entrada in entradas %}
    <li>
❹     <p>{ entry.date_added|date:'M d, Y H:i' }}</p>
❺     <p>{{ entry.text|saltos de línea }}</p>
    </li> <li>
❻ {% empty %}
    <li>
      Aún no hay entradas para este tema.
    </li> <li>
  {% endfor %}
  </ul>

{% endblock content %}
```

Extendemos *base.html*, como hacemos para todas las páginas del proyecto. A continuación, mostramos el tema que se está mostrando actualmente ❶, que se almacena en la variable de plantilla `{{ topic }}`. El `tema` variable está disponible porque se incluye en el diccionario `contextual`. A continuación, iniciamos una lista con viñetas para mostrar cada una de las entradas ❷ y las recorremos en bucle como hicimos anteriormente con los temas ❸.

Cada viñeta mostrará dos informaciones: la marca de tiempo y el texto completo de cada entrada. Para la marca de tiempo ❹, mostramos el valor del atributo `date_added`. En las plantillas de Django, una línea vertical (`|`) representa un filtro *de plantilla*: una función que modifica el valor de una variable de plantilla. El filtro `date:'M d, Y H:i'` muestra marcas de tiempo en el formato *1 de enero de 2015 23:00*. La línea siguiente muestra el valor completo del `texto en` lugar de sólo los 50 primeros caracteres de la `entrada`. El filtro `saltos de línea` ❺ garantiza que las entradas de texto

largas incluyan saltos de línea en un formato que entiendan los navegadores en lugar de mostrar un bloque de texto ininterrumpido. En ❻ utilizamos la etiqueta de plantilla `{% empty %}` para imprimir un mensaje que informa al usuario de que no se han realizado entradas.

Enlaces de la página de temas

Antes de ver la página de temas en un navegador, necesitamos modificar la plantilla de temas para que cada tema enlace a la página apropiada. Aquí está el cambio a *topics.html*:

temas.html

```
--snip--
    {% for tema in temas %}
      <li>
        <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
      </li> <li>
    {% empty %}
--snip--
```

Utilizamos la etiqueta de plantilla de URL para generar el enlace adecuado, basándonos en el patrón de URL de `learning_logs` con el nombre `'topic'`. Este patrón de URL requiere un argumento `topic_id`, por lo que añadimos el atributo `topic.id` a la etiqueta de plantilla de URL. Ahora, cada tema de la lista de temas es un enlace a una página de temas, como http://localhost:8000/topics/1/.

Si actualiza la página de temas y hace clic en un tema, debería ver una página parecida a la Figura 18-5.

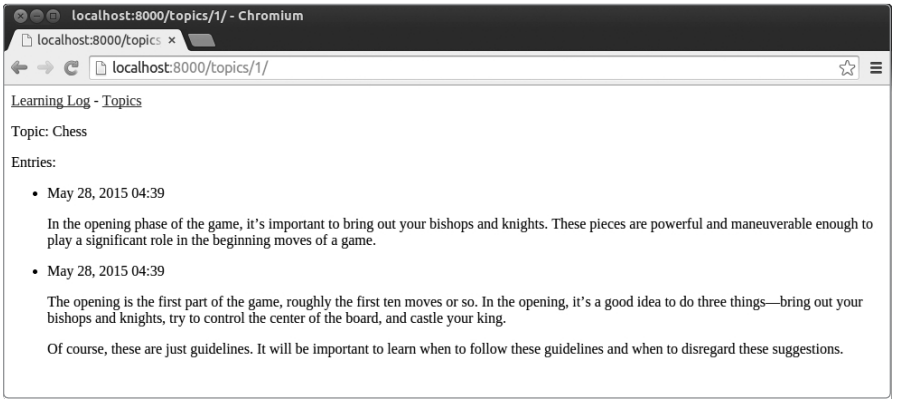


Figura 18-5: Página de detalles de un tema, que muestra todas las entradas de un tema

TRY IT yOURSELF

18-7. Documentación de plantillas: Hojea la documentación de plantillas de Django en https://docs.djangoproject.com/en/1.8/ref/templates/. Puedes consultarla cuando trabajes en tus propios proyectos.

18-8. Páginas de Pizzería: Añada una página al proyecto *Pizzería* del Ejercicio 18-6 (página 416) que muestre los nombres de las pizzas disponibles. Luego vincule cada nombre de pizza a una página que muestre los ingredientes de la pizza. Asegúrese de usar la herencia de plantillas para construir sus páginas eficientemente.

resumen

En este capítulo comenzaste a aprender cómo construir aplicaciones web utilizando el framework Django. Escribiste una breve especificación del proyecto, instalaste Django en un entorno virtual, aprendiste a configurar un proyecto y verificaste que el proyecto estuviera configurado correctamente. Aprendiste a configurar una aplicación y definiste modelos para representar los datos de tu aplicación. Aprendiste sobre bases de datos y cómo Django te ayuda a migrar tu base de datos después de hacer un cambio en tus modelos. Aprendiste cómo crear un superusuario para el sitio de administración, y usaste el sitio de administración para introducir algunos datos iniciales.

También exploraste el shell de Django, que te permite trabajar con los datos de tu proyecto en una sesión de terminal. Aprendiste a definir URLs, crear funciones de vista y escribir plantillas para crear páginas para tu sitio. Finalmente, usaste la herencia de plantillas para simplificar la estructura de plantillas individuales y para hacer más fácil modificar el sitio a medida que el proyecto evoluciona.

En el capítulo 19 crearemos páginas intuitivas y fáciles de usar que permitan a los usuarios añadir nuevos temas y entradas y editar las entradas existentes sin pasar por el sitio de administración. También añadiremos un sistema de registro de usuarios que les permita crear una cuenta y crear su propio registro de aprendizaje. Este es el corazón de una aplicación web: la capacidad de crear algo con lo que pueda interactuar cualquier número de usuarios.

19

ACCIONESUSERAS



En el corazón de una aplicación web está la posibilidad de que cualquier usuario, en cualquier parte del mundo, registre una cuenta en tu aplicación y empezar a utilizarlo. En este capítulo construirás formularios para que los usuarios puedan añadir sus propios temas y entradas, y editar las entradas existentes. También aprenderás cómo Django protege contra ataques comunes a páginas basadas en formularios para que no tengas que pasar mucho tiempo pensando en asegurar tus aplicaciones.

A continuación, implementaremos un sistema de autenticación de usuarios. Construirás una página de registro para que los usuarios creen cuentas y luego restringirás el acceso a ciertas páginas sólo a los usuarios registrados. A continuación, modificaremos algunas de las funciones de vista para que los usuarios sólo puedan ver sus propios datos. Aprenderás a mantener seguros los datos de tus usuarios.

Permitir a los usuarios introducir datos

Antes de construir un sistema de autenticación para crear cuentas, primero añadiremos algunas páginas que permitan a los usuarios introducir sus propios datos. Daremos a los usuarios la posibilidad de añadir un nuevo tema, añadir una nueva entrada y editar sus entradas anteriores.

Actualmente, sólo un superusuario puede introducir datos a través del sitio de administración. No queremos que los usuarios interactúen con el sitio de administración, así que usaremos las herramientas de creación de formularios de Django para crear páginas que permitan a los usuarios introducir datos.

Añadir nuevos temas

Empecemos dando a los usuarios la posibilidad de añadir un nuevo tema. Añadir una página basada en formularios funciona de forma muy parecida a las páginas que ya hemos construido: definimos una URL, escribimos una función de vista y escribimos una plantilla. La única diferencia importante es la adición de un nuevo módulo llamado *forms.py*, que contendrá los formularios.

El Tema ModelForm

Cualquier página que permita a un usuario introducir y enviar información en una página web es un *formulario*, aunque no lo parezca. Cuando los usuarios introducen información, necesitamos *validar* que la información proporcionada es el tipo correcto de datos y no algo malicioso, como código para interrumpir nuestro servidor. Luego tenemos que procesar y guardar la información válida en el lugar apropiado de la base de datos. Django automatiza gran parte de este trabajo.

La forma más sencilla de construir un formulario en Django es usar un *ModelForm*, que usa la información de los modelos que definimos en el Capítulo 18 para construir automáticamente un formulario. Escribe tu primer formulario en el archivo *forms.py*, que deberías crear en el mismo directorio que *models.py*:

formularios.py

```
from django import formularios

from .models import Tema

❶ class TopicForm(forms.ModelForm):
    class Meta:
❷       modelo = Tema
❸       campos = ['texto']
❹       etiquetas = {'texto': ''}
```

Primero importamos el módulo `forms` y el modelo con el que trabajaremos, `Topic`. En ❶ definimos una clase llamada `TopicForm`, que hereda de `forms.ModelForm`.

La versión más simple de un `ModelForm` consiste en una clase `Meta` anidada que le dice a Django en qué modelo basar el formulario y qué campos

incluir en el formulario. En ❷ construimos un formulario a partir del modelo `Topic` e incluimos sólo el campo de `texto` ❸. El código en ❹ le dice a Django que no genere una etiqueta para el campo de `texto`.

La URL del nuevo tema

La URL para una nueva página debe ser corta y descriptiva, de modo que cuando el usuario quiera añadir un nuevo tema, le enviemos a http://localhost:8000/new_topic/. Este es el patrón de URL para la página `new_topic`, que añadimos a *learning_logs/ urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Página para añadir un nuevo tema
    url(r'^nuevo_tema/$', vistas.nuevo_tema, name='nuevo_tema'),
]
```

Este patrón de URL enviará peticiones a la función de vista `new_topic()`, que escribiremos a continuación.

La función de vista new_topic()

La función `new_topic()` necesita manejar dos situaciones diferentes: las peticiones iniciales para la página `new_topic` (en cuyo caso debería mostrar un formulario en blanco) y el procesamiento de cualquier dato enviado en el formulario. A continuación, debe redirigir al usuario a la página de `temas`:

vistas.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse

from .models import Topic
from .forms import TopicForm

--snip--
def nuevo_tema(solicitud):
    """Añade un nuevo
    tema."""
❶   si request.method != 'POST':
        # No hay datos enviados; crear un formulario en blanco.
❷       form = TopicForm()
    else:
        # POST datos enviados; procesar datos.
❸       form = TopicForm(request.POST)
❹       si form.is_valid():
❺           form.save()
❻           return HttpResponseRedirect(reverse('learning_logs:topics'))

❼   contexto = {'formulario': formulario}
    return render(request, 'learning_logs/new_topic.html', context)
```

Importamos la clase `HttpResponseRedirect`, que usaremos para redirigir al lector de vuelta a la página de `temas` después de que envíe su tema. La función `reverse()` determina la URL a partir de un patrón de URL con nombre, lo que significa que Django generará la URL cuando se solicite la página. También importamos el formulario que acabamos de escribir,

`TopicForm`.

Solicitudes GET y POST

Los dos tipos principales de peticiones que utilizarás cuando construyas aplicaciones web son las peticiones GET y las peticiones POST. Las peticiones *GET* se utilizan para páginas que sólo leen datos del servidor. Usualmente usas peticiones *POST* cuando el usuario necesita enviar información a través de un formulario. Especificaremos el método POST para procesar todos nuestros formularios. (Existen algunos otros tipos de peticiones, pero no los usaremos en este proyecto).

La función `new_topic()` toma como parámetro el objeto request. Cuando el usuario solicita inicialmente esta página, su navegador enviará una petición GET. Cuando el usuario haya rellenado y enviado el formulario, su navegador enviará una petición POST. Dependiendo de la petición, sabremos si el usuario está solicitando un formulario en blanco (una petición GET) o pidiéndonos que procesemos un formulario rellenado (una petición POST).

La prueba de ❶ determina si el método de petición es GET o POST. Si el método de solicitud no es POST, la solicitud es probablemente GET, por lo que necesitamos devolver un formulario en blanco (si es otro tipo de solicitud, sigue siendo seguro devolver un formulario en blanco).

Hacemos una instancia de `TopicForm` ❷, la almacenamos en el `formulario` vari- able, y enviamos el formulario a la plantilla en el diccionario de contexto ❼. Como no incluimos argumentos al instanciar `TopicForm`, Django crea un formulario en blanco que el usuario puede rellenar.

Si el método de solicitud es POST, el bloque `else` se ejecuta y procesa los datos enviados en el formulario. Creamos una instancia de `TopicForm` ❸ y le pasamos los datos introducidos por el usuario, almacenados en `request.POST`. El objeto `formulario` que se devuelve contiene la información enviada por el usuario.

No podemos guardar la información enviada en la base de datos hasta que hayamos comprobado que es válida ❹. La función `is_valid()` comprueba que se han rellenado todos los campos obligatorios (todos los campos de un formulario son obligatorios por defecto) y que los datos introducidos coinciden con los tipos de campo esperados -por ejemplo, que la longitud del `texto` es inferior a 200 caracteres, como especificamos en *models.py* en el Capítulo 18. Esta validación automática nos ahorra mucho trabajo. Si todo es válido, podemos llamar a `save()` ❺, que escribe los datos del formulario
a la base de datos. Una vez guardados los datos, podemos abandonar esta página. Nos
utilizar `reverse()` para obtener la URL de la `página de temas` y pasar la URL a `HttpResponseRedirect()` ❻, que redirige el navegador del usuario a la página de `temas`. En la `página de temas`, el usuario debería ver el tema que acaba de introducir en la lista de temas.

Plantilla new_topic

Ahora creamos una nueva plantilla llamada *new_topic.html* para mostrar el formulario que acabamos de crear:

nuevo_tema.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}
  <p>Añadir un nuevo tema:</p>
```

```
❶ <form action="{% url 'learning_logs:new_topic' %}" method='post'>
❷   {% csrf_token %}
❸   {{ form.as_p }}
❹   <button name="submit">añadir tema</button>
  </form>

{% endblock content %}
```

Esta plantilla extiende *base.html*, por lo que tiene la misma estructura base que el resto de páginas de Learning Log. En ❶ definimos un formulario HTML. El argumento `action` indica al servidor dónde enviar los datos enviados en el formulario; en este caso, los enviamos de vuelta a la función de la vista `new_topic()`. El argumento `method` le dice al navegador que envíe los datos como una petición POST.

Django utiliza la etiqueta de plantilla `{% csrf_token %}` ❷ para evitar que los atacantes utilicen el formulario para obtener acceso no autorizado al servidor (este tipo de ataque se denomina *falsificación de solicitud de sitio cruzado*). En ❸ mostramos el formulario; aquí se ve lo sencillo que Django puede hacer tareas como mostrar un formulario. Sólo necesitamos incluir la variable de plantilla `{{ form.as_p }}` para que Django cree todos los campos necesarios para mostrar el formulario automáticamente. El modificador `as_p` indica a Django que renderice todos los elementos del formulario en formato párrafo, que es una forma sencilla de mostrar el formulario de forma ordenada.

Django no crea un botón de envío para los formularios, así que definimos uno en ❹.

Enlace a la página new_topic

A continuación, incluimos un enlace a la página `new_topic` en la página de `temas`:

temas.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}

  <p>Temas</p>

  <ul>
    --snip--
  </ul>

  <a href="{% url 'learning_logs:new_topic' %}">Añade un nuevo tema:</a>

{% endblock content %}
```

Coloque el enlace después de la lista de temas existentes. La Figura 19-1 muestra el formulario resultante. Siga adelante y utilice el formulario para añadir algunos temas nuevos.



Figura 19-1: Página para añadir un nuevo tema

Añadir nuevas entradas

Ahora que el usuario puede añadir un nuevo tema, también querrá añadir nuevas entradas. Definiremos de nuevo una URL, escribiremos una función de vista y una plantilla, y enlazaremos a la página. Pero primero añadiremos otra clase a *forms.py*.

Modelo de formulario de entrada

Necesitamos crear un formulario asociado al modelo `Entry`, pero esta vez con un poco más de personalización que `TopicForm`:

formularios.py

```
from django import formularios

from .models import Tema, Entrada

class TemaForm(forms.ModelForm):
    --snip--

clase EntryForm(forms.ModelForm):
    clase Meta:
        modelo = Entrada
        campos = ['texto']
        etiquetas = {'texto': ''}
        widgets = {'text': forms.Textarea(attrs={'cols': 80})}
```

Primero actualizamos la sentencia `import` para incluir `Entry` además de `Topic`. La nueva clase `EntryForm` hereda de `forms.ModelForm` y tiene una Metaclase anidada que lista el modelo en el que se basa y el campo a incluir en el formulario. De nuevo damos al campo `'text'` una etiqueta en blanco ❶.

En ❷ incluimos el atributo `widgets`. Un *widget* es un elemento de formulario HTML, como un cuadro de texto de una línea, un área de texto de varias líneas o una lista desplegable. Al incluir el atributo `widgets` puedes anular las opciones predeterminadas de widgets de Django. Al decirle a Django que utilice un elemento `forms.Textarea`, estamos personalizando el widget de entrada

para el campo `'text' de` modo que el área de texto tendrá 80 columnas de ancho en lugar de las 40 predeterminadas. Esto dará a los usuarios suficiente espacio para escribir una entrada significativa.

La URL de new_entry

Necesitamos incluir un argumento `topic_id` en la URL para añadir una nueva entrada, porque la entrada debe estar asociada a un tema concreto. Aquí está la URL, que añadimos a *learning_logs/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Página para añadir una nueva entrada
    url(r'^new_entry/(?P<topic_id>\d+)/$', views.new_entry, name='new_entry'),
]
```

Este patrón de URL coincide con cualquier URL de la forma `http://localhost:8000/new_entry/id/`, donde *id* es un número que coincide con el ID del tema. El código `(?P<topic_id>\d+)` captura un valor numérico y lo almacena en la variable `topic_id`. Cuando se solicita una URL que coincide con este patrón, Django envía la solicitud y el ID del tema a la función de vista `new_entry()`.

La función de vista new_entry()

La función de vista para `new_entry` es muy parecida a la función para añadir un nuevo tema:

vistas.py

```
from django.shortcuts import render
--snip--

from .models import Tema
from .forms import TopicForm, EntryForm

--snip--
def nueva_entrada(solicitud, tema_id):
    """Añadir una nueva entrada para un tema concreto."""
❶   topic = Topic.objects.get(id=topic_id)

❷   si request.method != 'POST':
        # No hay datos enviados; crear un formulario en blanco.
❸       form = EntryForm()
    else:
        # POST datos enviados; procesar datos.
❹       form = EntryForm(data=request.POST)
        if form.is_valid():
❺           nueva_entrada = form.save(commit=False)
❻           nueva_entrada.tema =
            tema
            nueva_entrada.guardar()
❼           return HttpResponseRedirect(reverse('learning_logs:topic',
                                        args=[topic_id]))

    context = {'tema': tema, 'formulario': formulario}
    return render(request, 'learning_logs/new_entry.html', context)
```

Actualizamos la sentencia `import` para incluir el `EntryForm` que acabamos de hacer. La definición de `new_entry()` tiene un parámetro `topic_id` para almacenar el valor que recibe de la URL. Necesitaremos el topic para renderizar la página y procesar los datos del formulario, así que usamos `topic_id` para obtener el objeto topic correcto en ❶.

En ❷ comprobamos si el método de petición es POST o GET. El bloque `if` se ejecuta si es una petición GET, y creamos una instancia vacía de `EntryForm` ❸. Si el método de petición es `POST`, procesamos los datos creando una `instancia` de `EntryForm`, rellenada con los datos POST del objeto de `petición` ❹. A continuación, comprobamos si el formulario es válido. Si lo es, necesitamos establecer el atributo `topic del objeto entry` antes de guardarlo en la base de datos.

Cuando llamamos a save(), incluimos el argumento `commit=False` ❺ para decirle a Django que cree un nuevo objeto de entrada y lo almacene en `new_entry` sin guardarlo todavía en la base de datos. Establecemos el atributo `topic` de `new_entry` al topic que sacamos de la base de datos al principio de la función ❻, y luego llamamos a `save()` sin argumentos. Esto guarda la entrada en la base de datos con el tema asociado correcto.

En ❼ redirigimos al usuario a la página del tema. La llamada a `reverse()` requiere dos argumentos: el nombre del patrón de URL para el que queremos generar una URL y una lista `args` que contenga cualquier argumento que deba incluirse en la URL. La lista `args` contiene un elemento, `topic_id`. La llamada a `HttpResponseRedirect()` redirige al usuario a la página del tema para el que hizo una entrada, y debería ver su nueva entrada en la lista de entradas.

Plantilla new_entry

Como puedes ver en el siguiente código, la plantilla para `new_entry` es similar a la plantilla para `new_topic`:

nueva_entrada.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}

❶ <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Añade una nueva entrada:</p>
❷ <form action="{% url 'learning_logs:new_entry' topic.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
    <button name='submit'>añadir entrada</button>
  </form>

{% endblock content %}
```

Mostramos el tema en la parte superior de la página ❶, para que el usuario pueda ver a qué tema está añadiendo una entrada. También sirve

de enlace a la página principal del tema.

El argumento de `la acción del` formulario incluye el valor `topic_id` en la URL, para que la función de vista pueda asociar la nueva entrada con el tema correcto ❷. Aparte de eso, esta plantilla tiene el mismo aspecto que *new_topic.html*.

Enlace a la página new_entry

A continuación, tenemos que incluir un enlace a la página `new_entry` desde cada página de tema:

tema.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}

  <p>Tema: {{ tema }}</p>

  <p>Entradas:</p>
  <p>
    <a href="{% url 'learning_logs:new_entry' topic.id %}">añadir nueva
    entrada</a>
  </p>
  <ul>
  --snip--
  </ul>

{% endblock content %}
```

Añadimos el enlace justo antes de mostrar las entradas, porque añadir una nueva entrada será la acción más común en esta página. La Figura 19-2 muestra la página `new_entry`. Ahora los usuarios pueden añadir nuevos temas y tantas entradas como quieran para cada tema. Prueba la página `new_entry` añadiendo algunas entradas a algunos de los temas que has creado.



Figura 19-2: Página new_entry

Edición de entradas

Ahora crearemos una página que permita a los usuarios editar las entradas que ya han añadido.

La URL de edit_entry

La URL de la página debe contener el ID de la entrada que se va a editar. Es la siguiente
learning_logs/urls.py:

urls.py

```
--snip--
urlpatterns = [
    --snip--
    # Página para editar una entrada
    url(r'^edit_entry/(?P<entry_id>\d+)/$', views.edit_entry,
        name='editar_entrada'),
]
```

El ID pasado en la URL (por ejemplo, *http://localhost:8000/edit_entry/1/)* se almacena en el parámetro `entry_id`. El patrón de URL envía las solicitudes que coinciden con este formato a la función de la vista `edit_entry()`.

La función de vista edit_entry()

Cuando la página `edit_entry` recibe una petición GET, `edit_entry()` devolverá un formulario para editar la entrada. Cuando la página reciba una petición POST con el texto de la entrada revisado, guardará el texto modificado en la base de datos:

vistas.py

```
from django.shortcuts import render
--snip--

from .models import Tema, Entrada
from .forms import TopicForm, EntryForm
--snip--

def editar_entrada(solicitud,
    entrada_id): """Editar una
    entrada existente."""
❶   entrada =
    Entry.objects.get(id=entry_id) tema =
    entrada.tema

    si request.method != 'POST':
        # Solicitud inicial; rellenar el formulario con la entrada actual.
❷       form = EntryForm(instance=entry)
    else:
        # POST datos enviados; procesar datos.
❸       form = EntryForm(instance=entry, data=request.POST)
        if form.is_valid():
❹           form.guardar()
❺           return HttpResponseRedirect(reverse('learning_logs:topic',
                                    args=[topic.id]))

    context = {'entrada': entrada, 'tema': tema, 'formulario': formulario}
    return render(request, 'learning_logs/edit_entry.html', context)
```

Primero necesitamos importar el modelo de `Entrada`. En ❶ obtenemos el objeto entrada que el usuario quiere editar y el tema asociado a esta entrada. En el bloque `if`, que se ejecuta para una petición GET, creamos una instancia de `EntryForm` con

el argumento `instance=entry` ❷. Este argumento le dice a Django que cree el formulario pre-rellenado con la información del objeto de entrada existente. El usuario verá sus datos existentes y podrá editarlos.

Cuando procesamos una petición POST, pasamos el argumento `instance=entry` y el argumento `data=request.POST` ❸ para decirle a Django que cree un formulario
basado en la información asociada con el objeto de entrada existente, actualizado con cualquier dato relevante de `request.POST`. A continuación comprobamos si el formulario es válido; si lo es, llamamos a `save()` sin argumentos ❹. Luego redirigimos a la página del `tema` ❺, donde el usuario debería ver la versión actualizada de la entrada que editó.

La plantilla edit_entry

Aquí está *edit_entry.html*, que es similar a *new_entry.html*:

editar_entrada.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}

  <p><a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a></p>

  <p>Editar entrada:</p>

❶ <form action="{% url 'learning_logs:edit_entry' entry.id %}" method='post'>
    {% csrf_token %}
    {{ form.as_p }}
❷   <button name="submit">guardar cambios</button>
  </form>

{% endblock content %}
```

En ❶ el argumento `action` devuelve el formulario a la función `edit_entry()` para su procesamiento. Incluimos el ID de la entrada como argumento en la etiqueta `{% url %}`, para que la función view pueda modificar el objeto de entrada correcto. Etiquetamos el botón de envío como *guardar cambios* para recordar al usuario que están guardando ediciones, no creando una nueva entrada ❷.

Enlace a la página edit_entry

Ahora tenemos que incluir un enlace a la página `edit_entry` para cada entrada en la página del tema:

topic.html

```
--snip--
  {% for entrada in entradas %}
    <li>
      <p>{{ entry.date_added|date:'M d, Y H:i' }}</p>
      <p>{{entrada.texto|saltos de línea }}</p>
      <p>
```

```
            <a href="{% url 'learning_logs:edit_entry' entry.id %}">editar
            entrada</a>
          </p>
</li> <li>
 --snip--
```

Incluimos el enlace de edición después de mostrar la fecha y el texto de cada entrada. Utilizamos la etiqueta de plantilla `{% url %}` para determinar la URL del patrón de URL con nombre `edit_entry`, junto con el atributo ID de la entrada actual en el bucle (`entry.id`). El texto del enlace `"editar entrada"` aparece después de cada entrada de la página. La Figura 19-3 muestra el aspecto de la página de tema con estos enlaces.

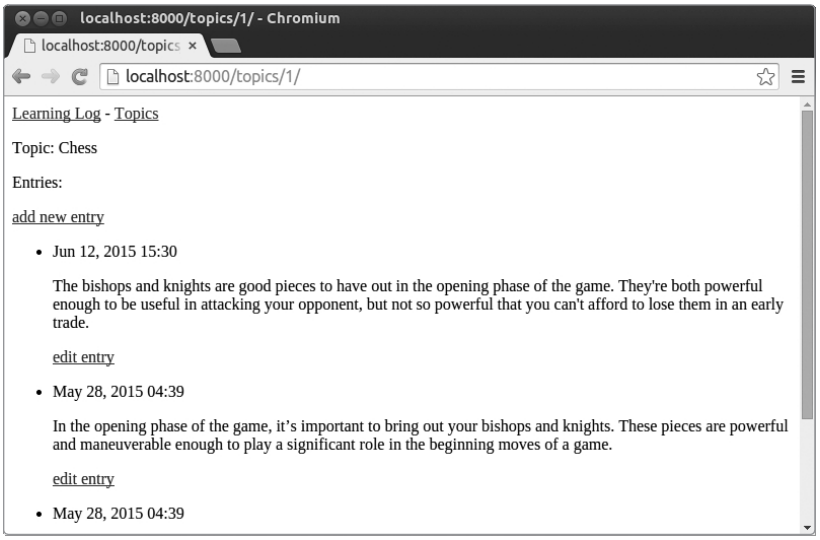


Figura 19-3: Cada `entrada` tiene ahora un enlace para editarla.

Learning Log dispone ahora de la mayor parte de las funciones que necesita. Los usuarios pueden añadir temas y entradas, y leer cualquier conjunto de entradas que deseen. En la próxima sección, implementaremos un sistema de registro de usuarios para que cualquiera pueda crear una cuenta en Learning Log y crear su propio conjunto de temas y entradas.

TRY IT yOURSELF

19-1. Blog: Inicia un nuevo proyecto Django llamado *Blog*. Crea una app llamada *blogs* en el proyecto, con un modelo llamado `BlogPost`. El modelo debe tener campos como `title`, `text` y `date_added`. Crear un superusuario para el proyecto, y utilizar el sitio de administración para hacer un par de mensajes cortos. Haz una página de inicio que muestre todos los posts en orden cronológico.

Cree un formulario para crear nuevas entradas y otro para editar las existentes.

Rellene sus formularios para asegurarse de que funcionan.

Configuración de cuentas de usuario

En esta sección configuraremos un sistema de registro y autorización de usuarios para permitir que las personas registren una cuenta e inicien y cierren sesión. Crearemos una nueva aplicación para contener toda la funcionalidad relacionada con el trabajo con usuarios. También modificaremos ligeramente el modelo de temas para que cada tema pertenezca a un usuario determinado.

La App de los usuarios

Empezaremos creando una nueva aplicación llamada `users`, usando el comando `startapp`:

```
(ll_env)learning_log$ python manage.py startapp users
(ll_env)learning_log$ ls
❶ db.sqlite3 learning_log learning_logs ll_env manage.py users
(ll_env)learning_log$ ls users
❷ admin.py init .py migrations models.py tests.py views.py
```

Este comando crea un nuevo directorio llamado *users* ❶ con una estructura idéntica a la app `learning_logs` ❷.

Añadir usuarios a settings.py

Tenemos que añadir nuestra nueva aplicación a `INSTALLED_APPS` en *settings.py*, así:

settings.py

```
--snip--
APLICACIONES_INSTALADAS = (
    --snip--
    # Mis
    aplicacio
    nes
    'learning_logs',
    'users',
)
--snip--
```

Ahora Django incluirá la aplicación `del usuario` en el proyecto global.

Incluir las URL de los usuarios

A continuación, tenemos que modificar la raíz *urls.py* para que incluya las direcciones URL que vamos a escribir para la aplicación de `los usuarios`:

urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^users/', include('users.urls', namespace='users')),
```

```
    url(r'', include('learning_logs.urls', namespace='learning_logs')),
]
```

Añadimos una línea para incluir el archivo *urls.py* de `users`. Esta línea coincidirá con cualquier URL que comience con la palabra *users*, como *http://localhost:8000/users/*

login/. También creamos el espacio de nombres `'users'` para poder distinguir las URL que pertenecen a la aplicación `learning_logs` de las URL que pertenecen a la aplicación `users`.

Página de inicio de sesión

Primero implementaremos una página de login. Utilizaremos la vista de `inicio de sesión` predeterminada que proporciona Django, por lo que el patrón de URL es un poco diferente. Crea un nuevo archivo *urls.py* en el directorio *learning_log/users/*, y añádele lo siguiente:

urls.py

```
"""Define patrones de URL para

usuarios""" from django.conf.urls

import url
❶ from django.contrib.auth.views import login

from . import views

urlpatterns = [
    # Página de inicio de sesión
❷   url(r'^login/$', login, {'template_name': 'users/login.html'},
        name='login'),
]
```

Primero importamos la vista de `inicio de sesión` por defecto ❶. El patrón de la página de inicio de sesión coincide con la URL *http://localhost:8000/users/login/* ❷. Cuando Django lee esta URL, la palabra *users* le dice a Django que busque en *users/urls.py*, y *login* le dice que envíe peticiones a la vista de `login por defecto` de Django (fíjate que el argumento view es `login`, no `views.login`). Como no estamos escribiendo nuestra propia función de vista, le pasamos un diccionario que le dice a Django dónde encontrar la plantilla que vamos a escribir. Esta plantilla será parte de la aplicación `users`, no de la aplicación `learning_logs`.

Plantilla de inicio de sesión

Cuando el usuario solicita la página de inicio de sesión, Django utilizará su vista de `inicio de sesión` por defecto, pero todavía tenemos que proporcionar una plantilla para la página. Dentro del directorio *learning_log/users/,* crea un directorio llamado *templates*; dentro de éste, crea otro directorio llamado *users*. Aquí está la plantilla *login.html*, que debes guardar en *learning_log/users/templates/users/*:

login.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% block content %}

❹   <button name="submit">iniciar
    sesión</button>
```

```
❶ {% if form.errors %}
  <p>Tu nombre de usuario y contraseña no coinciden. Por favor, inténtelo de
  nuevo.</p> <p>
  {% endif %}

❷ <form method="post" action="{% url 'users:login' %}">
  {% csrf_token %}
❸ {{ form.as_p }}
```

```
❺  <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
   </form>

{% endblock content %}
```

Esta plantilla amplía *base.html* para garantizar que la página de inicio de sesión tenga el mismo aspecto que el resto del sitio. Ten en cuenta que una plantilla de una aplicación puede extender una plantilla de otra aplicación.

Si se establece el atributo `errors` del formulario, mostramos un mensaje de error ❶, informando de que la combinación de nombre de usuario y contraseña no coincide con nada almacenado en la base de datos.

Queremos que la vista de inicio de sesión procese el formulario, así que establecemos el argumento de `acción` como la URL de la página de inicio de sesión ❷. La vista de inicio de sesión envía un formulario a la plantilla, y depende de nosotros mostrar el formulario ❸ y añadir un botón de envío ❹. En ❺ incluimos un elemento de formulario oculto, `'next'`; el argumento `value` le dice a Django a dónde redirigir al usuario después de que haya iniciado sesión con éxito. En este caso, enviamos al usuario de vuelta a la página de inicio.

Enlace a la página de inicio de sesión

Vamos a añadir el enlace de inicio de sesión a *base.html* para que aparezca en todas las páginas. No queremos que el enlace aparezca cuando el usuario ya ha iniciado sesión, así que lo anidamos dentro de una etiqueta `{% if %}`:

base.html

```
<p>
  <a href="{% url 'learning_logs:index' %}">Registro de aprendizaje</a> -
  <a href="{% url 'learning_logs:topics' %}">Temas</a> -
❶ {% if user.is_authenticated %}
❷   Hola, {{ user.username }}.
  {% else %}
❸   <a href="{% url 'users:login' %}">conectarse</a>
  {% endif %}
</p>

{% block content %}{% endblock content %}
```

En el sistema de autenticación de Django, cada plantilla tiene una variable de `usuario` disponible, que siempre tiene un atributo `is_authenticated` establecido: el atributo es `True` si el usuario está conectado y `False` si no lo está. Esto permite mostrar un mensaje a los usuarios autenticados y otro a los no autenticados.

Aquí mostramos un saludo a los usuarios actualmente conectados ❶. Los usuarios autenticados tienen un atributo de `nombre de usuario` adicional, que utilizamos para personalizar el saludo y recordar al usuario que ha iniciado

```
❹ <button name="submit">iniciar
  sesión</button>
```

sesión ❷. En ❸ mostramos un enlace a la página de inicio de sesión para los usuarios que no se han a u t e n t i c a d o .

Uso de la página de inicio de sesión

Ya hemos creado una cuenta de usuario, así que vamos a iniciar sesión para ver si la página funciona. Vaya a *http://localhost:8000/admin/*. Si aún estás conectado como administrador, busca un enlace de cierre de sesión en la cabecera y haz clic en él.

Cuando haya cerrado la sesión, vaya a http://localhost:8000/users/login/. Deberías ver una página de inicio de sesión similar a la que se muestra en la Figura 19-4. Introduce el nombre de usuario y la contraseña que configuraste anteriormente, y deberías volver a la página de índice. La cabecera de la página de inicio debería mostrar un saludo personalizado con tu nombre de usuario.

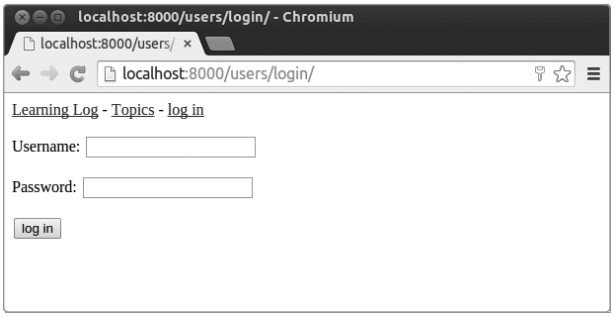


Figura 19-4: Página de inicio de sesión

Cerrar sesión

Ahora tenemos que proporcionar a los usuarios una forma de cerrar la sesión. No vamos a construir una página para cerrar la sesión; los usuarios simplemente harán clic en un enlace y serán enviados de vuelta a la página de inicio.
Definiremos un patrón de URL para el enlace de cierre de sesión, escribiremos una función de vista y proporcionaremos un enlace de cierre de sesión en *base.html*.

La URL de cierre de sesión

El siguiente código define el patrón de URL para cerrar la sesión, que coincide con la URL http://localhost:8000/users/logout/. Aquí está *users/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    # Página de inicio de sesión
    --snip--
    # Página de cierre de sesión
    url(r'^logout/$', views.logout_view, name='logout'),
]
```

El patrón URL envía la petición a la función `logout_view()`, que se llama así para distinguirla de la función `logout()` que llamaremos desde dentro de la vista. (Asegúrate de que estás modificando *users/urls.py*, no *learning_log/urls.py*).

Función de la vista logout_view()

La función `logout_view()` es sencilla: simplemente importamos la función `logout()` de Django, la llamamos, y luego redirigimos de nuevo a la página de inicio. Abre *users/views.py* e introduce el siguiente código.

vistas.py

```
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
❶ from django.contrib.auth import logout

def logout_view(request):
    """Cerrar sesión del
    usuario."""
❷   cerrar sesión(solicitud)
❸   return HttpResponseRedirect(reverse('learning_logs:index'))
```

Importamos la función `logout()` de `django.contrib.auth` ❶. En la función, llamamos a `logout()` ❷, que requiere el objeto `request` como argumento. Luego redirigimos a la página de inicio ❸.

Enlace a la vista de cierre de sesión

Ahora necesitamos un enlace de cierre de sesión. Lo incluiremos como parte de *base.html para que esté* disponible en todas las páginas y lo incluiremos en la porción `{% if user.is_authenticated %}` para que sólo los usuarios que ya han iniciado sesión puedan v e r l o :

base.html

```
--snip-
  {% if user.is_authenticated %}
    Hola, {{ user.username }}.
    <a href="{% url 'usuarios:cerrar sesión' %}">cerrar sesión</a>
  {% else %}
    <a href="{% url 'users:login' %}">conectarse</a>
  {% endif %}
--snip--
```

La Figura 19-5 muestra la página de inicio actual tal y como aparece a un usuario conectado. El estilo es mínimo porque nos estamos centrando en construir un sitio que funcione correctamente. Cuando el conjunto de características requeridas funcione, le daremos al sitio un aspecto más profesional.

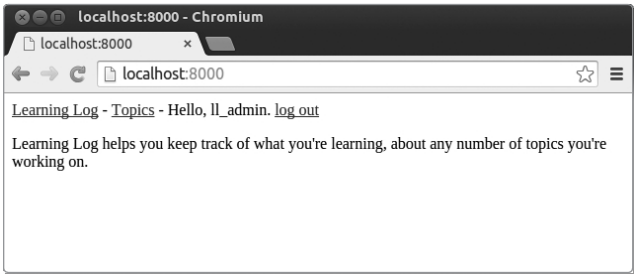


Figura 19-5: Página de inicio con un saludo personalizado y un enlace de cierre de sesión

Página de registro

A continuación, vamos a construir una página para permitir que los nuevos usuarios se registren. Usaremos el `UserCreationForm` por defecto de Django pero escribiremos nuestra propia función de vista y plantilla.

La URL de registro

El siguiente código proporciona el patrón de URL para la página de registro, de nuevo en *users/urls.py*:

urls.py

```
--snip--
urlpatterns = [
    # Página de inicio de sesión
    --snip--
    # Página de inscripción
    url(r'^register/$', views.register, name='register'),
]
```

Este patrón coincide con la URL *http://localhost:8000/users/register/* y envía peticiones a la función `register()` que vamos a escribir.

Función de visualización register()

La función de la vista `register()` debe mostrar un formulario de registro en blanco cuando se solicite por primera vez la página de registro y, a continuación, procesar los formularios de registro completados cuando se envíen. Cuando el registro se realiza correctamente, la función también debe iniciar la sesión del nuevo usuario. Añade el siguiente código a *users/views.py*:

vistas.py

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.contrib.auth import login, logout, authenticate
from django.contrib.auth.forms import UserCreationForm

def logout_view(request):
--snip--

def registrar(solicitud):
    """Registrar un nuevo
    usuario.""" if
    request.method != 'POST':
        # Mostrar formulario de registro en blanco.
1       form = UserCreationForm()
    else:
        # Tramitar formulario cumplimentado.
2       form = UserCreationForm(data=request.POST)

3       si form.is_valid():
4           nuevo_usuario = form.guardar()
            # Registrar al usuario y luego redirigirlo a la página de inicio.
5           usuario_autenticado =
                authenticate(nombre_usuario=nuevo_usuario.nombre_usuario
                , contraseña=petición.POST['contraseña1'])
6           login(solicitud, usuario_autenticado)
7           return HttpResponseRedirect(reverse('learning_logs:index'))

    context = {'formulario': formulario}
```

```
    return render(request, 'usuarios/registro.html', context)
```

Primero importamos la función `render()`. Luego importamos las funciones `login()` y `authenticate()` para iniciar la sesión del usuario si su información de registro es correcta. También importamos el `UserCreationForm` por defecto. En la función `register()`, comprobamos si estamos respondiendo a una petición POST. Si no es así, creamos una instancia de `UserCreationForm` sin datos iniciales ❶.

Si estamos respondiendo a una petición POST, creamos una instancia de `UserCreationForm` basada en los datos enviados ❷. Comprobamos que los datos son válidos ❸-en este caso, que el nombre de usuario tiene los caracteres apropiados, las contraseñas coinciden, y el usuario no está tratando de hacer nada malicioso en su presentación.

Si los datos enviados son válidos, llamamos al método save() del `formulario` para guardar el nombre de usuario y el hash de la contraseña en la base de datos ❹. El método `save()` devuelve el objeto usuario recién creado, que almacenamos en `new_user`.

Cuando se guarda la información del usuario, lo registramos, que es un proceso de dos pasos: llamamos a `authenticate()` con los argumentos `new_user.username` y su contraseña ❺. Cuando se registran, se le pide al usuario que introduzca dos contraseñas que coincidan, y como el formulario es válido, sabemos que las contraseñas coinciden, así que podemos usar cualquiera de las dos. Aquí obtenemos el valor asociado a la clave `'password1'` en los datos POST del formulario. Si el nombre de usuario y la contraseña son correctos, el método devuelve un objeto usuario autenticado, que almacenamos en `authenticated_user`. A continuación, llamamos a la función `login()` con los objetos `request` y `authenticated_user` ❻, que crea una sesión válida para el nuevo usuario. Por último, redirigimos al usuario a la página de inicio ❼, donde un saludo personalizado en la cabecera le informa de que su registro se ha realizado correctamente.

La plantilla de registro

La plantilla de la página de registro es similar a la de la página de inicio de sesión. Asegúrese de guardarla en el mismo directorio que *login.html*:

registro.html

```
{% extends "registros_de_aprendizaje/base.html"
                %}

{% block content %}

<form method="post" action="{% url 'usuarios:registro' %}">
    {% csrf_token %}
    {{ form.as_p }}

    <button name="submit">registrar</button>
    <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
  </form>
```

```
{% endblock content %}
```

Volvemos a utilizar el método `as_p` para que Django muestre todos los campos del formulario adecuadamente, incluyendo cualquier mensaje de error si el formulario no se rellena correctamente.

Enlace a la página de inscripción

A continuación, añadiremos el código para mostrar el enlace de la página de registro a cualquier usuario que no haya iniciado sesión:

base.html

```
--snip--
  {% if user.is_authenticated %}
    Hola, {{ user.username }}.
    <a href="{% url 'usuarios:cerrar sesión' %}">cerrar sesión</a>
  {% else %}
    <a href="{% url 'usuarios:registrarse' %}">registrarse</a> -
    <a href="{% url 'users:login' %}">conectarse</a>
  {% endif %}
--snip--
```

Ahora los usuarios que han iniciado sesión ven un saludo personalizado y un enlace de cierre de sesión. Los usuarios que no han iniciado sesión ven un enlace de página de registro y un enlace de inicio de sesión. Pruebe la página de registro creando varias cuentas de usuario con distintos nombres de usuario.

En la siguiente sección, restringiremos algunas de las páginas para que sólo estén disponibles para usuarios registrados, y nos aseguraremos de que cada tema pertenezca a un usuario específico.

NOTE *El sistema de registro que hemos creado permite a cualquier persona crear un número ilimitado de cuentas para Learning Log. Pero algunos sistemas requieren que los usuarios confirmen su identidad enviando un correo electrónico de confirmación al que el usuario debe responder. De este modo, el sistema genera menos cuentas de spam que el sistema sencillo que utilizamos aquí. Sin embargo, cuando estás aprendiendo a construir aplicaciones, es perfectamente apropiado practicar con un sistema simple de registro de usuarios como el que estamos usando.*

TRY IT yOURSELF

19-2. Cuentas de Blog: Agregue un sistema de autenticación y registro de usuarios al proyecto Blog que inició en el Ejercicio 19-1 (página 438). Asegúrese de que los usuarios registrados vean su nombre de usuario en algún lugar de la pantalla y que los usuarios no registrados vean un enlace a la página de registro.

Permitir que los usuarios sean propietarios de sus datos

Los usuarios deben poder introducir datos exclusivos para ellos, así que crearemos un sistema para averiguar qué datos pertenecen a cada usuario, y luego restringiremos el acceso a determinadas páginas para que los usuarios puedan trabajar sólo con sus propios datos.

En esta sección, modificaremos el modelo de `temas` para que cada tema pertenezca a un usuario específico. Esto también se ocupará de las entradas, porque cada entrada pertenece a un tema específico. Comenzaremos restringiendo el acceso a ciertas páginas.

Restringir el acceso con @login_required

Django hace que sea fácil restringir el acceso a ciertas páginas a los usuarios registrados a través del decorador `@login_required`. Un *decorador* es una directiva que se coloca justo antes de la definición de una función y que Python aplica a la función antes de que se ejecute para alterar cómo se comporta el código de la función. Veamos un ejemplo.

Restringir el acceso a la página de temas

Cada tema será propiedad de un usuario, por lo que sólo los usuarios registrados podrán solicitar la página de temas. Añade el siguiente código a *learning_logs/views.py*:

views.py

```
--snip--
from django.core.urlresolvers import reverse
from django.contrib.auth.decorators import login_required

from .models import Tema, Entrada
--snip--

@login_required
def temas(solicitud):
    """Mostrar todos los
    temas."""
    --snip--
```

Primero importamos la función login_required(). Aplicamos login_required() como un decorador a la función de vista `topics()` anteponiendo `login_required` con el símbolo `@` para que Python sepa que debe ejecutar el código en `login_required()` antes del código en `topics()`.

El código en `login_required()` comprueba si un usuario está conectado, y Django ejecutará el código en `topics()` sólo si lo está. Si el usuario no ha iniciado sesión, se le redirige a la página de inicio de sesión.

Para que esta redirección funcione, necesitamos modificar settings.*py* para que Django sepa dónde encontrar la página de inicio de sesión. Añade lo siguiente al final de settings.*py*:

settings.py

```
"""
Configuración de Django para el proyecto learning_log
--snip--

# Mi configuración
LOGIN_URL = '/usuarios/acceso/'
```

Ahora cuando un usuario no autenticado solicite una página protegida por el decorador `@login_required`, Django enviará al usuario a la URL definida por `LOGIN_URL` en *settings.py*.

Puede probar esta configuración cerrando la sesión de cualquier cuenta de usuario y accediendo a la página de inicio. A continuación, haga clic en el enlace Temas, que le redirigirá a la página de inicio de sesión. A continuación, inicie sesión en cualquiera de sus cuentas y, desde la

página de inicio, vuelva a hacer clic en el enlace Temas. Debería poder acceder a la página de temas.

Restricción del acceso a través de Learning Log

Django hace que sea fácil restringir el acceso a las páginas, pero tienes que decidir qué páginas proteger. Es mejor pensar primero qué páginas necesitan no estar restringidas y luego restringir todas las demás páginas del proyecto. Puedes corregir fácilmente el exceso de restricción de acceso, y es menos peligroso que dejar las páginas sensibles sin restricciones.

En Learning Log, mantendremos la página de inicio, la página de registro y el cierre de sesión sin restricciones. Restringiremos el acceso a todas las demás páginas.

Aquí está *learning_logs/views.py* con los decoradores `@login_required` aplicados a todas las vistas excepto a `index()`:

views.py

```
--snip--
@login_required
def temas(solicitud):
    --snip--

@login_required
def tema(solicitud, topic_id):
    --snip--

@login_required
def nuevo_tema(solicitud):
    --snip--

@login_required
def nueva_entrada(solicitud, tema_id):
    --snip--

@login_required
def editar_entrada(solicitud, entrada_id):
    --snip--
```

Intenta acceder a cada una de estas páginas mientras estás desconectado: serás redirigido a la página de inicio de sesión. Tampoco podrás hacer clic en enlaces a páginas como `nuevo_tema`. Pero si introduces la URL http://localhost:8000/new_topic/, serás redirigido a la página de inicio de sesión. Deberías restringir el acceso a cualquier URL que sea de acceso público y esté relacionada con datos privados del usuario.

Conexión de datos a determinados usuarios

Ahora tenemos que conectar los datos enviados al usuario que los envió. Tenemos que conectar sólo los datos más altos en la jerarquía a un usuario, y los datos de nivel inferior seguirán. Por ejemplo, en Learning Log, los temas son el nivel más alto de datos en la aplicación, y todas las entradas están conectadas a un tema. Mientras cada tema pertenezca a un usuario específico, podremos rastrear la propiedad de cada entrada en la base de datos.

Modificaremos el modelo de `temas` añadiendo una relación de clave

externa con un usuario. A continuación, tendremos que migrar la base de datos. Por último, tendremos que modificar algunas de las vistas para que sólo muestren los datos asociados al usuario conectado en ese momento.

Modificación del modelo temático

La modificación de *models.py* son sólo dos líneas:

modelos.py

```
from django.db import modelos
from django.contrib.auth.models import Usuario

clase Topic(models.Model):
   """Un tema sobre el que el usuario está
    aprendiendo""" text =
    models.CharField(max_length=200)
    date_added = models.DateTimeField(auto_now_add=True)
    owner = models.ForeignKey(User)

    def str (self):
        """Devuelve una representación de cadena del
        modelo.""" return self.text

clase Entrada(modelos.Modelo):
    --snip--
```

Primero importamos el modelo `User` de `django.contrib.auth`. A continuación, añadimos un campo `owner` a `Topic`, que establece una relación de clave externa con el modelo `User`.

Identificar a los usuarios existentes

Cuando migremos la base de datos, Django modificará la base de datos para que pueda almacenar una conexión entre cada tema y un usuario. Para hacer la migración, Django necesita saber qué usuario asociar con cada tema existente. El enfoque más simple es dar todos los temas existentes a un usuario, por ejemplo, el superusuario. Primero, necesitamos saber el ID de ese usuario.

Veamos los ID de todos los usuarios creados hasta ahora. Inicia una sesión shell de Django y ejecuta los siguientes comandos:

```
(venv)learning_log$ python manage.py shell
❶ >>> from django.contrib.auth.models import User
❷  >>> User.objects.all()
[<Usuario: ll_admin>, <Usuario: eric>, <Usuario: willie>]
❸ >>> for user in Usuario.objetos.todos():
...       print(nombre.usuario, id.usuario)
...
ll_admin 1
eric 2
willie 3
>>>
```

En ❶ importamos el modelo de `Usuario` a la sesión shell. Luego miramos todos los usuarios que han sido creados hasta ahora ❷. La salida muestra tres usuarios: *ll_admin*, *eric*, y *willie*.

En ❸ hacemos un bucle a través de la lista de usuarios e imprimimos el

nombre de usuario y el ID de cada usuario. Cuando Django pregunte a qué usuario asociar los temas existentes, usaremos uno de estos valores de ID.

Migración de la base de datos

Ahora que conocemos los ID, podemos migrar la base de datos.

```
❶ (venv)learning_log$ python manage.py makemigrations learning_logs
❷ Está intentando añadir un campo no nulo 'owner' a topic sin un valor por
  defecto; no podemos hacerlo (la base de datos necesita algo para rellenar las
  filas existentes).
❸ Seleccione una solución:
   1) Proporcionar un único valor por defecto ahora (se establecerá en todas las
      filas existentes)
   2) Salir, y déjame añadir un valor por defecto en models.py
❹ Seleccione una opción: 1
❺ Por favor, introduzca el valor por defecto ahora, como Python válido.
  Los módulos datetime y django.utils.timezone están disponibles, por lo que
  puedes hacer
  por ejemplo timezone.now()
❻ >>> 1
  Migraciones para 'learning_logs':
    0003_topic_owner.py:
      - Añadir el propietario del campo al tema
```

Comenzamos emitiendo el comando `makemigrations` ❶. En la salida de ❷, Django indica que estamos intentando añadir un campo obligatorio (no anulable) a un modelo existente (`tema`) sin valor por defecto especificado. Django nos da dos opciones en ❸: podemos proporcionar un valor por defecto ahora mismo, o podemos abandonar y añadir un valor por defecto en *models.py*. En ❹ hemos elegido la primera opción. Django nos pide entonces que introduzcamos el valor por defecto ❺.

Para asociar todos los temas existentes con el usuario administrador original, *ll_admin*, introduje el ID de usuario de 1 en ❻. Puedes usar el ID de cualquier usuario que hayas creado; no tiene que ser un superusuario. Django entonces migra la base de datos usando este valor y genera el archivo de migración *0003_topic_owner.py*, que añade el campo `owner` al modelo `Topic`.

Ahora podemos realizar la migración. Introduzca lo siguiente en un entorno virtual activo:

```
(venv)learning_log$ python manage.py migrate
Operaciones a realizar:
  Sincronizar aplicaciones no migradas: mensajes, archivos estáticos
  Aplicar todas las migraciones: learning_logs, contenttypes, sessions, admin,
  auth
--snip--
Ejecución de migraciones:
  Renderizado de los estados del modelo... HECHO
❶ Aplicando learning_logs.0003_topic_owner...
OK (venv)learning_log$
```

Django aplica la nueva migración, y el resultado es `OK` ❶.

Podemos comprobar que la migración ha funcionado como se esperaba en la sesión de shell, de la siguiente manera:

```
❶ >>> from learning_logs.models import Tema
❷ >>> for tema in Tema.objetos.todos():
...         print(tema, tema.propietario)
```

```
...
Ajedrez ll_admin
Escalada ll_admin
>>>
```

Importamos `Topic` de `learning_logs.models` ❶ y luego hacemos un bucle a través de todos los temas existentes, imprimiendo cada tema y el usuario al que pertenece ❷. Puedes ver que cada tema pertenece ahora al usuario *ll_admin*.

NOTE *Puede simplemente reiniciar la base de datos en lugar de migrarla, pero perderá todos los datos existentes. Es una buena práctica aprender a migrar una base de datos manteniendo la integridad de los datos de los usuarios. Si quieres empezar con una base de datos nueva, ejecuta el* comando manage.py *para reconstruir la estructura de la base de datos. Tendrás que crear un nuevo superusuario, y todos tus datos desaparecerán.*

Restringir el acceso a los temas a los usuarios apropiados

Actualmente, si estás conectado, podrás ver todos los temas, independientemente del usuario con el que estés conectado. Vamos a cambiar esto mostrando a los usuarios sólo los temas que les pertenecen.

Realice el siguiente cambio en la función `topics()` de *views.py*:

views.py

```
--snip--
@login_required
def temas(solicitud):
    """Mostrar todos los
    temas."""
    topics = Topic.objects.filter(owner=request.user).order_by('date_added')
    context = {'topics': topics}
    return render(request, 'learning_logs/topics.html', context)
--snip--
```

Cuando un usuario inicia sesión, el objeto request tiene un atributo `request.user` que almacena información sobre el usuario. El fragmento de código `Topic.objects .filter(owner=request.user)` le dice a Django que recupere sólo los objetos `Topic` de la base de datos cuyo atributo `owner` coincida con el usuario actual. Debido a que no estamos cambiando la forma en que se muestran los temas, no necesitamos cambiar la plantilla de la página de temas en absoluto.

Para ver si esto funciona, inicia sesión como el usuario al que conectaste todos los temas existentes, y ve a la página de temas. Deberías ver todos los temas. Ahora cierra la sesión y vuelve a entrar como un usuario diferente. La página de temas no debería mostrar ningún tema.

Proteger los temas de un usuario

Aún no hemos restringido el acceso a las páginas temáticas, por lo que cualquier usuario registrado puede probar varias URL, como

http://localhost:8000/topics/1/, y recuperar las páginas temáticas que coincidan.

Pruébalo tú mismo. Mientras está conectado como el usuario propietario de todos los temas, copie la URL o anote el ID en la URL de un tema y, a continuación, cierre la sesión y vuelva a conectarse

como un usuario diferente. Introduce la URL de ese tema. Deberías poder leer las entradas, aunque estés conectado como un usuario diferente.

Arreglaremos esto ahora realizando una comprobación antes de recuperar las entradas solicitadas en la función de vista `topic()`:

vistas.py

```
from django.shortcuts import render
❶ from django.http import HttpResponseRedirect,
Http404 from django.core.urlresolvers import reverse
--snip--

@login_required
def tema(solicitud, topic_id):
    """Mostrar un único tema y todas sus
    entradas."""  topic =
    Topic.objects.get(id=topic_id)
    # Asegúrate de que el tema pertenece al usuario actual.
❷   if topic.owner != request.user:
        raise Http404

    entradas = topic.entry_set.order_by('-fecha_añadida')
    context = {'topic': topic, 'entradas': entries}
    return render(request, 'learning_logs/topic.html', context)
--snip--
```

Una respuesta 404 es una respuesta de error estándar que se devuelve cuando un recurso solicitado no existe en un servidor. Aquí importamos la excepción `Http404` ❶, que lanzaremos si el usuario solicita un tema que no debería ver. Después de recibir una petición de tema, nos aseguramos de que el usuario del tema coincide con el usuario actualmente conectado antes de mostrar la página. Si el usuario actual no posee el tema solicitado, lanzamos la excepción `Http404` ❷, y Django devuelve una página de error 404.

Ahora, si intentas ver las entradas de tema de otro usuario, verás un mensaje de *página no encontrada* de Django. En el Capítulo 20, configuraremos el proyecto para que los usuarios vean una página de error adecuada.

Protección de la página edit_entry

Las páginas `edit_entry` tienen URLs de la forma `http://localhost:8000/edit_entry/ entry_id/`, donde *entry_id* es un número. Protejamos esta página para que nadie pueda utilizar la URL para acceder a las entradas de otra persona:

views.py

```
--snip--
@login_required
def editar_entrada(solicitud,
    entrada_id): """Editar una
    entrada existente."""
    entrada = Entry.objects.get(id=entry_id)
    tema = entrada.tema
```

```
if topic.owner != request.user:
    raise Http404
```

```
    si request.method != 'POST':
        # Solicitud inicial; rellenar el formulario con la entrada actual.
        --snip--
```

Obtenemos la entrada y el tema asociado a esta entrada. A continuación, comprobamos si el propietario del tema coincide con el usuario actualmente conectado; si no coinciden, lanzamos una excepción `Http404`.

Asociar nuevos temas al usuario actual

Actualmente, nuestra página para añadir nuevos temas está rota, porque no asocia los nuevos temas a ningún usuario en particular. Si intentas añadir un nuevo tema, verás el mensaje de error `IntegrityError` junto con `learning_logs_topic.user_id no puede ser NULL`. Django está diciendo que no puedes crear un nuevo tema sin especificar un valor para el campo `owner` del tema.

Hay una solución sencilla para este problema, porque tenemos acceso al usuario actual a través del objeto `request`. Añade el siguiente código, que asocia el nuevo tema con el usuario actual:

views.py

```
--snip--
@login_required
def nuevo_tema(solicitud):
    """Añade un nuevo
    tema."""
    si request.method != 'POST':
        # No hay datos enviados; crear un formulario
        en blanco. form = TopicForm()
    si no:
        # POST datos enviados; procesar datos.
        form = TopicForm(request.POST)
        si form.is_valid():
❶           nuevo_tema = form.save(commit=False)
❷           nuevo_tema.propietario = solicitud.usuario
❸           nuevo_tema.guardar()
            return HttpResponseRedirect(reverse('learning_logs:topics'))

    context = {'formulario': formulario}
    return render(request, 'learning_logs/new_topic.html', context)
 --snip--
```

Cuando llamamos por primera vez a `form.save()`, pasamos el argumento `commit=False` porque necesitamos modificar el nuevo tema antes de guardarlo en la base de datos ❶. A continuación, establecemos el atributo `owner` del `nuevo tema` al usuario actual ❷. Finalmente, llamamos a `save()` en la instancia del tema que acabamos de definir ❸. Ahora el tema tiene todos los datos necesarios y se guardará con éxito.

Deberías poder añadir tantos temas nuevos como quieras para tantos usuarios diferentes como desees. Cada usuario sólo tendrá acceso a sus propios datos, ya sea para v i s u a l i z a r l o s , introducir datos nuevos o

modificarlos.
datos antiguos.

TRY IT yOURSELF

19-3. Refactorización: Hay dos lugares en *views.py* donde nos aseguramos de que el usuario asociado a un tema coincide con el usuario actualmente conectado. Ponga el código para esta comprobación en una función llamada `check_topic_owner()`, y llame a esta función donde sea apropiado.

19-4. Protección de new_entry: Un usuario puede añadir una nueva entrada al registro de aprendizaje de otro usuario introduciendo una URL con el ID de un tema perteneciente a otro usuario. Evite este ataque comprobando que el usuario actual es el propietario del tema de la entrada antes de guardar la nueva entrada.

19-5. Blog protegido: En su proyecto Blog, asegúrese de que cada entrada del blog está conectada a un usuario en particular. Asegúrese de que todas las entradas sean accesibles públicamente pero que sólo los usuarios registrados puedan añadir entradas y editar las existentes. En la vista que permite a los usuarios editar sus entradas, asegúrese de que el usuario está editando su propia entrada antes de acceder al formulario.

resumen

En este capítulo aprendiste a usar formularios para permitir a los usuarios añadir nuevos temas y entradas, y editar entradas existentes. Luego aprendiste a implementar cuentas de usuario. Permitiste a los usuarios existentes iniciar y cerrar sesión, y aprendiste a usar el `UserCreationForm` por defecto de Django para permitir a la gente crear nuevas cuentas.

Después de crear un sistema sencillo de autenticación y registro de usuarios, ha restringido el acceso a determinadas páginas a los usuarios que han iniciado sesión mediante el decorador `@login_ required`. Luego atribuyó datos a usuarios específicos a través de una relación de clave foránea. También aprendiste a migrar la base de datos cuando la migración requiere que especifiques algunos datos por defecto.

Por último, has aprendido a asegurarte de que un usuario sólo puede ver los datos que le pertenecen modificando las funciones de la vista. Recuperaste los datos apropiados utilizando el método `filter()`, y aprendiste a comparar el propietario de los datos solicitados con el usuario actualmente conectado.

Puede que no siempre sea inmediatamente obvio qué datos debes poner a disposición y qué datos debes proteger, pero esta habilidad se adquiere con la práctica. Las decisiones que hemos tomado en este capítulo para proteger los datos de nuestros usuarios ilustran por qué trabajar con otros es una buena idea cuando se construye un proyecto: tener a otra persona revisando tu proyecto hace más probable que detectes las áreas vulnerables.

Ahora tenemos un proyecto completamente funcional ejecutándose en nuestra máquina local. En el último capítulo daremos estilo a Learning Log para hacerlo visualmente atractivo, y desplegaremos el proyecto en un servidor para que cualquiera con acceso a Internet pueda registrarse y crear

una cuenta.

20

STyLING ANDEPLOyING ANAPP



Learning Log es completamente funcional ahora, pero no tiene estilo y sólo se ejecuta en su máquina local. En este capítulo vamos a estilizar el proj-

ect de una manera sencilla pero profesional y luego desplegarlo en un servidor en vivo para que cualquier persona en el mundo puede hacer una cuenta.

Para el diseño utilizaremos la librería Bootstrap, una colección de herramientas para diseñar aplicaciones web de forma que tengan un aspecto profesional en todos los dispositivos modernos, desde una gran pantalla plana hasta un smartphone. Para e11o, utilizaremos la aplicación django-bootstrap3, que también te permitirá practicar el uso de aplicaciones creadas por otros desarrolladores de Django.

Desplegaremos Learning Log utilizando Heroku, un sitio que te permite empujar tu proyecto a uno de sus servidores, poniéndolo a disposición de cualquier persona con una conexión a Internet. También vamos a empezar a utilizar un sistema de control de versiones llamado Git para realizar un seguimiento de los cambios en el proyecto.

Cuando termine Learning Log, será capaz de desarrollar aplicaciones web sencillas, hacer que tengan buen aspecto y desplegarlas en un servidor activo. También podrás utilizar recursos de aprendizaje más avanzados a medida que desarrolles tus habilidades.

Registro de aprendizaje de estilismo

Hasta ahora hemos ignorado deliberadamente el estilo para centrarnos primero en la funcionalidad de Learning Log. Es una buena forma de enfocar el desarrollo, porque una aplicación sólo es útil si funciona. Por supuesto, una vez que funciona, la apariencia es crítica para que la gente quiera usarla.

En esta sección presentaré la aplicación django-bootstrap3 y te mostraré cómo integrarla en un proyecto para que esté lista para su despliegue.

La aplicación django-bootstrap3

Usaremos django-bootstrap3 para integrar Bootstrap en nuestro proyecto. Esta aplicación descarga los archivos de Bootstrap necesarios, los coloca en una ubicación adecuada de tu proyecto y hace que las directivas de estilo estén disponibles en las plantillas de tu proyecto.

Para instalar django-bootstrap3, ejecuta el siguiente comando en un entorno virtual activo:

```
(ll_env)learning_log$ pip install django-bootstrap3
--snip--
Instalado con éxito django-bootstrap3
```

A continuación, tenemos que añadir el siguiente código para incluir django-boostrap3 en `INSTALLED_APPS` en *settings.py*:

settings.py

```
--snip--
APLICACIONES_INSTALADAS = (
    --snip--
    'django.contrib.staticfiles',

    # Aplicaciones de
    terceros 'bootstrap3',

    # Mis aplicaciones
    'learning_logs',
    'users',
)
--snip--
```

Crea una nueva sección llamada *Aplicaciones de terceros* para aplicaciones creadas por otros desarrolladores y añade `'bootstrap3'` a esta sección. La mayoría de las aplicaciones deben incluirse en `INSTALLED_APPS`, pero para estar seguro, lee las instrucciones de configuración de la aplicación concreta que estés utilizando.

Necesitamos que django-bootstrap3 incluya jQuery, una librería JavaScript que habilita algunos de los elementos interactivos que proporciona la plantilla Bootstrap. Añade este código al final de *settings.py*:

settings.py

```
--snip--
# Mi configuración
LOGIN_URL = '/usuarios/acceso/'

# Settings for django-bootstrap3
BOOTSTRAP3 = {
    'include_jquery': True,
    }
```

Este código nos evita tener que descargar jQuery y colocarlo en la ubicación correcta manualmente.

Uso de Bootstrap para estilizar el registro de aprendizaje

Bootstrap es básicamente una gran colección de herramientas de estilo. También tiene un número de plantillas que puedes aplicar a tu proyecto para crear un estilo general particular. Si estás empezando, es mucho más fácil utilizar estas plantillas que las herramientas de estilo individuales. Para ver las plantillas que ofrece Bootstrap, ve a la sección *Primeros pasos* en http://getbootstrap.com/; a continuación, desplázate hacia abajo hasta el encabezado *Ejemplos* y busca la sección *Barras de navegación en acción*. Utilizaremos la plantilla *Static top navbar*, que proporciona una sencilla barra de navegación superior, una cabecera de página y un contenedor para el contenido de la página.

La Figura 20-1 muestra el aspecto que tendrá la página de inicio después de que apliquemos la plantilla de Bootstrap a *base.html* y modifiquemos ligeramente *index.html*.

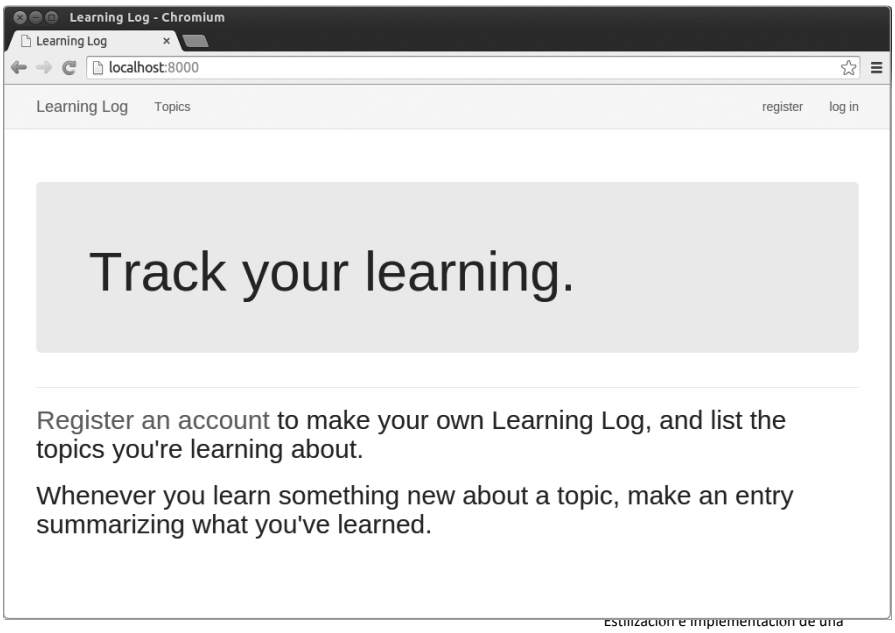


Figura 20-1: Página de inicio de Learning Log con Bootstrap

Ahora que conoces el resultado que buscamos, las secciones que siguen serán más fáciles de entender.

Modificación de base.html

Necesitamos modificar la plantilla base.*html* para acomodar la plantilla Bootstrap. Presentaré el nuevo base.*html* por partes.

Definición de las cabeceras HTML

El primer cambio en *base.html* define las cabeceras HTML en el archivo para que cuando se abra una página Learning Log, la barra de título del navegador muestre el nombre del sitio. También añadiremos algunos requisitos para usar Bootstrap en nuestras plantillas.
Borra todo lo que haya en *base.html* y sustitúyelo por el siguiente código:

base.html

```
❶ {% load bootstrap3 %}

❷ <!DOCTYPE html>
❸ <html lang="es">
❹  <head>
     <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale=1">

❺    <title>Registro de aprendizaje</title>

❻    {% bootstrap_css %}
     {% bootstrap_javascript %}

❼  </head>
```

En ❶ cargamos la colección de etiquetas de plantilla disponibles en django- bootstrap3. A continuación, declaramos este archivo como un documento HTML ❷ escrito en inglés ❸. Un archivo HTML se divide en dos partes principales, la *cabeza* y el *cuerpo-la* cabeza del archivo comienza en ❹. La cabeza de un archivo HTML no contiene ningún contenido: sólo indica al navegador lo que necesita saber para mostrar la página correctamente. En ❺ incluimos un elemento `title` para la página, que se mostrará en la barra de título del navegador siempre que Learning Log esté abierto.

En ❻ usamos una de las etiquetas de plantilla personalizadas de django-bootstrap3, que le dice a Django que incluya todos los archivos de estilo de Bootstrap. La etiqueta que sigue habilita todo el comportamiento interactivo que podrías usar en una página, como barras de navegación col- lapsibles. En ❼ se encuentra la etiqueta de cierre `</head>`.

Definición de la barra de navegación

Ahora definiremos la barra de navegación en la parte superior de la página:

```
--snip--
```

```
</head>

<body>

  <!-- Barra de navegación estática -->
```

```
❶   <nav class="navbar navbar-default navbar-static-top">
      <div class="container">

        <div class="navbar-header">
❷         <button type="button" class="navbar-toggle collapsed"
              data-toggle="collapse" data-target="#navbar"
              aria-expanded="false" aria-controls="navbar">
          </button>
❸         <a class="navbar-brand" href="{% url 'learning_logs:index' %}">
              Registro de aprendizaje</a>
        </div>

❹       <div id="navbar" class="navbar-collapse collapse">
❺         <ul class="navbar-nav">
❻           <li><a href="{% url 'learning_logs:topics' %}">Temas</a></li>
          </ul>

❼         <ul class="nav navbar-nav navbar-right">
            {% if user.is_authenticated %}
              <li><a>Hola, {{ user.username }}.</a></li>.
              <li><a href="{% url 'usuarios:cerrar sesión' %}">cerrar
              sesión</a></li>
            {% else %}
              <li><a href="{% url 'usuarios:registrarse'
              %}">registrarse</a></li>
              <li><a href="{% url 'users:login' %}">iniciar sesión</a></li>
            {% endif %}
❽         </ul>
        </div><!--/.nav-collapse -->

      </div>
    </nodo>
```

El primer elemento es la etiqueta `<body>`. El cuerpo de un archivo HTML contiene el contenido que los usuarios verán en una página. A ❶ es un elemento `<nav>` que indica la sección de enlaces de navegación de la página. Todo lo que contiene este elemento está estilizado según las reglas de estilo de Bootstrap definidas por los selectores `navbar`, `navbar-default` y `navbar-static-top`. Un *selector* determina a qué elementos de una página se aplica una determinada regla de estilo.

En ❷ la plantilla define un botón que aparecerá si la ventana del navegador es demasiado estrecha para mostrar horizontalmente toda la barra de navegación. Cuando el usuario pulse el b o t ó n , los elementos de navegación aparecerán en una lista desplegable. La referencia de `colapso` hace que la barra de navegación se colapse cuando el usuario reduce la ventana del navegador o cuando el sitio se muestra en dispositivos móviles con pantallas pequeñas.

En ❸ configuramos el nombre del proyecto para que aparezca en el extremo izquierdo de la barra de navegación y lo convertimos en un enlace a la página de inicio, porque aparecerá en todas las páginas del proyecto.

En ❹ definimos un conjunto de enlaces que permite a los usuarios navegar por el sitio. Una barra de navegación es básicamente una lista que

empieza por `<ul>` ❺, y cada enlace es un elemento de esta lista (`<li>`) ❻. Para añadir más enlaces, inserte más líneas utilizando la siguiente estructura:

```
<li><a href="{% url 'learning_logs:title' %}">Título</a></li>
```

Esta línea representa un único enlace en la barra de navegación. El enlace está tomado directamente de la versión anterior de *base.html*.

En ❼ colocamos una segunda lista de enlaces de navegación, esta vez utilizando el selector `navbar-right`. El selector `navbar-right` estiliza el conjunto de enlaces para que aparezca en el borde derecho de la barra de navegación, donde normalmente se ven los enlaces de inicio de sesión y registro. Aquí mostraremos el saludo al usuario y el enlace de cierre de sesión o los enlaces para registrarse o iniciar sesión. El resto del código en esta sección cierra los elementos que contienen la barra de navegación ❽.

Definir la parte principal de la página

El resto de *base.html* contiene la parte principal de la página:

```
--snip--
    </nodo>

❶   <div class="container">

      <div class="page-header">
❷       {% block header %}{% endblock header %}
      </div>
      <div>
❸       {% block content %}{% endblock content %}
      </div>

    </div> <!-- /container -->

  </body>
</html>
```

En ❶ es un div de apertura con el `contenedor de` clase. Un *div* es una sección de una página web que puede utilizarse para cualquier propósito y a la que se puede aplicar estilo con un borde, espacio alrededor del elemento (márgenes), espacio entre el contenido y el borde (relleno), colores de fondo y otras reglas de estilo. Este div en concreto actúa como contenedor en el que colocamos dos elementos: un nuevo bloque llamado `cabecera` ❷ y el bloque de `contenido` que utilizamos en el capítulo 18 ❸. El bloque de cabecera contiene información que indica al usuario qué tipo de información contiene la página y qué puede hacer en ella. Tiene la clase `page-header`, que aplica un conjunto de reglas de estilo al bloque. El bloque de contenido está en un div separado sin clases de estilo específicas.

Cuando cargue la página de inicio de Learning Log en un navegador, debería ver una barra de navegación de aspecto profesional que coincida con la que se muestra en la Figura 20-1. Intente cambiar el tamaño de la ventana para que sea muy estrecha; la barra de navegación debería ser sustituida por un botón. Haga clic en el botón y todos los enlaces aparecerán en una lista desplegable.

NOTE *Esta versión simplificada de la plantilla Bootstrap debería funcionar en los*

navegadores más recientes. Es posible que los navegadores anteriores no muestren correctamente algunos estilos. La plantilla completa, disponible en http://getbootstrap.com/getting-started/#examples/, funcionará en casi todos los navegadores disponibles.

Estilizar la página de inicio con un Jumbotron

Actualicemos la página de inicio utilizando el bloque de `cabecera` recién definido y otro elemento de Bootstrap llamado *jumbotrón: un* gran recuadro que destacará del resto de la página y que puede contener cualquier cosa que deseemos. Se suele utilizar en las páginas de inicio para incluir una breve descripción del proyecto. De paso, actualizaremos también el mensaje de la página de inicio. Aquí está *index.html*:

index.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% encabezado de bloque %}
  <div class='jumbotron'>
    <h1>Sigue tu aprendizaje.</h1>
  </div>
{% endblock header %}

{% block content %}
  <h2>
    <a href="{% url 'users:register' %}">Registra una cuenta</a> para
    crear tu propio Registro de aprendizaje y enumerar los temas sobre
    los que estás aprendiendo.
  </h2>
  <h2>
    Cada vez que aprendas algo nuevo sobre un tema, haz una entrada
    resumiendo lo que has aprendido.
  </h2>
{% endblock content %}
```

En ❶ le decimos a Django que estamos a punto de definir lo que va en el bloque de `cabecera`. Dentro de un elemento `jumbotron` ❷ colocamos un breve eslogan, *Track your learning*, para dar a los primeros visitantes una idea de lo que hace Learning Log.

En ❸ añadimos texto para dar un poco más de orientación. Invitamos a la gente a crear una cuenta y describimos las dos acciones principales: añadir nuevos temas y hacer entradas de temas. La página de índice tiene ahora el aspecto de la Figura 20-1 y supone una mejora significativa con respecto a nuestro proyecto sin estilo.

Diseño de la página de inicio de sesión

Hemos refinado la apariencia general de la página de inicio de sesión, pero aún no el formulario de inicio de sesión, así que vamos a hacer que el formulario tenga un aspecto coherente con el resto de la página:

login.html

```
{% extends "registros_de_aprendizaje/base.html" %}
{% load bootstrap3 %}

{% encabezado de bloque %}
  <h2>Conéctese a su cuenta.</h2>
{% endblock header %}

{% block content %}
```

❸ <form method="post" action="{% url 'users:login' %}" class="form">
{% csrf_token %}

```
❹     {% bootstrap_form form %}

❺     {% botones %}
        <button name="submit" class="btn btn-primary">iniciar sesión</button>
      {% endbuttons %}

      <input type="hidden" name="next" value="{% url 'learning_logs:index' %}" />
    </form>

{% endblock content %}
```

En ❶ cargamos las etiquetas de la plantilla bootstrap3 en esta plantilla. En ❷ definimos el bloque de `cabecera`, que describe para qué es la página. Observa que hemos eliminado el bloque `{% if form.errors %}` de la plantilla; django- bootstrap3 gestiona los errores de formulario automáticamente.

En ❸ añadimos un atributo `class="form"`, y luego utilizamos la etiqueta de plantilla `{% bootstrap_form %}` cuando mostramos el formulario ❹; esto sustituye a la etiqueta `{{ form.as_p }}` que usamos en el Capítulo 19. La etiqueta de plantilla `{% booostrap_form %}` inserta reglas de estilo Bootstrap en los elementos individuales del formulario a medida que se renderiza. En ❺ abrimos una etiqueta de plantilla bootstrap3 `{% buttons %}`, que añade estilo Bootstrap a los botones.

La Figura 20-2 muestra el formulario de acceso tal y como se muestra ahora. La página está mucho más limpia y tiene un estilo consistente y un propósito claro. Intenta iniciar sesión con un nombre de usuario o contraseña incorrectos; verás que incluso los mensajes de error tienen un estilo consistente y se integran bien con el sitio en general.

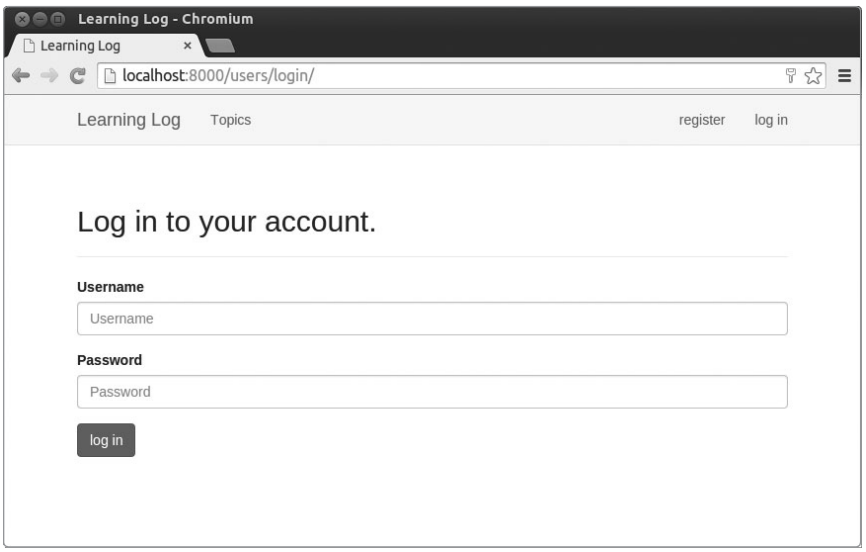


Figura 20-2: Página de inicio de sesión diseñada con Bootstrap

Estilizar la página new_topic

Hagamos que el resto de las páginas también tengan un aspecto coherente. Actualizaremos el
nuevo_tema página siguiente:

nuevo_tema.html

```
{% extends "registros_de_aprendizaje/base.html" %}
{% load bootstrap3 %}

❶ {% encabezado de bloque %}
    <h2>Añadir un nuevo tema:</h2>
  {% endblock header %}

  {% block content %}

❷  <form action="{% url 'learning_logs:new_topic' %}" method='post'
       class="form">

     {% csrf_token %}
❸    {% bootstrap_form form %}

❹    {% botones %}
       <button name="submit" class="btn btn-primary">añadir tema</button>
     {% endbuttons %}

   </form>

  {% endblock content %}
```

La mayoría de los cambios aquí son similares a los aplicados en *login.html*: cargamos bootstrap3 y añadimos el bloque de `cabecera` con un mensaje apropiado en ❶. A continuación, añadimos el atributo `class="form"` a la etiqueta `<form>` ❷, usamos el atributo `{% bootstrap_form %}` etiqueta de plantilla en lugar de `{{ form.as_p }}` ❸, y usa la estructura bootstrap3 para el botón de enviar ❹. Inicia sesión y navega hasta la página `new_topic`; ahora debería tener un aspecto similar al de la página de inicio de sesión.

Estilizar la página de temas

Ahora vamos a asegurarnos de que las páginas para ver la información también tienen el estilo adecuado, empezando por la página de temas:

temas.html

```
{% extends "registros_de_aprendizaje/base.html" %}

❶ {% encabezado de bloque %}
     <h1>Temas</h1>
  {% endblock header %}

  {% block content %}
```

```
<ul>
  {% for tema in temas %}
    <li>
```

```
❷         <h3>
            <a href="{% url 'learning_logs:topic' topic.id %}">{{ topic }}</a>
          </h3>
        </li> <li>
      {% empty %}
        <li>Aún no se ha añadido ningún tema.</li>.
      {% endfor %}
    </ul>

❸   <h3><a href="{% url 'learning_logs:new_topic' %}">Añadir nuevo tema</h3>

    {% endblock content %}
```

No necesitamos la etiqueta `{% load bootstrap3 %}`, porque no estamos usando ninguna etiqueta personalizada de plantilla bootstrap3 en este archivo.

Añadimos el encabezado *Temas* dentro del bloque de `cabecera` ❶. Estilizamos cada tema como un elemento `<h3>` para hacerlos un poco más grandes en la página ❷ y hacemos lo mismo con el enlace para añadir un nuevo tema ❸.

Estilizar las entradas de la página de temas

La página de temas tiene más contenido que la mayoría de las páginas, por lo que necesita un poco más de trabajo. Utilizaremos los paneles de Bootstrap para destacar cada entrada. Un *panel* es un div con un estilo predefinido y es perfecto para mostrar las entradas de un tema:

topic.html

```
{% extends 'learning_logs/base.html' %}

❶ {% encabezado de bloque %}
    <h2>{{ tema }}</h2>
  {% endblock header %}

  {% block content %}
    <p>
      <a href="{% url 'learning_logs:new_entry' topic.id %}">añadir nueva
      entrada</a>
    </p>

    {% for entrada in entradas %}
❷     <div class="panel panel-default">
❸       <div class="panel-heading">
❹         <h3>
            {{ entry.date_added|date:'M d, Y H:i' }}
❺           <small>
              <a href="{% url 'learning_logs:edit_entry' entry.id %}">
                editar entrada</a>
            </small>
          </h3>
        </div>
❻       <div class="panel-body">
          {{ entry.text|linebreaks }}
        </div>
      </div> <!-- panel -->
```

```
    {% empty %}
      Aún no hay entradas para este tema.
    {% endfor %}

{% endblock content %}
```

Primero colocamos el tema en el bloque de `cabecera` ❶. A continuación, eliminamos la estructura de lista desordenada utilizada anteriormente en esta plantilla. En lugar de hacer de cada entrada un elemento de lista, creamos un elemento div de `panel` en ❷, que contiene otros dos divs anidados: un div de encabezado de `panel` ❸ y un div de `cuerpo de panel` ❻. El div `panel-encabezado` contiene la fecha de la entrada y el enlace para editarla. Ambos están estilizados como elementos `<h3>` ❹, pero añadimos etiquetas `<small>` alrededor del `enlace edit_entry` para hacerlo un poco más pequeño que la marca de tiempo ❺.

En ❻ está el div `panel-body`, que contiene el texto real de la entrada. Observa que el código de Django para incluir la información en la página no ha cambiado en absoluto; solo han cambiado los elementos que afectan a la apariencia de la página.

La Figura 20-3 muestra la página de temas con su nuevo aspecto. La funcionalidad de Learning Log no ha cambiado, pero tiene un aspecto más profesional y atractivo para los usuarios.

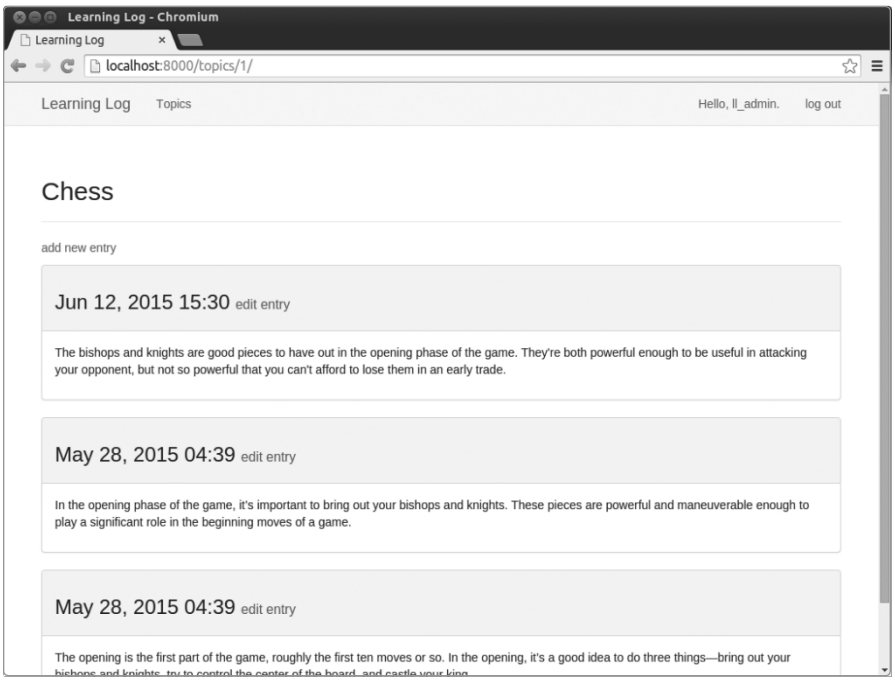


Figura 20-3: Página del tema con estilo Bootstrap

NOTE *Si deseas utilizar una plantilla Bootstrap diferente, sigue un proceso similar al que hemos realizado hasta ahora en este capítulo. Copie la plantilla en* base.html *y modifique los elementos que contienen contenido real para que la plantilla muestre la información de su proyecto. A continuación, utiliza las herramientas de estilo individuales de Bootstrap para dar estilo al contenido de cada página.*

TRY IT yOURSELF

20-1. Otros formularios: Hemos aplicado los estilos de Bootstrap a las páginas `login` y `add_topic`. Haz cambios similares en el resto de las páginas basadas en formularios: `new_entry` y `edit_entry`, y `register`.

20-2. Blog con estilo: Utiliza Bootstrap para dar estilo al proyecto Blog que creaste en el Capítulo 19.

Despliegue de Learning Log

Ahora que tenemos un proyecto de aspecto profesional, vamos a desplegarlo en un servidor en vivo para que cualquier persona con una conexión a Internet pueda utilizarlo. Utilizaremos Heroku, una plataforma basada en web que te permite gestionar el despliegue de aplicaciones web. Pondremos Learning Log en marcha en Heroku.

El proceso es ligeramente diferente en Windows que en Linux y OS X. Si utilizas Windows, busca notas en cada sección que especifiquen lo que tendrás que hacer de forma diferente en tu sistema.

Crear una cuenta Heroku

Para crear una cuenta, ve a *https://heroku.com/* y haz clic en uno de los enlaces de registro. Crear una cuenta es gratis, y Heroku tiene un nivel gratuito que te permite probar tus proyectos en despliegue real.

NOTE *El nivel gratuito de Heroku tiene límites, como el número de aplicaciones que puedes desplegar y la frecuencia con la que la gente puede visitar tu aplicación. Pero estos límites son lo suficientemente generosos como para permitirte practicar el despliegue de aplicaciones sin coste alguno.*

Instalación de Heroku Toolbelt

Para desplegar y gestionar un proyecto en los servidores de Heroku, necesitarás las herramientas disponibles en Heroku Toolbelt. Para instalar la última versión de Heroku Toolbelt, visita *https://toolbelt.heroku.com/* y sigue las instrucciones para tu sistema operativo, que incluirán un comando de terminal de una línea o un instalador que puedes descargar y ejecutar.

Instalación de los paquetes necesarios

También necesitarás instalar una serie de paquetes que ayudan a servir proyectos Django en un servidor activo. En un entorno virtual activo, ejecuta los siguientes comandos:

```
(ll_env)learning_log$ pip install dj-database-url
(ll_env)learning_log$ pip install dj-static
```

```
(ll_env)learning_log$ pip install static3
(ll_env)learning_log$ pip install gunicorn
```

Asegúrate de ejecutar los comandos de uno en uno para saber si algún paquete no se instala correctamente. El paquete `dj-database-url` ayuda a Django a comunicarse con la base de datos que utiliza Heroku, `dj-static` y `static3` ayudan a Django a gestionar correctamente los archivos estáticos, y `gunicorn` es un servidor capaz de servir aplicaciones en un entorno en vivo. (Los archivos *estáticos* contienen reglas de estilo y archivos JavaScript).

NOTE *Algunos de los paquetes requeridos pueden no instalarse en Windows, así que no te preocupes si recibes un mensaje de error cuando intentas instalar algunos de ellos. Lo que importa es conseguir que Heroku instale los paquetes en el despliegue en vivo, y lo haremos en la siguiente sección.*

Creación de una lista de paquetes con un archivo requirements.txt

Heroku necesita saber de qué paquetes depende nuestro proyecto, así que usaremos pip para generar un archivo que los enumere. De nuevo, desde un entorno virtual activo, emite el siguiente comando:

```
(ll_env)learning_log$ pip freeze > requirements.txt
```

El comando `freeze` le dice a pip que escriba los nombres de todos los paquetes actualmente instalados en el proyecto en el archivo requirements*.txt*. Abra *requirements.txt* para ver los paquetes y números de versión instalados en su proyecto (los usuarios de Windows podrían no ver todas estas líneas):

requisitos.txt

```
Django==1.8.4
dj-database-url==0.3.0
dj-static==0.0.6
django-bootstrap3==6.2.2
gunicorn==19.3.0
static3==0.6.1
```

Learning Log ya depende de seis paquetes diferentes con números de versión específicos, por lo que requiere un entorno específico para ejecutarse correctamente. Cuando desplegamos Learning Log, Heroku instalará todos los paquetes listados en *requirements.txt*, creando un entorno con los mismos paquetes que estamos usando localmente. Por esta razón, podemos estar seguros de que el proyecto desplegado se comportará igual que en nuestro sistema local. Esta es una gran ventaja cuando empiezas a construir y mantener varios proyectos en tu sistema.

A continuación, tenemos que añadir `psycopg2`, que ayuda a Heroku a gestionar la base de datos en vivo, a la lista de paquetes. Abre *requirements.txt* y añade la línea `psycopg2>=2.6.1`. Esto instalará la versión 2.6.1 de `psycopg2`, o una versión más reciente si está disponible:

requisitos.txt

```
Django==1.8.4
dj-database-url==0.3.0
dj-static==0.0.6
django-bootstrap3==6.2.2
```

```
gunicorn==19.3.0
```

```
static3==0.6.1
psycopg2>=2.6.1
```

Si alguno de los paquetes no se instaló en tu sistema, añádelo también. Cuando hayas terminado, tu archivo *requirements.txt* debería incluir cada uno de los paquetes mostrados arriba. Si un paquete aparece en su sistema pero el número de versión difiere del que se muestra aquí, mantenga la versión que tiene en su sistema.

NOTE *Si utiliza Windows, asegúrese de que su versión de* requirements.txt *coincide con la lista que se muestra aquí, independientemente de los paquetes que haya podido instalar en su sistema.*

Especificación del tiempo de ejecución de Python

A menos que especifiques una versión de Python, Heroku usará su propia versión actual de Python por defecto. Asegurémonos de que Heroku usa la misma versión de Python que estamos usando. En un entorno virtual activo, emite el comando `python --versión`:

```
(ll_env)learning_log$ python --version
Python 3.5.0
```

En este ejemplo estoy ejecutando Python 3.5.0. Crea un nuevo archivo llamado *runtime.txt* en el mismo directorio que *manage.py*, e introduce lo siguiente:

runtime.txt

```
python-3.5.0
```

Este archivo debe contener una línea con su versión de Python especificada en el formato mostrado; asegúrese de introducir `python` en minúsculas, seguido de un guión, seguido del número de versión en tres partes.

NOTE *Si aparece un error informando de que el tiempo de ejecución de Python que ha solicitado no está disponible, vaya a* https://devcenter.heroku.com/ *y haga clic en* ***Python****; a continuación, busque un enlace a* Especificar un tiempo de ejecución de Python*. Busca en el artículo los tiempos de ejecución disponibles y utiliza el que más se aproxime a tu versión de Python.*

Modificación de settings.py para Heroku

Ahora necesitamos añadir una sección al final de *settings.py* para definir algunas configuraciones específicas para el entorno Heroku:

settings.py

```
--snip--
# Settings for django-bootstrap3
BOOTSTRAP3 = {
    'include_jquery': True,
    }
```

```
# Configuración de Heroku
❶ if os.getcwd() == '/app':
❷     import dj_database_url
      BASES DE DATOS = {
          default': dj_database_url.config(default='postgres://localhost')
      }

      # Respetar la cabecera 'X-Forwarded-Proto' para request.is_secure().
❸     SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

      # Permitir todas las cabeceras de host.
❹     ALLOWED_HOSTS = ['*']

      # Static asset configuration
❺     BASE_DIR = os.path.dirname(os.path.abspath( file ))
      STATIC_ROOT = 'staticfiles'
      STATICFILES_DIRS = (
          os.path.join(BASE_DIR, 'static'),
      )
```

En ❶ usamos la función `getcwd()`, que obtiene el directorio de *trabajo actual* desde el que se está ejecutando el archivo. En un despliegue Heroku, el directorio es siempre */app*. En un despliegue local, el directorio suele ser el nombre de la carpeta del proyecto (*learning_log* en nuestro caso). La prueba `if` garantiza que la configuración de este bloque sólo se aplica cuando el proyecto se despliega en Heroku. Esta estructura nos permite tener un archivo de configuración que funciona para nuestro entorno de desarrollo local, así como el servidor en vivo.

En ❷ importamos `dj_database_url` para ayudar a configurar la base de datos en Heroku. Heroku utiliza PostgreSQL (también llamado Postgres), una base de datos más avanzada que SQLite, y estos ajustes configuran el proyecto para utilizar Postgres en Heroku. El resto de los ajustes soportan peticiones HTTPS ❸, aseguran que Django servirá el proyecto desde la URL de Heroku ❹, y configuran el proyecto para servir archivos estáticos correctamente en Heroku ❺.

Creación de un Procfile para iniciar procesos

Un *Procfile* le dice a Heroku qué procesos iniciar para servir el proyecto correctamente. Se trata de un archivo de una línea que debes guardar como *Procfile*, con una *P* mayúscula y sin extensión de archivo, en el mismo directorio que *manage.py*.

Esto es lo que va en *Procfile*:

Procfile
```
web: gunicorn learning_log.wsgi --log-file -
```

Esta línea le dice a Heroku que use gunicorn como servidor y que use la configuración en *learning_log/wsgi.py* para lanzar la aplicación. La bandera `log-file` le dice a Heroku los tipos de eventos a registrar.

Modificación de wsgi.py para Heroku

También tenemos que modificar *wsgi.py* para Heroku, porque Heroku necesita una configuración ligeramente diferente a la que hemos estado utilizando:

wsgi.py

```
--snip--
import os

from django.core.wsgi import get_wsgi_application
from dj_static import Cling

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "learning_log.settings")
application = Cling(get_wsgi_application())
```

Importamos Cling, que ayuda a servir archivos estáticos correctamente, y lo usamos para lanzar la aplicación. Este código funcionará localmente también, así que no necesitamos ponerlo en un bloque `if`.

Crear un directorio para archivos estáticos

En Heroku, Django recoge todos los archivos estáticos y los coloca en un solo lugar para que pueda gestionarlos de manera eficiente. Crearemos un directorio para estos archivos estáticos. Dentro de la carpeta learning_log desde la que hemos estado trabajando hay otra carpeta llamada *learning_log*. En esta carpeta anidada, cree una nueva carpeta llamada *static* con la ruta *learning_log/learning_log/static/*. También tenemos que hacer un archivo de marcador de posición para almacenar en este directorio por ahora, porque los directorios vacíos no se incluirán en el proyecto cuando se empuja a Heroku. En el directorio *static/*, crea un archivo llamado *placeholder.txt*:

marcador.txt

```
Este archivo asegura que learning_log/static/ será añadido al proyecto.
Django recopilará los archivos estáticos y los colocará en
learning_log/static/.
```

Este texto no tiene nada de especial; sólo nos recuerda por qué incluimos este archivo en el proyecto.

Uso local del servidor gunicorn

Si estás usando Linux o OS X, puedes probar a usar el servidor gunicorn localmente antes de desplegarlo en Heroku. Desde un entorno virtual activo, ejecuta el comando `heroku local` para iniciar los procesos definidos en *Procfile* :

```
(ll_env)learning_log$ heroku local
Instalando Heroku Toolbelt v4... hecho
--snip--
forego | starting web.1 on port 5000
❶ web.1 | [2015-08-13 22:00:45 -0800] [12875] [INFO] Iniciando gunicorn 19.3.0
❷ web.1 | [2015-08-13 22:00:45 -0800] [12875] [INFO] Escuchando en:
      http://0.0.0.0:5000 (12875)
❸ web.1 | [2015-08-13 22:00:45 -0800] [12878] [INFO] Arrancando trabajador con pid:
12878
```

La primera vez que ejecutes `heroku local`, un número de paquetes de Heroku Toolbelt serán instalados. La salida muestra que gunicorn se ha iniciado con un id de proceso de 12875 en este ejemplo ❶. En ❷ gunicorn está recibiendo peticiones en el puerto 5000. Además, gunicorn ha iniciado un *proceso trabajador* (12878) para ayudarle a servir peticiones ❸.

Visita *http://localhost:5000/* para asegurarte de que todo funciona; deberías ver la página de inicio de Learning Log, tal y como aparece cuando usas el servidor Django (`runserver`). Presiona CTRL-C para detener los procesos iniciados por `heroku local`. Deberías continuar usando `runserver` para el desarrollo local.

NOTE *gunicorn no se ejecutará en Windows, así que omita este paso si está utilizando Windows. Esto no afectará a su capacidad para implementar el proyecto en Heroku.*

Uso de Git para rastrear los archivos del proyecto

Si has completado el Capítulo 17, sabrás que Git es un programa de control de versiones que te permite tomar una instantánea del código de tu proyecto cada vez que implementas una nueva característica con éxito. Esto te permite volver fácilmente a la última instantánea de trabajo de tu proyecto si algo va mal; por ejemplo, si accidentalmente introduces un error mientras trabajas en una nueva característica. Cada una de estas instantáneas se denomina *commit*.

Usar Git significa que puedes probar nuevas funcionalidades sin preocuparte de romper tu proyecto. Cuando despliegues en un servidor activo, debes asegurarte de que estás desplegando una versión funcional de tu proyecto. Si quieres leer más sobre Git y el control de versiones, consulta el Apéndice D.

Instalación de Git

Heroku Toolbelt incluye Git, por lo que ya debería estar instalado en tu sistema. Pero las ventanas de terminal que estaban abiertas antes de instalar Heroku Toolbelt no tendrán acceso a Git, así que abre una nueva ventana de terminal y ejecuta el comando `git --version`:

```
(ll_env)learning_log$ git --version
git versión 2.5.0
```

Si por alguna razón recibe un mensaje de error, consulte las instrucciones del Apéndice D para instalar Git.

Configuración de Git

Git mantiene un registro de quién realiza cambios en un proyecto, incluso en casos como éste, cuando sólo hay una persona trabajando en el proyecto. Para ello, Git necesita saber tu nombre de usuario y tu correo electrónico.

Tienes que proporcionar un nombre de usuario, pero siéntete libre de inventar un correo electrónico para tus proyectos de práctica:

```
(ll_env)learning_log$ git config --global user.name "ehmatthes"
(ll_env)learning_log$ git config --global user.email "eric@example.com"
```

Si olvidas este paso, Git te pedirá esta información cuando hagas tu primer commit.

Ignorar archivos

No necesitamos que Git rastree todos los archivos del proyecto, así que le diremos a Git que ignore algunos archivos. Crea un archivo llamado *.gitignore* en la carpeta que contiene *manage.py*. Fíjate en que el nombre de este archivo empieza con un punto y no tiene extensión. Esto es lo que va en *.gitignore*:

.gitignore

```
ll_env/
 pycache /
*.sqlite3
```

Le decimos a Git que ignore todo el directorio *ll_env*, porque podemos recrearlo automáticamente en cualquier momento. Tampoco rastreamos el *p y c a c h e* que contiene los archivos *.pyc* que se crean automáticamente cuando Django ejecuta los archivos *.py*. No rastreamos los cambios en la base de datos local, porque es un mal hábito: si alguna vez usas SQLite en un servidor, podrías accidentalmente sobrescribir la base de datos en vivo con tu base de datos de prueba local cuando empujas el proyecto al servidor.

NOTE *Si utilizas Python 2.7, sustituye* pycache *por* *.pyc *porque Python 2.7 no crea un directorio* pycache.

Comprometer el proyecto

Necesitamos inicializar un repositorio Git para Learning Log, añadir todos los archivos necesarios al repositorio, y confirmar el estado inicial del proyecto. He aquí cómo hacerlo:

```
❶ (ll_env)learning_log$ git init
Inicializado repositorio Git vacío en /home/ehmatthes/pcc/learning_log/.git/
❷ (ll_env)learning_log$ git add .
❸ (ll_env)learning_log$ git commit -am "Listo para el despliegue en heroku."
[master (root-commit) dbc1d99] Listo para el despliegue en heroku.
 43 ficheros modificados, 746
 inserciones(+) crear modo 100644
 .gitignore crear modo 100644
 Procfile
 --snip--
 crear modo 100644 users/views.py
❹ (ll_env)learning_log$ git status
# On branch master
nada que confirmar, directorio de trabajo
limpio (ll_env)learning_log$
```

En ❶ ejecutamos el comando `git init` para inicializar un repositorio vacío en el directorio que contiene Learning Log. En ❷ usamos el comando `git add .` que añade todos los archivos que no están siendo ignorados al repositorio. (No olvides el punto.) En ❸ emitimos el comando `git commit -am` *mensaje*

de confirmación: la bandera `-a` le dice a Git que incluya todos los archivos cambiados en esta confirmación, y la bandera `-m` le dice a Git que grabe un mensaje de registro.

Emitir el comando `git status` ❹ indica que estamos en la rama *master* y que nuestro directorio de trabajo está *limpio*. Este es el estado que querrás ver cada vez que envíes tu proyecto a Heroku.

Envío a Heroku

Finalmente estamos listos para empujar el proyecto a Heroku. En una sesión de terminal activa, emite los siguientes comandos:

```
❶ (ll_env)learning_log$ heroku login
  Introduce tus credenciales de
  Heroku. Correo electrónico:
  eric@example.com
  Contraseña (se ocultará lo que se escriba):
  Conectado como eric@example.com
❷ (ll_env)learning_log$ heroku create
  Creando afternoon-meadow-2775... hecho, la pila es cedar-
  14 https://afternoon-meadow-2775.herokuapp.com/ |
      https://git.heroku.com/afternoon-meadow-2775.git
  Git remoto heroku añadido
❸ (ll_env)learning_log$ git push heroku master
  --snip--
  remoto: -----> Lanzando... hecho, v6
remoto:           https://afternoon-meadow-2775.herokuapp.com/desplegado en
  Heroku remoto: Verificando deploy.      done.
  A https://git.heroku.com/afternoon-meadow-2775.git
     bdb2a35..62d711d master -> master
  (ll_env)learning_log$
```

En primer lugar, inicia sesión en Heroku en la sesión de terminal con el nombre de usuario y la contraseña que utilizaste para crear una cuenta en *https://heroku.com/* ❶. A continuación, dile a Heroku que construya un proyecto vacío ❷. Heroku genera un nombre compuesto por dos palabras y un número; puedes cambiarlo más adelante. A continuación, emitimos el comando `git push heroku master` ❸, que le dice a Git que empuje la rama maestra del proyecto al repositorio que Heroku acaba de crear. Heroku entonces construye el proyecto en sus servidores usando estos archivos. En ❹ está la URL que usaremos para acceder al proyecto en vivo.

Cuando hayas emitido estos comandos, el proyecto estará desplegado pero no completamente configurado. Para comprobar que el proceso del servidor se ha iniciado correctamente, utiliza el comando `heroku ps`:

```
  (ll_env)learning_log$ heroku ps
❶ Cuota libre restante: 17h 40m
❷ === web (Free): `gunicorn learning_log.wsgi log-file -`
  web.1: up 2015/08/14 07:08:51 (~ 10m ago)
  (ll_env)learning_log$
```

El resultado muestra cuánto tiempo más puede estar activo el proyecto en las próximas 24 horas ❶. En el momento de escribir esto, Heroku permite

que los despliegues gratuitos estén activos hasta 18 horas en cualquier periodo de 24 horas. Si un proyecto

excede estos límites, se mostrará una página de error estándar del servidor; personalizaremos esta página de error en breve. En ❷ vemos que se ha iniciado el proceso definido en *Procfile*.

Ahora podemos abrir la aplicación en un navegador usando el comando `heroku open`:

```
(ll_env)learning_log$ heroku open
Apertura tarde-pradera-2775... hecho
```

Este comando te ahorra abrir un navegador e introducir la URL que Heroku te mostró, pero esa es otra forma de abrir el sitio. Deberías ver la página de inicio de Learning Log, con el estilo correcto. Sin embargo, aún no puedes usar la aplicación porque no hemos configurado la base de datos.

NOTE *El proceso de despliegue de Heroku cambia de vez en cuando. Si tienes algún problema que no puedes resolver, busca ayuda en la documentación de Heroku. Ir a* https://devcenter.heroku.com/*, haz clic en* ***Python****, y busca un enlace a* Getting Started with Django*. Si no entiendes lo que ves ahí, consulta las sugerencias del Apéndice C.*

Configuración de la base de datos en Heroku

Necesitamos ejecutar `migrate` una vez para configurar la base de datos en vivo y aplicar todas las migraciones que generamos durante el desarrollo. Puedes ejecutar comandos Django y Python en un proyecto Heroku usando el comando `heroku run`. He aquí cómo ejecutar `migrate` en el despliegue Heroku:

```
❶ (ll_env)learning_log$ heroku run python manage.py migrate
❷ Ejecutando `python manage.py migrate` en afternoon-meadow-2775... up, run.2435
    --snip--
❸ Ejecución de migraciones:
    --snip--
    Aplicando learning_logs.0001_initial... OK
    Aplicando learning_logs.0002_entry...
    Aceptar Aplicando
    learning_logs.0003_topic_user... Aceptar
    Aplicando sessions.0001_initial... OK
(ll_env)learning_log$
```

Primero ejecutamos el comando `heroku run python manage.py migrate` ❶. Heroku crea entonces una sesión de terminal para ejecutar el comando `migrate` ❷. En ❸ Django aplica las migraciones por defecto y las migraciones que generamos durante el desarrollo de Learning Log.

Ahora, cuando visites tu aplicación desplegada, deberías poder utilizarla igual que lo hacías en tu sistema local. Sin embargo, no verás ninguno de los datos que has introducido en tu despliegue local, porque no hemos copiado los datos al servidor en vivo. Esta es una práctica normal: normalmente no se copian los datos locales a

un despliegue en vivo porque los datos locales suelen ser datos de prueba.

Puedes compartir tu enlace Heroku para que cualquiera pueda utilizar tu versión de Learning Log. En la siguiente sección completaremos algunas tareas más para finalizar el proceso de despliegue y prepararte para continuar desarrollando Learning Log.

Perfeccionamiento de la implantación de Heroku

En esta sección refinaremos el despliegue creando un superusuario, tal y como hicimos localmente. También haremos el proyecto más seguro cambiando la configuración `DEBUG` a `False`, para que los usuarios no vean ninguna información extra en los mensajes de error que puedan usar para atacar al servidor.

Creación de un superusuario en Heroku

Ya has visto que podemos ejecutar comandos puntuales usando el comando `heroku run`. Pero también puedes ejecutar comandos abriendo una sesión de terminal Bash mientras estás conectado al servidor Heroku usando el comando `heroku run bash`. *Bash* es el lenguaje que se ejecuta en muchos terminales Linux. Vamos a utilizar la sesión de terminal Bash para crear un superusuario para que podamos acceder al sitio de administración en la aplicación en vivo:

```
(ll_env)learning_log$ heroku run bash
Ejecutando `bash` en afternoon-meadow-2775... up, run.6244
❶ ~ $ ls
learning_log learning_logs manage.py Procfile requirements.txt runtime.txt
users
archivos estáticos
❷ ~ $ python manage.py createsuperuser
Nombre de usuario (déjelo en blanco para utilizar "u41907"): ll_admin
Dirección de correo electrónico:
Contraseña:
Contraseña (otra vez):
Superusuario creado con éxito.
❸ ~ $ exit
exit
(ll_env)learning_log$
```

En ❶ ejecutamos `ls` para ver qué archivos y directorios existen en el servidor, que deberían ser los mismos que tenemos en nuestro sistema local. Puedes navegar por este sistema de archivos como por cualquier otro.

NOTE *Los usuarios de Windows utilizarán los mismos comandos que se muestran aquí (como `ls` en lugar de `dir`), porque estás ejecutando un terminal Linux a través de una conexión remota.*

En ❷ ejecutamos el comando para crear un superusuario, que muestra las mismas indicaciones que vimos en nuestro sistema local cuando creamos un superusuario en el Capítulo 18. Cuando hayas terminado de crear el superusuario en esta sesión de terminal, utiliza el comando `exit` para volver a la sesión de terminal ❸ de tu sistema local.

Ahora puedes añadir */admin/ al* final de la URL de la aplicación en vivo e iniciar sesión en el sitio de administración. Para mí, la URL es *https://afternoon-meadow-2775*

.herokuapp.com/admin/.

Si otras personas ya han empezado a utilizar tu proyecto, ten en cuenta que tendrás acceso a todos sus datos. No te lo tomes a la ligera, y los usuarios seguirán confiándote sus datos.

Creación de una URL fácil de usar en Heroku

Probablemente querrás que tu URL sea más amigable y fácil de recordar que https://afternoon-meadow-2775.herokuapp.com/. Puedes cambiar el nombre de la aplicación con un solo comando:

```
(ll_env)learning_log$ heroku apps:rename learning-log
Renombrar afternoon-meadow-2775 a learning-log... hecho
https://learning-log.herokuapp.com/ | https://git.heroku.com/learning-log.git
Git remoto heroku actualizado
(ll_env)learning_log$
```

Puedes utilizar letras, números y guiones para nombrar tu aplicación, y llamarla como quieras, siempre que nadie más haya reclamado el nombre. Este despliegue se encuentra ahora en https://learning-log.herokuapp.com/. El proyecto ya no está disponible en la URL anterior; el comando `apps:rename` mueve completamente el proyecto a la nueva URL.

NOTE *Cuando despliegas tu proyecto utilizando el servicio gratuito de Heroku, Heroku pone tu despliegue en reposo si no ha recibido ninguna solicitud después de un cierto tiempo o si ha sido demasiado activo para el nivel gratuito. La primera vez que un usuario acceda al sitio después de que haya estado en reposo, tardará más en cargarse, pero el servidor responderá a las solicitudes posteriores con mayor rapidez. Así es como Heroku puede permitirse ofrecer despliegues gratuitos.*

Asegurar el proyecto en directo

Existe un problema de seguridad evidente en la forma en que nuestro proyecto se despliega actualmente: la configuración `DEBUG=True` en *settings.py*, que proporciona mensajes de depuración cuando se producen errores. Las páginas de error de Django te dan información vital de depuración cuando estás desarrollando un proyecto, pero dan demasiada información a los atacantes si las dejas activadas en un servidor activo. También tenemos que asegurarnos de que nadie pueda obtener información o redirigir peticiones haciéndose pasar por el host del proyecto.

Vamos a modificar *settings.py para que* podamos ver los mensajes de error localmente pero no en el despliegue en vivo:

settings.py

```
--snip--
# Configuración de Heroku
si os.getcwd() == '/app':
    --snip--
    # Respetar la cabecera 'X-Forwarded-Proto' para request.is_secure().
    SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

 # Permitir sólo Heroku para alojar el proyecto.
1    ALLOWED_HOSTS = ['learning-log.herokuapp.com']

2    DEBUG = Falso

    # Static asset configuration
    --snip--
```

Solo necesitamos hacer dos cambios: en ❶ modificamos `ALLOWED_HOSTS`, para que el único servidor permitido para alojar el proyecto sea `Heroku`. Necesitas usar el nombre de tu app, ya sea el nombre que Heroku te proporcionó, como por ejemplo *afternoon-meadow-2775.herokuapp.com*, o el nombre que hayas elegido. En ❷ ponemos `DEBUG` a `False`, para que Django no comparta información sensible cuando ocurra un error.

Comprometiendo y empujando cambios

Ahora tenemos que confirmar los cambios realizados en *settings.py* en el repositorio Git, y luego empujar los cambios a Heroku. Aquí hay una sesión de terminal que muestra este proceso:

```
❶ (ll_env)learning_log$ git commit -am "Set DEBUG=False for Heroku."
[master 081f635] Establecer DEBUG=False para Heroku.
 1 fichero modificado, 4 inserciones(+), 2 supresiones(-)
❷ (ll_env)learning_log$ git status
# On branch master
nada que confirmar, directorio de trabajo
limpio (ll_env)learning_log$
```

Emitimos el comando `git commit` con un mensaje de commit corto pero descriptivo ❶. Recuerda que la bandera `-am` se asegura de que Git confirma todos los archivos que han cambiado y registra el mensaje de registro. Git reconoce que un archivo ha cambiado y confirma este cambio en el repositorio.

En ❷ el estado muestra que estamos trabajando en la rama maestra del repositorio y que ahora no hay nuevos cambios que confirmar. Es esencial que compruebe el estado de este mensaje antes de empujar a Heroku. Si no ves este mensaje, algunos cambios no han sido confirmados, y esos cambios no serán enviados al servidor. Puedes intentar enviar el `commit` command de nuevo, pero si no estás seguro de cómo resolver el problema, lee el Apéndice D para entender mejor cómo trabajar con Git.

Ahora vamos a empujar el repositorio actualizado a Heroku:

```
(ll_env)learning_log$ git push heroku master
--snip--
remoto: -----> Aplicación Python detectada
remoto: -----> Instalación de dependencias con pip
--snip--
remoto: -----> Lanzando... hecho, v8
remote         :https://learning-log.herokuapp.com/ desplegado en
Heroku remote: Verificando deploy.      done.
A https://git.heroku.com/learning-log.git
   4c9d111..ef65d2b maestro -> maestro
(ll_env)learning_log$
```

Heroku reconoce que el repositorio ha sido actualizado, y reconstruye el proyecto para asegurarse de que todos los cambios se han tenido en cuenta. No reconstruye la base de datos, por lo que no tendremos que ejecutar `migrate` para esta actualización.

Para comprobar que el despliegue es ahora más seguro, introduce la URL de tu proyecto con una extensión que no hayamos definido. Por ejemplo, intenta visitar http://learning-log.herokuapp.com/letmein/. Debería ver una página de error genérica en su despliegue en vivo que no proporciona ninguna información específica sobre el proyecto. Si intentas la misma petición en la versión local
de Learning Log en http://localhost:8000/letmein/, deberías ver la página de error completa de Django. El resultado es perfecto: verás mensajes de error informativos cuando sigas desarrollando el proyecto, pero los usuarios no verán información crítica sobre el código del p r o y e c t o .

Creación de páginas de error personalizadas

En el Capítulo 19, configuramos el Registro de aprendizaje para que devuelva un error 404 si el usuario solicita un tema o una entrada que no le pertenece. Probablemente también hayas visto algunos errores 500 del servidor (errores internos) en este punto. Un error 404 normalmente significa que tu código Django es correcto, pero el objeto solicitado no existe; un error 500 normalmente significa que hay un error en el código que has escrito, como un error en una función en *views.py*. Actualmente, Django devuelve la misma página de error genérica en ambas situaciones, pero podemos escribir nuestras propias plantillas de páginas de error 404 y 500 que coincidan con la apariencia general de Learning Log. Estas plantillas deben ir en el directorio raíz de plantillas.

Creación de plantillas personalizadas

En la carpeta *learning_log/learning_log*, cree una nueva carpeta llamada *templates*. A continuación, cree un nuevo archivo llamado *404.html* con el siguiente código:

404.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% encabezado de bloque %}
    <h2>El artículo que ha solicitado no está disponible. (404)</h2>
    {% endblock header %}
```

Esta sencilla plantilla proporciona la información genérica de la página de error 404, pero con un estilo que hace juego con el resto del sitio.

Crea otro archivo llamado *500.html* usando el siguiente código:

500.html

```
{% extends "registros_de_aprendizaje/base.html" %}

{% encabezado de bloque %}
    <h2>Ha habido un error interno. (500)</h2>
{% endblock header %}
```

Estos nuevos archivos requieren un ligero cambio en *settings.py*.

settings.py

```
--snip--
PLANTILLAS = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'learning_log/templates')],
```

```
        'APP_DIRS': True,
        --snip--
    },
]
--snip--
```

Este cambio indica a Django que busque las plantillas de las páginas de error en el directorio raíz de plantillas.

Visualización local de las páginas de error

Si quieres ver cómo se ven las páginas de error en tu sistema antes de enviarlas a Heroku, primero tendrás que establecer `Debug=False` en tu configuración local para suprimir las páginas de depuración predeterminadas de Django. Para ello, realice los siguientes cambios en *settings.py* (asegúrese de que está trabajando en la parte de *settings.py* que se aplica al entorno local, no la parte que se aplica a Heroku):

settings.py

```
--snip--
# ADVERTENCIA DE SEGURIDAD: ¡no ejecutar con debug activado en
producción! DEBUG = Falso

ALLOWED_HOSTS = ['localhost']
--snip--
```

Debe tener al menos un host especificado en `ALLOWED_HOSTS` cuando `DEBUG` está en `False`. Ahora solicite un tema o entrada que no le pertenezca para ver la página de error 404, y solicite una URL que no exista (como *localhost:8000/letmein/*) para ver la página de error 500.

Cuando haya terminado de comprobar las páginas de error, vuelva a establecer `DEBUG` en `True` para seguir desarrollando Learning Log. (Asegúrate de que `DEBUG` sigue establecido en `False` en la sección de *settings.py* que se aplica a la implementación de Heroku).

NOTE *La página de error 500 no mostrará ninguna información sobre el usuario que ha iniciado sesión, porque Django no envía ninguna información de contexto en la respuesta cuando hay un error de servidor.*

Enviar los cambios a Heroku

Ahora tenemos que confirmar los cambios de plantilla y empujarlos en vivo a Heroku:

```
❶ (ll_env)learning_log$ git add .
❷ (ll_env)learning_log$ git commit -am "Añadidas páginas de error 404 y 500
personalizadas."
   3 ficheros cambiados, 15 inserciones(+), 10
   eliminaciones(-)      crear      modo      100644
   learning_log/templates/404.html crear modo 100644
   learning_log/templates/500.html
❸ (ll_env)learning_log$ git push heroku master
   --snip--
```

```
remoto: Verificando despliegue.  Hecho.
```

```
A https://git.heroku.com/learning-log.git
   2b34ca1..a64d8d3 maestro -> maestro
(ll_env)learning_log$
```

Emitimos el comando `git add .` en ❶ porque hemos creado algunos archivos nuevos en el proyecto, por lo que necesitamos decirle a Git que comience a rastrear estos archivos. Luego confirmamos los cambios ❷ y empujamos el proyecto actualizado a Heroku ❸.

Ahora, cuando aparezca una página de error, tendrá el mismo estilo que el resto del sitio, lo que facilitará la experiencia del usuario cuando se produzcan errores.

Uso del método get_object_or_404()

En este punto, si un usuario solicita manualmente un tema o entrada que no existe, obtendrá un error de servidor 500. Django intenta renderizar la página pero no tiene suficiente información para hacerlo, y el resultado es un error 500. Esta situación se maneja mejor como un `error 404`, y podemos implementar este comportamiento con la función de acceso directo de Django `get_object_or_404()`. Esta función intenta obtener el objeto solicitado de la base de datos, pero si ese objeto no existe, lanza una excepción 404. Importaremos esta función a *views.py* y la usaremos en lugar de `get()`:

views.py

```
--snip--
from django.shortcuts import render, get_object_or_404
from django.http import HttpResponseRedirect, Http404
--snip--@login_required
def tema(solicitud, topic_id):
    """Mostrar un único tema y todas sus entradas."""
    topic = get_object_or_404(Topic, id=topic_id)
    # Asegúrate de que el tema pertenece al usuario actual.
    --snip--
```

Ahora, cuando solicites un tema que no existe (por ejemplo, *http://localhost:8000/topics/999999/*), verás una página de error 404. Para desplegar este cambio, haz un nuevo commit, y luego empuja el proyecto a Heroku.

Desarrollo continuo

Es posible que desee desarrollar más Learning Log después de su empuje inicial a un servidor en vivo o desarrollar sus propios proyectos para desplegar. Existe un proceso bastante coherente para actualizar los proyectos.

En primer lugar, realizarás los cambios necesarios en tu proyecto local. Si tus cambios resultan en nuevos archivos, añádelos al repositorio Git usando el comando `git add .` (asegúrate de incluir el punto al final del comando). Cualquier cambio que requiera una migración de la base de datos necesitará este comando, porque cada migración genera un nuevo archivo de migración.

A continuación, confirma los cambios en tu repositorio usando `git`

`commit -am "`*commit message*`"`. Después, empuja tus cambios a Heroku usando el comando `git push heroku master`. Si migraste tu base de datos localmente, necesitarás migrar la base de datos en vivo también. Puedes usar el comando

comando `heroku run python manage.py migrate`, o abre una sesión de terminal remota con `heroku run bash` y ejecuta el comando `python manage.py migrate`. A continuación, visita tu proyecto en vivo, y asegúrate de que los cambios que esperas ver han surtido efecto.

Es fácil cometer errores durante este proceso, así que no te sorprendas si algo sale mal. Si el código no funciona, revisa lo que has hecho e intenta detectar el error. Si no encuentras el error o no sabes cómo deshacerlo, consulta las sugerencias para obtener ayuda en el Apéndice C. No seas tímido a la hora de pedir ayuda: todo el mundo aprendió para construir proyectos haciendo las mismas preguntas que probablemente te harás tú, así que alguien estará encantado de ayudarte. Resolver cada problema que surja te ayudará a desarrollar tus habilidades hasta que construyas proyectos significativos y fiables y respondas también a las preguntas de los demás.

Configuración de SECRET_KEY

Django utiliza el valor de la configuración `SECRET_KEY` en *settings.py* para implementar una serie de protocolos de seguridad. En este proyecto, hemos enviado nuestro archivo de configuración al repositorio con la configuración `SECRET_KEY` incluida. Esto está bien para un proyecto de práctica, pero la configuración `SECRET_KEY` debe ser manejada con más cuidado para un sitio de producción. Si usted construye un proyecto que está recibiendo un uso significativo, asegúrese de investigar cómo manejar su configuración `SECRET_KEY` de forma más segura.

Eliminar un proyecto en Heroku

Es una buena práctica realizar el proceso de despliegue varias veces con el mismo proyecto o con una serie de proyectos pequeños para familiarizarse con el despliegue. Pero necesitarás saber cómo borrar un proyecto que ha sido desplegado. Heroku también puede limitar el número de proyectos que puedes alojar de forma gratuita, y no querrás saturar tu cuenta con proyectos de práctica.

Accede al sitio web de Heroku (https://heroku.com/), y serás redirigido a una página que muestra una lista de tus proyectos. Haga clic en el proyecto que desea eliminar, y verá una nueva página con información sobre el proyecto. Haz clic en el enlace **Configuración** y desplázate hacia abajo hasta que veas un enlace para eliminar el proyecto. Esta acción no se puede revertir, por lo que Heroku te pedirá que confirmes la solicitud de eliminación introduciendo manualmente el nombre del proyecto.

Si prefiere trabajar desde un terminal, también puede eliminar un proyecto mediante el comando `destroy`:

```
(ll_env)learning_log$ heroku apps:destroy --app appname
```

Aquí *appname* es el nombre de tu proyecto, que puede ser algo como `afternoon-meadow-2775` o `learning-log` si has renombrado el proyecto. Se te pedirá que vuelvas a introducir el nombre del proyecto para confirmar la eliminación.

NOTE *Borrar un proyecto en Heroku no hace nada a tu versión local del proyecto. Si nadie ha utilizado tu proyecto desplegado y sólo estás practicando el proceso de despliegue, es perfectamente razonable eliminar tu proyecto en Heroku y volver a desplegarlo.*

TRY IT yOURSELF

20-3. Blog en vivo: Despliega el proyecto Blog en el que has estado trabajando en Heroku. Asegúrate de poner `DEBUG` en `False` y cambia la configuración de `ALLOWED_HOSTS`, para que tu despliegue sea razonablemente seguro.

20-4. Más 404s: La función `get_object_or_404()` debería usarse también en las vistas `new_entry()` y `edit_entry()`. Haga este cambio, pruébelo introduciendo una URL como http://localhost:8000/new_entry/99999/, y compruebe que ve un error 404.

20-5. Registro de aprendizaje ampliado: Añada una función al Registro de Aprendizaje e introduzca el cambio en su despliegue activo. Pruebe con un cambio sencillo, como escribir
más información sobre el proyecto en la página de inicio. A continuación, intente añadir una función más avanzada, como dar a los usuarios la opción de hacer público un tema. Esto requeriría un atributo llamado `public` como parte del modelo de `temas` (debería estar en `False` por defecto) y un elemento de formulario en la página new_topic que permita al usuario cambiar un tema de privado a público. A continuación, tendrá que migrar el proyecto y revisar *views.py* para que cualquier tema que sea público también sea visible para los usuarios no autorizados. Recuerde que debe migrar la base de datos en vivo después de haber empujado sus cambios a Heroku.

resumen

En este capítulo aprendiste a dar a tus proyectos una apariencia simple pero profesional usando la librería Bootstrap y la aplicación django-bootstrap3. Usar Bootstrap significa que los estilos que elijas funcionarán consistentemente en casi cualquier dispositivo que la gente use para acceder a tu proyecto.

Conociste las plantillas de Bootstrap y utilizamos la plantilla *Static top navbar* para crear un aspecto sencillo para Learning Log. Has aprendido a utilizar un jumbotrón para destacar el mensaje de una página de inicio y a aplicar un estilo coherente a todas las páginas de un sitio.

En la parte final del proyecto, aprendiste a desplegar un proyecto en los servidores de Heroku para que cualquiera pueda acceder a él. Creaste una cuenta Heroku e instalaste algunas herramientas que ayudan a gestionar el proceso de despliegue. Utilizaste Git para confirmar el proyecto de trabajo en un repositorio y luego empujaste el repositorio a los servidores de Heroku. Finalmente, aprendiste a proteger tu aplicación configurando `DEBUG=False` en el servidor activo.

Ahora que ha terminado el Registro de aprendizaje, puede empezar a construir sus propios proyectos. Empieza de forma sencilla y asegúrate de que el proyecto funciona antes de añadirle complejidad. Disfruta de tu aprendizaje y ¡buena suerte con tus proyectos!

AFTERWORD



¡Enhorabuena! Has aprendido los conceptos básicos de Python y has aplicado tus conocimientos a proyectos significativos. Has creado un juego, visualizar algunos datos y crear una aplicación web. A partir de aquí, puedes ir en diferentes direcciones para continuar desarrollando tus habilidades de programación.

En primer lugar, debes seguir trabajando en proyectos significativos que te interesen. La programación es más atractiva cuando se resuelven problemas relevantes y significativos, y ahora tienes las habilidades para participar en una variedad de proyectos. Podrías inventar tu propio juego o escribir tu propia versión de un juego arcade clásico. Podrías explorar algunos datos importantes para ti y hacer visualizaciones que muestren patrones y conexiones interesantes. Podrías crear tu propia aplicación web o intentar emular una de tus aplicaciones favoritas.

Siempre que sea posible, invita a otras personas a probar tus programas. Si escribes un juego, deja que otros lo jueguen. Si haces una visualización, enséñasela a otros y comprueba si tiene sentido para ellos. Si creas una aplicación web, despliégala en línea e invita a otros a probarla. Escucha a tus usuarios y prueba
para incorporar sus comentarios a tus proyectos; te convertirás en un mejor pro- gramador si lo haces.

Cuando trabajes en tus propios proyectos, te encontrarás con problemas que son difíciles, o incluso imposibles, de resolver por ti mismo. Sigue buscando formas de pedir ayuda y encuentra tu propio lugar en la comunidad Python. Únete a un grupo local de usuarios de Python o explora algunas comunidades de Python en línea. Considera también la posibilidad de asistir a una PyCon cerca de ti.

Debes esforzarte por mantener un equilibrio entre trabajar en proyectos que te interesen y desarrollar tus habilidades con Python en general. Muchas fuentes de aprendizaje de Python están disponibles en línea, y un gran número de libros de Python están dirigidos a programadores intermedios. Muchos de estos recursos serán accesibles para ti ahora que sabes lo básico y cómo aplicar tus habilidades. Trabajar con tutoriales y libros de Python se basará directamente en lo que aprendiste aquí y profundizará tu comprensión de la programación en general y de Python en particular. Entonces, cuando vuelvas a trabajar en proyectos después de centrarte en el aprendizaje de Python, serás capaz de resolver una mayor variedad de problemas de forma más eficiente.

Enhorabuena por lo lejos que has llegado, y ¡suerte con tu aprendizaje continuo!

A

INSTALLING PyTHON



Python tiene varias versiones diferentes y varias formas de configurarlo en cada sistema operativo. Este apéndice es útil si el enfoque del Capítulo 1 no funcionó, o si quieres instalar una versión de Python diferente a la que vino con tu sistema.

Python en Linux

Python se incluye por defecto en casi todos los sistemas Linux, pero es posible que desee utilizar una versión diferente a la predeterminada. Si es así, primero averigua qué versión de Python tienes ya instalada.

Encontrar la versión instalada

Abra una ventana de terminal y ejecute el siguiente comando:

```
$ python --version
Python 2.7.6
```

El resultado muestra que la versión por defecto es 2.7.6. Sin embargo, es posible que también tenga instalada una versión de Python 3. Para comprobarlo, introduzca el siguiente comando:

```
$ python3 --version
Python 3.5.0
```

Python 3.5.0 también está instalado. Merece la pena ejecutar ambos comandos antes de intentar instalar una nueva versión.

Instalación de Python 3 en Linux

Si no tienes Python 3 , o si quieres instalar una versión más reciente de Python 3, puedes instalarla en unas pocas líneas. Usaremos un paquete llamado `deadsnakes`, que facilita la instalación de múltiples versiones de Python:

```
$ sudo add-apt-repository ppa:fkrull/deadsnakes
$ sudo apt-get update
$ sudo apt-get install python3.5
```

Estos comandos instalarán Python 3.5 en tu sistema. El siguiente código iniciará una sesión de terminal ejecutando Python 3.5:

```
$ python3.5
>>>
```

También querrás usar este comando cuando configures tu editor de texto para usar Python 3 y cuando ejecutes programas desde la terminal.

Python en OS X

Python ya está instalado en la mayoría de los sistemas OS X, pero es posible que desee utilizar una versión diferente a la predeterminada. Si es así, primero averigua qué versión de Python tienes ya instalada.

Encontrar la versión instalada

Abra una ventana de terminal e introduzca el siguiente comando:

```
$ python --version
Python 2.7.6
```

También deberías probar a ejecutar el comando **python3 --version**. Probablemente obtendrá un mensaje de error, pero vale la pena comprobar si la versión que desea ya está instalada.

Uso de Homebrew para instalar Python 3

Si sólo tienes Python 2 instalado, o si tienes una versión antigua de Python 3, puedes instalar la última versión de Python 3 utilizando un paquete llamado Homebrew.

Instalación de Homebrew

Homebrew depende del paquete Xcode de Apple, así que abre un terminal y ejecuta este comando:

```
$ xcode-select --install
```

Haz clic en los cuadros de diálogo de confirmación que aparecen (esto puede tardar un poco, dependiendo de la velocidad de tu conexión). A continuación, instala Homebrew:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
master/install)"
```

Puedes encontrar este comando en la página principal del sitio Homebrew en http://brew.sh/. Asegúrate de incluir un espacio entre `curl -fsSL` y la URL.

NOTE *El `-e` de este comando indica a Ruby (el lenguaje de programación en el que está escrito Homebrew) que ejecute el código que se descarga aquí. Sólo deberías ejecutar comandos como éste desde fuentes en las que confíes.*

Para confirmar que Homebrew se ha instalado correctamente, ejecute este comando:

```
$ brew doctor
Su sistema está listo para elaborar cerveza.
```

Esta salida significa que estás listo para instalar paquetes Python a través de Homebrew.

Instalación de Python 3

Para instalar la última versión de Python 3, introduzca el siguiente comando:

```
$ brew install python3
```

Comprobemos qué versión se instaló utilizando este comando:

```
$ python3 --version
Python 3.5.0
$
```

Ahora puedes iniciar una sesión de terminal de Python 3 usando el comando `python3`, y puedes usar el comando `python3` para configurar tu editor de texto para que ejecute programas Python con Python 3 en lugar de Python 2.

Python en Windows

Python no suele estar incluido por defecto en Windows, pero vale la pena comprobar si existe en el sistema. Abra una ventana de terminal haciendo clic con el botón derecho del ratón en el escritorio mientras mantiene pulsada la tecla MAYÚS y, a continuación, seleccione **Abrir ventana de comandos aquí**. También puedes introducir **comando** en el menú Inicio. En la ventana de terminal que aparece, ejecute el siguiente comando:

```
> python --version
Python 3.5.0
```

Si ves una salida como ésta, Python ya está instalado, pero aún así puede que quieras instalar una versión más reciente. Si ves un mensaje de error, tendrás que descargar e instalar Python.

Instalación de Python 3 en Windows

Ve a *http://python.org/downloads/* y haz clic en la versión de Python que quieras. Descarga el instalador, y cuando lo ejecutes asegúrate de marcar la opción *Add Python to PATH*. Esto te permitirá utilizar el comando `python` en lugar de tener que introducir la ruta completa de tu sistema a `python`, y no tendrás que modificar manualmente las variables de entorno de tu sistema. Una vez que hayas instalado Python, ejecuta el comando **python --version en una** nueva ventana de terminal. Si funciona, ya está.

Cómo encontrar el intérprete de Python

Si el simple comando `python` no funciona, tendrás que decirle a Windows dónde encontrar el intérprete de Python. Para encontrarlo, abre la unidad C y busca la carpeta cuyo nombre empiece por *Python* (puede que tengas que introducir la palabra `python` en la barra de búsqueda del Explorador de Windows para encontrar la carpeta correcta). Abra la carpeta y busque un archivo con el nombre *python* en minúsculas. Haga clic con el botón derecho en este archivo y seleccione **Propiedades**; verá la ruta a este archivo bajo el título *Ubicación*.

En la ventana del terminal, utilice la ruta para confirmar la versión que acaba de instalar:

```
C:Python35python --version
Python 3.5.0
```

Añadir Python a su variable de ruta

Es molesto teclear la ruta completa cada vez que quieres iniciar un ter- minal de Python, así que añadiremos la ruta al sistema para que sólo tengas que usar el comando `python`. Si ya ha marcado la casilla *Añadir Python al PATH* durante la instalación, puede omitir este paso. Abra el **Panel de control de** su sistema, seleccione **Sistema y seguridad** y, a continuación, seleccione **Sistema**. Haga clic en **Configuración avanzada del sistema**. En la ventana que aparece, haz clic en **Variables de entorno**.

En la casilla *Variables del sistema*, busque una variable llamada `Ruta`. Haz clic en **Editar**. En el cuadro que aparece, haga clic en el cuadro *Valor de la variable* y utilice la tecla de flecha derecha para desplazarse hacia la derecha. Ten cuidado de no sobrescribir la variable existente; si lo haces, haz clic en Cancelar y vuelve a intentarlo. Añade un punto y coma y la ruta a tu archivo *python.exe* a la variable existente:

```
%SystemRoot%\system32\...\System32\WindowsPowerShell\v1.0\;C:\Python34
```

Cierre su ventana de terminal y abra una nueva. Esto cargará la nueva variable `Path` en tu sesión de terminal. Ahora, cuando introduzcas `python --version`, deberías ver la versión de Python que acabas de establecer en tu `Path` vari- able. Ahora puede iniciar una sesión de terminal Python con sólo introducir `python` en un símbolo del sistema.

Palabras clave y funciones incorporadas de Python

Python viene con su propio conjunto de palabras clave y funciones incorporadas. Es importante tenerlas en cuenta a la hora de nombrar variables. Un desafío en programación es encontrar buenos nombres para las variables, que pueden ser cualquier cosa que sea razonablemente corta y descriptiva. Pero no puedes usar ninguna de las palabras clave de Python, y no deberías usar el nombre de ninguna de las funciones incorporadas de Python porque sobrescribirías las funciones.

En esta sección enumeraremos las palabras clave de Python y los nombres de las funciones incorporadas, para que sepas qué nombres debes evitar.

Palabras clave de Python

Cada una de las siguientes palabras clave tiene un significado específico, y verá un error si intenta utilizarlas como nombre de variable.

```
Falso      clase      finalmente  es         devolver
Ninguno    continuar  para        lambda     pruebe
Verdadero  def        de          no local   mientras que
y          del        global      no         con
como       elif       si          o          rendimiento
afirme     si no      importar    pase
```

romper	excepto	en	subir

Funciones incorporadas de Python

No obtendrá un error si utiliza una de las siguientes funciones incorporadas fácilmente disponibles como nombre de variable, pero anulará el comportamiento de esa función:

abs()	divmod()	entrada()	abrir()	métodoestático()
todo()	enumerar()	int()	ord()	str()
cualquiera()	eval()	isinstance()	pow()	suma()
basestring()	execfile()	issubclass()	imprimir()	super()
bin()	archivo()	iter()	propiedad()	tupla()
bool()	filtrar()	len()	rango()	tipo()
bytearray()	float()	lista()	raw_input()	unichr()
invocable()	format()	locales()	reducir()	unicode()
chr()	frozenset()	largo()	recargar()	vars()
método de clase()	getattr()	mapa()	repr()	xrange()
cmp()	globales()	max()	invertido()	zip()
compilar()	hasattr()	memoryview()	redondo()	importar ()
complejo()	hash()	min()	set()	aplicar()
delattr()	ayuda()	next()	setattr()	búfer()
dict()	hex()	objeto()	cortar()	coaccionar()
dir()	id()	oct()	ordenado()	interno()

NOTE *En Python 2.7* print *es una palabra clave, no una función. Además,* unicode() *no está disponible en Python 3. Ninguna de estas palabras debe usarse como nombre de variable.*

B

TExTEDITOS



Los programadores dedican mucho tiempo a escribir, leer y editar código, y utilizan un editor de texto que hace que este trabajo sea lo más eficiente posible.

ciente como sea posible. Un editor eficaz debe resaltar la estructura del código para que pueda detectar errores comunes mientras trabaja. También debe incluir sangría automática, marcadores para mostrar la longitud de línea adecuada y atajos de teclado para operaciones comunes.

Como nuevo programador, deberías usar un editor que tenga estas características pero que no tenga una curva de aprendizaje muy pronunciada. También es bueno que conozcas un poco los editores más avanzados para saber cuándo debes plantearte actualizarte.

Veremos un editor de calidad para cada uno de los principales sistemas operativos: Geany para principiantes que trabajen en Linux o Windows, y Sublime Text para OS X (aunque también funciona bien en Linux y Windows). También veremos IDLE, el editor que viene con Python por

defecto. Por último, veremos

Emacs y vim, dos editores avanzados que oirás mencionar con frecuencia a medida que pases más tiempo programando. Utilizaremos *hello_world.py* como programa de ejemplo para ejecutar en cada editor.

Geany

Geany es un sencillo editor de texto que te permite ejecutar casi todos tus programas directamente desde el editor. También muestra tu salida en una ventana de terminal, lo que te ayuda a familiarizarte con el uso de terminales.

Instalación de Geany en Linux

Puede instalar Geany usando una línea en la mayoría de los sistemas Linux:

```
$ sudo apt-get install geany
```

Si tienes varias versiones de Python instaladas, tendrás que configurar Geany para que use la versión correcta. Abre Geany, selecciona □□□□□□□▸□□□□□□□ **como**, y guarda el archivo vacío como *hola_mundo.py*. Introduce la siguiente línea en el editor
ventana:

```
print("¡Hola mundo Python!")
```

Vaya a **Build▸Set Build Commands**. Deberías ver los campos Compilar y Ejecutar con un comando al lado de cada uno. Geany asume que `python` es el comando
comando correcto para cada uno de ellos, pero si su sistema utiliza el commandato `python3`, tendrá que cambiarlo. En **Compilar**, introduzca:

```
python3 -m py_compile "%f"
```

Asegúrese de que los espacios y las mayúsculas en su comando Compilar coinciden exactamente con lo que se muestra aquí.

Utilice este comando **Ejecutar**:

```
python3 "%f"
```

De nuevo, asegúrese de que el espaciado y las mayúsculas coinciden exactamente con lo que se muestra aquí.

Instalación de Geany en Windows

Puede descargar un instalador de Windows para Geany en *http://www .geany.org/* y haciendo clic en Releases en el menú Download. Ejecute el instalador llamado *Geany-1.25_setup.exe*, o algo similar, y acepte todos los valores predeterminados.

Abra Geany, seleccione **Archivo▸Guardar como**, y guarde el archivo vacío como

hello_world.py. Introduzca la siguiente línea en la ventana de edición:

```
print("¡Hola mundo Python!")
```

Ahora vaya a **Build▸Set Build Commands**. Debería ver los campos Compilar y Ejecutar con un comando junto a cada uno. Cada uno de estos comandos comienza con `python` (en minúsculas), pero Geany no sabe dónde almacenó tu sistema el comando `python`. Necesita agregar la ruta que usa cuando inicia una sesión de terminal. (Puede omitir estos pasos si configura la variante `Path` como se describe en el Apéndice A).

En los comandos Compilar y Ejecutar, añada la unidad en la que se encuentra su comando `python` y la carpeta en la que se almacena el comando `python`. Su comando **Compilar** debe tener este aspecto:

```
C:\Python35\python -m py_compile "%f"
```

Su ruta puede ser un poco diferente, pero asegúrese de que los espacios y la capi- talización coinciden exactamente con lo que se muestra aquí.

Su comando **Ejecutar** debería tener este aspecto:

```
C:\Python35\python "%f"
```

De nuevo, asegúrese de que el espaciado y las mayúsculas en su comando Ejecutar coinciden exactamente con lo que se muestra aquí. Cuando estas líneas estén correctamente configuradas, haga clic en **Aceptar**. Ahora debería ser capaz de ejecutar su programa con éxito.

Ejecutar programas Python en Geany

Hay tres formas de ejecutar un programa en Geany. Para ejecutar hello_world.*py*, seleccione **Build▸Execute** en el menú, o haga clic en el icono con un conjunto de engranajes, o presione F5. Cuando ejecutes hola_mundo.*py*, deberías ver una ventana de terminal con el siguiente resultado:

```
¡Hola mundo Python!

----------------
(el programa sale con el código:
0) Pulse Retorno para continuar.
```

Personalizar la configuración de Geany

Ahora configuraremos Geany para que sea lo más eficiente posible personalizando las características mencionadas al principio de este apéndice.

Convertir tabulaciones en espacios

Mezclar tabuladores y espacios en su código puede causar problemas en sus programas Python que son muy difíciles de diagnosticar. Para comprobar la configuración de la sangría en Geany, vaya a **Editar▸Preferencias▸Editor▸Sangría**. Configure el tabulador anchura a **4** y **Tipo** a **Espacios**.

Si tiene una mezcla de tabulaciones y espacios en uno de sus programas, puede convertir todas las tabulaciones en espacios con □□□□□□□□▸□□□□□□□ **Tabs by Spaces**.

Ajuste del indicador de longitud de línea

La mayoría de los editores le permiten configurar una señal visual, generalmente una línea vertical, para mostrar dónde deben terminar sus líneas. Configure esta función seleccionando **Editar▸Preferencias▸Editor▸Visualización**, y asegúrese de que **Línea larga** está activado. A continuación, asegúrese de que el valor de Columna se establece en 79.

Bloques de código con sangría y sin sangría

Para sangrar un bloque de código, resalte el código y vaya a □□□□□□▸□□□□□□□▸ **Aumentar sangría**, o presione CTRL-I. Para quitar la sangría de un bloque de código, vaya a □□□□□□▸ **Formato**▸ □□□□□□□□□ **sangría, o** presione CTRL-U.

Comentario de bloques de código

Para desactivar temporalmente un bloque de código, puedes resaltar el bloque y comentarlo para que Python lo ignore. Ve a □□□□▸□□□□□□▸□□□□□□ **Line Commentation** (CTRL-E). La línea será comentada con un (#~) para indicar que no es un comentario normal. Cuando quieras descomentar el bloque de código, resáltalo y vuelve a ejecutar el mismo comando.

Texto Sublime

Sublime Text es un sencillo editor de texto fácil de instalar en OS X (y también en otros sistemas), que te permite ejecutar casi todos tus programas directamente desde el editor. También ejecuta tu código en una sesión de terminal incrustada en la ventana de Sublime Text, lo que facilita ver la salida de tu código.

Sublime Text tiene una política de licencias muy liberal: puedes usar el editor gratuitamente todo el tiempo que quieras, pero el autor te pide que compres una licencia si te gusta y quieres seguir usándolo. Descargaremos Sublime Text 3, la versión más reciente en el momento de escribir este artículo.

Instalación de Sublime Text en OS X

Descarga el instalador de Sublime Text de *http://www.sublimetext. com/3*. Sigue el enlace de descarga y haz clic en el instalador para OS X. Cuando se haya descargado, abre el instalador y arrastra el icono de Sublime Text a la carpeta *Aplicaciones*.

Instalación de Sublime Text en Linux

En la mayoría de los sistemas Linux, lo más fácil es instalar Sublime Text desde una sesión de terminal, de la siguiente manera:

```
$ sudo add-apt-repository ppa:webupd8team/sublime-text-3
$ sudo apt-get update
$ sudo apt-get install sublime-text-installer
```

Instalación de Sublime Text en Windows

Descarga un instalador para Windows desde http://www.sublimetext.com/3. Ejecuta el instalador y deberías ver Sublime Text en tu menú Inicio.

Ejecutar programas Python en Sublime Text

Si estás usando la versión de Python que vino con tu sistema, probablemente podrás ejecutar tus programas sin ajustar ninguna configuración. Para ejecutar programas, ve a **Herramientas▸Construir** o presiona CTRL-B. Cuando ejecute *hello_world.py*, debería ver aparecer una pantalla de terminal en la parte inferior de la ventana de Sublime Text mostrando la siguiente salida:

```
¡Hola mundo Python!
[Terminado en 0.1s]
```

Configuración de Sublime Text

Si tienes múltiples versiones de Python instaladas o si Sublime Text no ejecuta programas Python automáticamente, tendrás que configurar un archivo de configuración. Primero, necesitarás saber la ruta completa a tu intérprete de Python. En Linux y OS X, ejecuta el siguiente comando:

```
$ type -a python3
python3 es /usr/local/bin/python3
```

Sustituya *python3* por el comando que utilice normalmente para iniciar una sesión de terminal.

Si utiliza Windows, consulte "Instalación de Python 3 en Windows" en la página 488 para encontrar la ruta a su intérprete de Python.

Ahora abre Sublime Text, y ve a **Tools▸Build System▸New Build System**, que abrirá un nuevo archivo de configuración para ti. Borre lo que ver, e introduzca lo siguiente:

Python3.sublime-build

```
{
    "cmd": ["/usr/local/bin/python3", "-u", "$archivo"],
}
```

Este código le dice a Sublime Text que use el comando `python3` cuando ejecute el archivo actualmente abierto. Asegúrate de usar la ruta que encontraste en el paso anterior (en Windows, tu ruta será algo así como `C:/Python35/ python`). Guarda el archivo como *Python3.sublime-build* en el directorio por defecto que Sublime Text abre cuando eliges Guardar.

Abre *hello_world.py*, selecciona □□□□□▸□□□□□ **System▸Python3**, y luego **Tools▸Build**. Deberías ver tu salida en un terminal incrustado en l a parte inferior de la ventana de Sublime Text.

Personalización de la configuración de Sublime Text

Ahora configuraremos Sublime Text para que sea lo más eficiente posible personalizando las características mencionadas al principio de este apéndice.

Convertir tabulaciones en espacios

Vaya a **Ver▸Indentación** y asegúrese de que hay una marca de verificación junto a Indentar usando espacios. Si no la hay, márquela.

Ajuste del indicador de longitud de línea

Ve a **Ver▸Regla** y haz clic en **80**. Sublime Text colocará una línea vertical en la marca de 80 caracteres.

Bloques de código con sangría y sin sangría

Para sangrar un bloque de código, resáltelo y seleccione **Editar▸Línea▸Sangrar** o pulse CTRL-]. Para quitar la sangría de un bloque de código, haz clic en **Editar▸Línea▸Sin sangría** o pulsa CTRL-[.

Comentario de bloques de código

Para comentar un bloque de código resaltado, seleccione □□□□□□▸ □□□□□□□□▸ **Alternar comentario**, o pulse CTRL-/. Para descomentar un bloque de código, emita el comando
mismo comando de nuevo.

IDLE

Idle es el editor por defecto de Python. Es un poco menos intuitivo para trabajar que Geany o Sublime Text, pero verás referencias a él en otros tutoriales dirigidos a principiantes, así que puede que quieras darle una oportunidad.

Instalación de IDLE en Linux

Si utilizas Python 3, instala el paquete `idle3` de la siguiente manera:

```
$ sudo apt-get install idle3
```

Si utilizas Python 2, instala el paquete `idle` de la siguiente manera:

```
$ sudo apt-get install idle
```

Instalación de IDLE en OS X

Si usaste Homebrew para instalar Python, IDLE probablemente ya esté en tu sistema. En un terminal, ejecuta el comando **brew linkapps**, que le dice a IDLE cómo encontrar el intérprete de Python correcto en tu sistema. Entonces

encontrarás IDLE en tu carpeta de aplicaciones de usuario.

Si no, vaya a *https://www.python.org/download/mac/tcltk/* y siga las instrucciones; también tendrá que instalar algunos paquetes gráficos de los que depende IDLE.

Instalación de IDLE en Windows

IDLE debería haberse instalado automáticamente cuando instalaste Python. Deberías encontrarlo en tu menú Inicio.

Personalizar la configuración de IDLE

Debido a que es el editor de Python por defecto, la mayoría de los ajustes en IDLE ya están en sintonía con los ajustes recomendados de Python: los tabuladores se convierten automáticamente en espacios, y el indicador de longitud de línea se establece en 80 caracteres de ancho.

Bloques de código con sangría y sin sangría

Para sangrar un bloque de código, resáltelo y seleccione □□□□□□□▸□□□□□□□ **región** o pulse CTRL-]. Para eliminar la sangría de un bloque de código, seleccione **Formato▸Región sangría** o pulse CTRL-[.

Comentario de bloques de código

Para comentar un bloque de código, resalte el código y seleccione □□□□□□□▸ □□□□□□□□**región**, o pulse ALT-3. Para descomentar el código, seleccione □□□□□□□▸□□□□□□□□□□ **región**, o pulse ALT-4.

Emacs y vim

Emacs y vim son dos editores populares preferidos por muchos programadores experimentados porque están diseñados para que tus manos nunca tengan que abandonar el teclado. Esto hace que escribir, leer y modificar código sea muy eficiente una vez que aprendes cómo funciona el editor. También significa que tienen una curva de aprendizaje bastante pronunciada.

Los programadores suelen recomendar que los pruebes, pero muchos programadores expertos olvidan lo mucho que los nuevos programadores ya están intentando aprender. Es beneficioso conocer estos editores, pero no los utilices hasta que te sientas cómodo escribiendo y trabajando con código en un editor más sencillo que te permita centrarte en aprender a programar en lugar de aprender a utilizar un editor.

C

GETTING HELP



Todo el mundo se atasca en algún momento cuando está aprendiendo a programar, y una de las habilidades más importantes que hay que aprender como profesional es la programación.

gramática es cómo desatascarse eficazmente. En este apéndice se describen varios métodos que le ayudarán a desenredarse cuando la programación se vuelva confusa.

Primeros pasos

Cuando te encuentres en un callejón sin salida, lo primero que debes hacer es evaluar tu situación. Antes de pedir ayuda a nadie, debes ser capaz de responder con claridad a las tres preguntas siguientes:

- ¿Qué intenta hacer?
- ¿Qué ha probado hasta ahora?

- ¿Qué resultados ha obtenido?

Tus respuestas deben ser lo más específicas posible. Para la primera pregunta, afirmaciones explícitas como "Estoy intentando instalar la última versión de Python 3 en mi máquina Windows 10" son lo suficientemente detalladas como para que otros miembros de la comunidad Python puedan ayudarte. Declaraciones como "Estoy tratando de instalar Python" no proporcionan suficiente información para que otros puedan ofrecer mucha ayuda.

Tu respuesta a la segunda pregunta debe proporcionar suficientes detalles como para que no te aconsejen repetir lo que ya has intentado: "Fui a *http:// python.org/downloads/* y pulsé el botón Download para Python 3. Luego ejecuté el instalador" es más útil que "Fui al sitio web de Python y descargué un instalador. Luego ejecuté el instalador" es más útil que "Fui al sitio web de Python y descargué un instalador".

Para la última pregunta, es útil conocer los mensajes de error exactos que recibiste al buscar una solución en Internet o al pedir ayuda.

A veces, responderte a ti mismo estas tres preguntas te permite ver algo que se te escapa y desatascarte sin tener que ir más lejos. Los programadores incluso tienen un nombre para esto: se llama *depuración con pato de goma*. Si le explicas claramente tu situación a un patito de goma (o a cualquier objeto inanimado) y le haces una pregunta concreta, a menudo serás capaz de responder a tu propia pregunta. Algunas tiendas de programación incluso tienen un patito de goma de verdad para animar a la gente a "hablar con el pato".

Inténtelo de nuevo

Volver al principio e intentarlo de nuevo puede ser suficiente para resolver muchos problemas. Digamos que está intentando escribir un bucle `for` basado en un ejemplo de este libro. Puede que sólo se te haya escapado algo simple, como dos puntos al final de la línea `for`. Repasar los pasos de nuevo puede ayudarte a evitar repetir el mismo error.

Tómate un descanso

Si llevas tiempo trabajando en el mismo problema, tomarte un descanso es una de las mejores tácticas que puedes probar. Cuando trabajamos en la misma tarea durante mucho tiempo, nuestro cerebro empieza a concentrarse en una única solución. Perdemos de vista las suposiciones que hemos hecho, y hacer una pausa nos ayuda a tener una nueva perspectiva del problema. No hace falta que sea una pausa larga, basta con algo que te saque de tu mentalidad actual. Si llevas mucho tiempo sentado, haz algo físico: da un pequeño paseo o sal un rato al aire libre; tal vez bebas un vaso de agua o comas un tentempié ligero y saludable.

Si te sientes frustrado, puede que merezca la pena dejar el trabajo a un lado durante el día. Una buena noche de sueño casi siempre hace que un problema sea más abordable.

Consulte los recursos de este libro

Los recursos en línea para este libro, disponibles a través de

https://www.nostarch
.com/pythoncrashcourse/, incluyen una serie de secciones útiles sobre cómo configurar tu sistema y trabajar a través de cada capítulo. Si aún no lo has hecho, echa un vistazo a estos recursos y comprueba si hay algo que te ayude.

Búsqueda en línea

Lo más probable es que alguien haya tenido el mismo problema que tú y haya escrito sobre él en Internet. Unas buenas habilidades de búsqueda y unas consultas específicas te ayudarán a encontrar recursos existentes para resolver el problema al que te enfrentas. Por ejemplo, si tienes problemas para instalar Python 3 en Windows 10, la búsqueda *python 3 windows 10* puede dirigirte a la respuesta.

Buscar el mensaje de error exacto también puede ser muy útil. Por ejemplo, supongamos que se produce el siguiente error al intentar iniciar una sesión ter- minal de Python:

```
> python
'python' no se reconoce como comando interno o externo
>
```

Buscar la frase completa *python is not recognized as an internal or external command* probablemente arrojará algunos buenos consejos.

Cuando empieces a buscar temas relacionados con la programación, aparecerán repetidamente algunos sitios. Describiré brevemente algunos de ellos para que sepas hasta qué punto pueden serte útiles.

Stack Overflow

Stack Overflow *(*http://stackoverflow.com/) es uno de los sitios más populares de preguntas y respuestas para programadores, y a menudo aparece en la primera página de resultados de búsquedas relacionadas con Python. Los usuarios envían preguntas cuando están atascados y otros miembros intentan dar respuestas útiles. Los usuarios pueden votar por las respuestas que consideran más útiles, por lo que las mejores respuestas suelen ser las primeras que encontrarás.

Muchas preguntas básicas sobre Python tienen respuestas muy claras en Stack Overflow, porque la comunidad las ha perfeccionado con el tiempo. También se anima a los usuarios a publicar actualizaciones, por lo que las respuestas tienden a mantenerse relativamente actualizadas. En el momento de escribir este artículo, más de 400.000 preguntas relacionadas con Python han sido respondidas en Stack Overflow.

Documentación oficial de Python

La documentación oficial de Python *(*http://docs.python.org/) es un poco más complicada para los principiantes, porque el propósito es más documentar el lenguaje que escribir explicaciones. Los ejemplos de la documentación oficial deberían funcionar, pero es posible que no entiendas todo lo que se muestra. Aún así, es un buen recurso para revisar cuando aparezca en tus búsquedas y se volverá más útil para ti a medida que continúes construyendo tu comprensión de Python.

Documentación oficial de la biblioteca

Si estás utilizando una biblioteca específica, como Pygame, matplotlib, Django, etc., los enlaces a la documentación oficial de ese proyecto aparecerán a menudo en las búsquedas; por ejemplo, *http://docs.djangoproject.com/* es muy útil. Si estás planeando trabajar con alguna de estas bibliotecas, es una buena idea familiarizarse con la documentación oficial.

r/aprenderpython

Reddit se compone de una serie de subforos llamados *subreddits*. El subreddit *r/learnpython* (http://reddit.com/r/learnpython/) es bastante activo y solidario. Aquí puedes leer las preguntas de los demás y publicar las tuyas.

Entradas de blog

Muchos programadores mantienen blogs y comparten entradas sobre las partes del lenguaje con las que trabajan. Antes de seguir cualquier consejo, deberías echar un vistazo a los primeros comentarios de una entrada de blog para ver qué reacciones han tenido otras personas. Si no hay comentarios, tómate la entrada con cautela. Es posible que nadie haya verificado el consejo.

IRC (Internet Relay Chat)

Los programadores interactúan en tiempo real a través del IRC. Si estás atascado en un problema y la búsqueda en Internet no te da respuestas, preguntar en un canal de IRC puede ser tu mejor opción. La mayoría de la gente que pasa el rato en estos canales es educada y servicial, especialmente si puedes ser específico sobre lo que estás tratando de hacer, lo que ya has intentado y los resultados que estás obteniendo.

Crear una cuenta IRC

Para crear una cuenta en el IRC, vaya a http://webchat.freenode.net/. Elige un apodo, rellena la casilla Captcha y haz clic en **Conectar**. Verás un mensaje que te da la bienvenida al servidor IRC freenode. En la casilla de la parte inferior de la ventana, introduce el siguiente comando:

```
/msg nickserv registrar contraseña email
```

Introduzca su propia contraseña y dirección *de* correo electrónico en lugar de *contraseña* y *correo electrónico*. Elige una contraseña sencilla que no utilices para ninguna otra cuenta. Esta contraseña no se transmite de forma segura, así que ni siquiera intentes crear una contraseña segura. Recibirás un correo electrónico con instrucciones para verificar tu cuenta. El correo electrónico te proporcionará un comando como este:

```
/msg nickserv verify register nickname codigo_verificacion
```

Pega esta línea en el sitio IRC con *nickname* como el nombre que elegiste antes y un valor para *verification_code*. Ahora estás listo para unirte a un canal.

Canales a los que unirse

Para unirte al canal principal de Python, introduce **/join #python** en la casilla de entrada. Verás una confirmación de que te has unido al canal y alguna información general sobre el canal.

El canal *##learnpython* (con dos hashtags) también suele ser bastante activo. Este canal está asociado a http://reddit.com/r/learnpython/, por lo que también verás mensajes sobre publicaciones en *r/learnpython*. El canal *#pyladies* se centra en apoyar a las mujeres que están aprendiendo Python, así como a las personas que apoyan a las mujeres programadoras. Quizás quieras unirte al canal *#django* si trabajas en aplicaciones web.

Una vez que te hayas unido a un canal, podrás leer las conversaciones que mantienen otras personas y hacer tus propias preguntas.

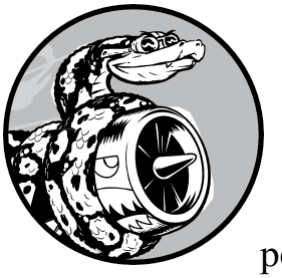
Cultura del IRC

Para obtener una ayuda eficaz, debes conocer algunos detalles sobre la cultura del IRC. Centrarte en las tres preguntas al principio de este apéndice te ayudará definitivamente a guiarte hacia una solución exitosa. La gente estará encantada de ayudarte si puedes explicar con precisión lo que estás tratando de hacer, lo que ya has intentado, y los resultados exactos que estás obteniendo. Si necesitas compartir código o resultados, los miembros del IRC utilizan sitios externos creados para este fin, como *https://bpaste.net/+python/*. (Aquí es donde `#python` te envía a compartir código y resultados.) Esto evita que los canales se inunden de código y también hace que sea mucho más fácil leer el código que la gente comparte.

Ser paciente siempre hará que la gente esté más dispuesta a ayudarte. Plantee su pregunta de forma concisa y espere a que alguien le responda. A menudo, la gente está en medio de muchas conversaciones, pero normalmente alguien se dirigirá a ti en un tiempo razonable. Si hay poca gente en el canal, puede que tardes un poco en recibir una respuesta.

D

USING GIT FOR VERSION CONTROL



Los programas de control de versiones permiten tomar instantáneas de un proyecto siempre que se encuentre en estado de trabajo. Cuando realice cambios en un proyecto -por ejemplo, cuando implementas una nueva función- tienes la opción de volver a un estado de trabajo anterior si el estado actual del proyecto no funciona bien.

El uso de software de control de versiones te da libertad para trabajar en mejoras y cometer errores sin preocuparte por arruinar tu proyecto. Esto es especialmente crítico en proyectos grandes, pero también puede ser útil en proyectos más pequeños, incluso cuando trabajas con programas contenidos en un único archivo.

En este apéndice aprenderás a instalar Git y a usarlo para el control de versiones en los programas en los que estás trabajando ahora. Git es el software de control de versiones más popular hoy en día. Muchas de sus herramientas avanzadas ayudan a los equipos

colaborar en grandes proyectos, pero sus características más básicas también funcionan bien para desarrolladores en solitario. Git implementa el control de versiones mediante el seguimiento de los cambios realizados en cada archivo de un proyecto; si cometes un error, puedes volver a un estado guardado previamente.

Instalación de Git

Git funciona en todos los sistemas operativos, pero hay diferentes enfoques para instalarlo en cada sistema. Las siguientes secciones proporcionan instrucciones específicas para cada sistema operativo.

Instalación de Git en Linux

Para instalar Git en Linux, introduce lo siguiente:

```
$ sudo apt-get install git
```

Ya está. Ya puedes usar Git en tus proyectos.

Instalación de Git en OS X

Puede que Git ya esté instalado en tu sistema, así que prueba a ejecutar el comando **git --version**. Si ves un listado con un número de versión específico, Git está instalado en tu sistema. Si ves un mensaje pidiéndote que instales o actualices Git, simplemente sigue las instrucciones en pantalla.

También puede ir a https://git-scm.com/, seguir el enlace Descargas y hacer clic en el instalador adecuado para su sistema.

Instalación de Git en Windows

Puedes instalar Git para Windows desde http://msysgit.github.io/.

Configuración de Git

Git mantiene un registro de quién realiza cambios en un proyecto, incluso cuando sólo hay una persona trabajando en él. Para ello, Git necesita saber tu nombre de usuario y tu dirección de correo electrónico. Tienes que proporcionar un nombre de usuario, pero siéntete libre de inventarte una dirección de correo falsa:

```
$ git config --global user.name "nombredeusuario"
$ git config --global user.email "username@example.com"
```

Si olvidas este paso, Git te pedirá esta información cuando hagas tu primer commit.

Elaboración de un proyecto

Vamos a crear un proyecto con el que trabajar. Crea una carpeta en algún lugar de tu sistema llamada *git_practice*. Dentro de la carpeta, haz un programa Python simple:

hola_mundo.py

```
print("¡Hola mundo Git!")
```

Utilizaremos este programa para explorar la funcionalidad básica de Git.

Ignorar archivos

Los archivos con la extensión *.pyc* se generan automáticamente a partir de los archivos *.py, por* lo q u e no necesitamos que Git haga un seguimiento de ellos. Estos archivos se almacenan en un directorio llamado *pycache* . Para decirle a Git que ignore este directorio, crea un archivo especial llamado *.gitignore* -con un punto al principio del nombre del archivo y sin extensión de archivo-y añádele la siguiente línea:

.gitignore

```
pycache /
```

Esto le dice a Git que ignore cualquier archivo en el directorio *pycache*. Usando un .gitignore mantendrá su proyecto libre de desorden y será más fácil trabajar con él.

NOTE *Si utilizas Python 2.7, sustituye esta línea por* *.pyc. *Python 2.7 no crea un directorio* pycache*; cada archivo* .pyc *se almacena en el mismo directorio que su archivo* .py *correlativo. El asterisco indica a Git que ignore cualquier archivo con la extensión* .pyc.

Es posible que tenga que modificar la configuración de su editor de texto para que muestre los archivos ocultos con el fin de abrir *.gitignore*. Algunos editores están configurados para ignorar los nombres de archivo que comienzan con un punto.

Inicializar un repositorio

Ahora que tienes un directorio que contiene un archivo Python y un archivo *.gitignore*, puedes inicializar un repositorio Git. Abre un terminal, navega a la carpeta *git_practice*, y ejecuta el siguiente comando:

```
git_practice$ git init
Inicializado repositorio Git vacío en git_practice/.git/
git_practice$
```

La salida muestra que Git ha inicializado un repositorio vacío en *git_practice*. Un *repositorio* es el conjunto de archivos de un programa que Git está rastreando activamente. Todos los archivos que Git utiliza para gestionar el repositorio se encuentran en el directorio oculto *.git/,* con el que no necesitarás trabajar en absoluto. Simplemente no borres ese directorio, o

perderás el historial de tu proyecto.

Comprobar el estado

Antes de nada, veamos el estado del proyecto:

```
git_practice$ git status
❶ # En la rama
master #
# Confirmación
inicial #
❷ # Archivos no rastreados:
#   (use "git add <file>..." para incluir en lo que será
comprometido) #
#   .gitignore
#   hello_world.py
#
❸ no se ha añadido nada a la confirmación, pero hay archivos sin seguimiento (usa
"git add" para hacer el seguimiento) git_practice$
```

En Git, una *rama* es una versión del proyecto en el que estás trabajando; aquí puedes ver que estamos en una rama llamada `master` ❶. Cada vez que compruebes el estado de tu proyecto, debería decir que estás en la rama `master`. Entonces vemos que estamos a punto de hacer el commit inicial. Un *commit* es una instantánea del proyecto en un momento determinado.

Git nos informa de que hay archivos sin rastrear en el proyecto ❷, porque aún no le hemos dicho qué archivos rastrear. Entonces se nos dice que no hay nada añadido al commit actual, pero que hay archivos sin rastrear presentes que podríamos querer añadir al repositorio ❸.

Añadir ficheros al repositorio

Vamos a añadir los dos archivos al repositorio, y comprobar el estado de nuevo:

```
❶ git_practice$ git add .
❷ git_practice$ git status
# On branch master
#
# Confirmación
inicial #
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
❸     #nuevo archivo:        .gitignore
#Nuevo           archivo:
    hola_mundo.py
git_practice$
```

El comando `git add .` añade al repositorio ❶ todos los archivos de un proyecto que aún no están siendo rastreados. No confirma los archivos; sólo le dice a Git que empiece a prestarles atención. Cuando comprobamos el estado de

proyecto ahora, podemos ver que Git reconoce algunos cambios que necesitan ser confirmados ❷. La etiqueta *nuevo archivo* significa que estos archivos fueron añadidos recientemente al repositorio ❸.

Comprometerse

Hagamos el primer commit:

```
❶ git_practice$ git commit -m "Iniciado proyecto."
❷ [master (root-commit) c03d2a3] Proyecto iniciado.
❸   2  ficheros cambiados, 1
   inserción(+) crear modo 100644
   .gitignore crear modo 100644
   hello_world.py
❹ git_practice$ git status
  # On branch master
  nada que confirmar, directorio de trabajo
  limpio git_practice$
```

Ejecutamos el comando `git commit -m "`*mensaje*`"` ❶ para tomar una instantánea del proyecto. La bandera `-m` indica a Git que registre el mensaje que sigue (`"Proyecto iniciado."`) en el registro del proyecto. La salida muestra que estamos en la rama `maestra` ❷ y que dos archivos han cambiado ❸.

Cuando comprobamos el estado ahora, podemos ver que estamos en la rama `maestra`, y tenemos un directorio de trabajo limpio ❹. Este es el mensaje que quieres ver cada vez que confirmes un estado de trabajo de tu proyecto. Si obtienes un mensaje diferente, léelo cuidadosamente; es probable que hayas olvidado añadir un archivo antes de hacer un commit.

Comprobación del registro

Git mantiene un registro de todos los commits realizados en el proyecto. Vamos a comprobar el registro:

```
git_practice$ git log
commit a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f
Autor: Eric Matthes <eric@example.com>
Fecha:  Lun Mar 16 07:23:32 2015 -0800

    Proyecto iniciado.
git_practice$
```

Cada vez que realizas un commit, Git genera un ID de referencia único de 40 caracteres. Registra quién hizo la confirmación, cuándo se hizo y el mensaje grabado. No siempre necesitarás toda esta información, por lo que Git proporciona una opción para imprimir una versión más simple de las entradas del registro:

```
git_practice$ git log --pretty=oneline
a9d74d87f1aa3b8f5b2688cb586eac1a908cfc7f Proyecto iniciado.
git_practice$
```

La opción `--pretty=oneline` proporciona los dos datos más importantes: el ID de referencia de la confirmación y el mensaje registrado para la confirmación.

El segundo compromiso

Para ver el verdadero poder del control de versiones, tenemos que hacer un cambio en el proyecto y confirmar ese cambio. Aquí sólo vamos a añadir otra línea a *hello_world.py*:

hola_mundo.py

```
print("¡Hola mundo Git!")
print("Hola a todos")
```

Si comprobamos el estado del proyecto, veremos que Git se ha dado cuenta de que el archivo ha cambiado:

```
git_practice$ git status
❶ # En la rama master
# Changes not staged for commit:
#   (use "git add <archivo>..." para actualizar lo que se confirmará)
#   (use "git checkout -- <fichero>..." para descartar los cambios en el
directorio de trabajo) #
❷ #modificado     :
    hello_world.py #
❸ no se han añadido cambios al commit (usa "git add" y/o "git commit -a")
git_practice$
```

Vemos la rama en la que estamos trabajando ❶, el nombre del archivo que fue modificado ❷, y que ningún cambio ha sido confirmado ❸. Vamos a confirmar el cambio y comprobar el estado de nuevo:

```
❶ git_practice$ git commit -am "Saludo extendido."
[master 08d4d5e] Saludo extendido.
 1 archivo modificado, 1 inserción(+)
❷ git_practice$ git status
# On branch master
nada que confirmar, directorio de trabajo limpio
❸ git_practice$ git log --pretty=oneline
08d4d5e39cb906f6cff197bd48e9ab32203d7ed6 Saludo extendido.
be017b7f06d390261dbc64ff593be6803fd2e3a1 Proyecto iniciado.
git_practice$
```

Hacemos un nuevo commit, pasando la bandera `-am` cuando usamos el comando `git commit` ❶. La opción `-a` indica a Git que añada todos los archivos modificados en el repositorio a la confirmación actual. (Si creas algún archivo nuevo entre confirmaciones, simplemente vuelve a emitir el comando `git add .` para incluir los nuevos archivos en el repositorio). La opción `-m` indica a Git que registre un mensaje en el log para esta confirmación.

Cuando comprobamos el estado del proyecto, vemos que una vez más

tenemos un directorio de trabajo limpio ❷. Por último, vemos los dos commits en el log ❸.

Anular un cambio

Ahora vamos a ver cómo abandonar un cambio y volver al estado de trabajo anterior. Primero, añade una nueva línea a *hello_world.py*:

hola_mundo.py

```
print("¡Hola mundo Git!")
print("Hola a todos")

print("¡Oh no, he roto el proyecto!")
```

Guarde y ejecute este archivo.

Comprobamos el estado y vemos que Git se da cuenta de este cambio:

```
git_practice$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <archivo>..." para actualizar lo que se confirmará)
#   (use "git checkout -- <fichero>..." para descartar los cambios en el
directorio de trabajo) #
❶ #modificado     :
    hello_world.py #
no se han añadido cambios al commit (usa "git add" y/o "git
commit -a") git_practice$
```

Git ve que hemos modificado *hello_world.py* ❶, y podemos confirmar el cambio si queremos. Pero esta vez, en lugar de confirmar el cambio, queremos volver a la última confirmación cuando sabíamos que nuestro proyecto estaba funcionando. No haremos nada en *hello_world.py*; no borraremos la línea ni usaremos la función Deshacer del editor de texto. En su lugar, introduzca los siguientes comandos en su sesión de terminal:

```
git_practice$ git checkout .
git_practice$ git status
# On branch master
nada que confirmar, directorio de trabajo
limpio git_practice$
```

El comando `git checkout` te permite trabajar con cualquier commit anterior. El comando git `checkout .` abandona cualquier cambio realizado desde la última confirmación y restaura el proyecto al último estado confirmado.

Cuando vuelvas a tu editor de texto, verás que *hello_world.py* ha cambiado de nuevo a esto:

```
print("¡Hola Git mundo!")
print("Hola a todos.")
```

Aunque volver a un estado anterior puede parecer trivial en este sencillo proyecto, si estuviéramos trabajando en un proyecto grande con docenas de archivos modificados, se revertirían todos los archivos que hubieran cambiado desde el último commit. Esta característica es increíblemente útil: puedes hacer tantos cambios como

desea al implementar una nueva función, y si no funcionan, puede descardarlos sin dañar el proyecto. No tienes que recordar esos cambios y deshacerlos manualmente. Git hace todo eso por ti.

NOTE *Puede que tenga que hacer clic en la ventana de su editor para actualizar el archivo y ver la versión anterior.*

Comprobación de confirmaciones anteriores

Puedes comprobar cualquier confirmación de tu registro, no sólo la más reciente, incluyendo los seis primeros caracteres del ID de referencia en lugar de un punto. Al comprobarlo, puedes revisar un commit anterior y volver al commit más reciente o abandonar tu trabajo reciente y retomar el desarrollo desde el commit anterior:

```
git_practice$ git log --pretty=oneline
08d4d5e39cb906f6cff197bd48e9ab32203d7ed6 Saludo extendido.
be017b7f06d390261dbc64ff593be6803fd2e3a1 Proyecto iniciado.
git_practice$ git checkout be017b
Nota: comprobando 'be017b'.

❶ Estás en el estado 'detached HEAD'. Puedes echar un vistazo, hacer cambios
  experimentales y confirmarlos, y puedes descartar cualquier confirmación que
  hagas en este estado sin afectar a ninguna rama realizando otra comprobación.

  Si desea crear una nueva rama para retener los commits que cree, puede
  hacerlo (ahora o más adelante) utilizando -b con el comando checkout de
  nuevo. Ejemplo:

    git checkout -b nuevo_nombre_de_rama

  HEAD está ahora en be017b7... Proyecto iniciado.
  git_practice$
```

Cuando compruebas un commit anterior, dejas la rama maestra y entras en lo que Git denomina un estado *HEAD separado* ❶. *HEAD* es el estado actual del proyecto; estamos *separados* porque hemos dejado una rama con nombre (`master`, en este caso).

Para volver a la rama `maestra`, la compruebas:

```
git_practice$ git checkout master
La posición HEAD anterior era be017b7... Proyecto iniciado.
Cambiado a la rama 'master
git_practice$
```

Esto te devuelve a la rama `maestra`. A menos que quieras trabajar con algunas características más avanzadas de Git, lo mejor es no hacer ningún cambio en tu proyecto cuando hayas comprobado un commit antiguo. Sin embargo, si estás

es el único que trabaja en un proyecto y desea descartar todas las confirmaciones más recientes y volver a un estado anterior, puede restablecer el proyecto a una confirmación anterior. Trabajando desde la rama `maestra`, introduce lo siguiente:

```
❶ git_practice$ git status
  # On branch master
  nada que confirmar, directorio de trabajo limpio
❷ git_practice$ git log --pretty=oneline
  08d4d5e39cb906f6cff197bd48e9ab32203d7ed6 Saludo extendido.
  be017b7f06d390261dbc64ff593be6803fd2e3a1 Proyecto iniciado.
❸ git_practice$ git reset --hard be017b
  HEAD está ahora en be017b7 Proyecto iniciado.
❹ git_practice$ git status
  # On branch master
  nada que confirmar, directorio de trabajo limpio
❺ git_practice$ git log --pretty=oneline
  be017b7f06d390261dbc64ff593be6803fd2e3a1 Proyecto iniciado.
  git_practice$
```

Primero comprobamos el estado para asegurarnos de que estamos en la rama `maestra` ❶. Cuando miramos el log, vemos ambos commits ❷. A continuación, ejecutamos el comando `git reset --hard` con los seis primeros caracteres del ID de referencia de la confirmación a la que queremos volver de forma permanente ❸. Comprobamos de nuevo el estado y vemos que estamos en la rama `maestra` sin nada que confirmar ❹. Cuando miramos el log de nuevo, vemos que estamos en el commit desde el que queríamos empezar de nuevo ❺.

Borrar el repositorio

A veces estropeará el historial de su repositorio y no sabrá cómo recuperarlo. Si esto ocurre, primero considere pedir ayuda usando los métodos discutidos en el Apéndice C. Si no puedes arreglarlo y estás trabajando en un proyecto en solitario, puedes continuar trabajando con los archivos pero deshacerte del historial del proyecto borrando el directorio *.git*. Esto no afectará al estado actual de ninguno de los archivos, pero borrará todos los commits, por lo que no podrás comprobar ningún otro estado del proyecto.

Para ello, abre un explorador de archivos y elimina el repositorio *.git* o hazlo desde la línea de comandos. Después, tendrás que empezar de nuevo con un repositorio nuevo para empezar a seguir tus cambios de nuevo. Este es el aspecto de todo el proceso en una sesión de terminal:

```
❶ git_practice$ git status
  # On branch master
  nada que confirmar, directorio de trabajo limpio
❷ git_practice$ rm -rf .git
❸ git_practice$ git status
```

```
fatal: No es un repositorio git (ni ninguno de los directorios padre): .git
❹ git_practice$ git init
Inicializado repositorio Git vacío en git_practice/.git/
```

```
❺ git_practice$ git status
  # On branch master
  #
  # Confirmación
  inicial #
  # Archivos no rastreados:
  #   (use "git add <file>..." para incluir en lo que será
  comprometido) #
  #   .gitignore
  #   hello_world.py
  #
  no se ha añadido nada a la confirmación, pero hay archivos sin seguimiento (use "git
  add" para realizar el seguimiento)
❻ git_practice$ git add .
  git_practice$ git commit -m "Empezando de nuevo".
  [master (root-commit) 05f5e01] Empezando de nuevo.
   2 ficheros cambiados, 2
   inserciones(+) create mode 100644
   .gitignore create mode 100644
   hello_world.py
❼ git_practice$ git status
  # On branch master
  nada que confirmar, directorio de trabajo limpio
  git_practice$
```

Primero comprobamos el estado y vemos que tenemos un directorio de trabajo limpio ❶. Luego usamos el comando `rm -rf .git` para borrar el directorio *.git* (`rmdir /s .git` en Windows) ❷. Cuando comprobamos el estado después de borrar el directorio *.git*, se nos dice que no es un repositorio Git ❸. Toda la información que Git utiliza para rastrear un repositorio se almacena en la *carpeta .git*, por lo que eliminarla borra todo el repositorio.

Entonces somos libres de utilizar `git init` para iniciar un repositorio fresco ❹. Comprobar el estado muestra que estamos de vuelta en la etapa inicial, a la espera de la primera confirmación ❺. Añadimos los archivos y hacemos el primer commit ❻. Comprobar el estado ahora nos muestra que estamos en la nueva rama `maestra` sin nada que confirmar ❼.

Utilizar el control de versiones requiere un poco de práctica, pero una vez que empieces a usarlo no querrás volver a trabajar sin él.

INDEx

Símbolos

+ (suma), 30
operador * (asterisco), 157
Archivos de imagen *.bmp* (Bitmap), 244
+ (operador de concatenación), 101-102
/ (división), 30
== (operador de igualdad), 77
> (mayor que), 79
>= (mayor o igual que), 79
(almohadilla), para comentarios, 33
!= (operador de desigualdad), 78
< (menos de), 79
<= (menor o igual que), 79
% (operador de módulo), 120-121
* (multiplicación), 30
\n (nueva línea), 26
¡! (no), 78
+= operador, 119
Extensión de archivo *.py*, 20
[] (corchetes), 38
- (resta), 30
\t (tab), 26

A

adición (+), 30
alias, 156, 157
alice.py, 203-205
Proyecto Invasión alienígena. *Ver también*
Pygame
extraterrestres
Clase `extranjera`, 267
construcción de la flota de, 269-275 cambio de dirección, 278
control de los bordes, 278
crear, 266 crear filas de, 270

dibujar en la pantalla, 268 soltar, 278
encajar en una fila, 269 movimiento, 276-279 varias filas de, 273 llegar al fondo de pantalla, 288 reconstrucción de la flota de, 282
color de fondo, configuración, 242 viñetas
Clase `bala`, 258-259 comprobación con `impresión` declaraciones, 262
borrar antiguo, 261
disparo, 260
limitar el número de, 262 hacer más grande, 281
ajustes, 257
acelerar, 283 clases
`Alien`, 267
`Bala`, 258-259
`Botón`, 292
`Marcador`, 301
`Ajustes`, 243
`Barco`, 245
colisiones, 280
bala alienígena, 280
nave-extraterrestre, 284-288 terminar una partida, 288-289 atajos de teclado, 266 niveles
añadir, 299-301 modificar la velocidad ajustes, 299
reajuste de las velocidades, 300
planificación, 236

Proyecto Invasión Extraterrestre, *continuación*
Botón de reproducción
añadiendo, 292-298
Clase de `botones`, 292
desactivación, 297
dibujo, 294 ocultar el ratón
cursor, 298
reiniciar el juego, 296
iniciar el juego, 295
revisión, 266
puntuación
visualización del nivel, 310-312 visualización del número de
barcos, 313-316 mostrar la puntuación, 301 aumentar los valores de los puntos, 306 hacer un sistema de puntuación,
301-316
redondeo y formateo de la puntuación, 307
Clase de `marcador`, 301
anotar todos los golpes, 305
seguimiento de la puntuación más alta, 308
seguimiento de la puntuación, 301
barco
ajuste de la velocidad, 253-254
movimiento continuo, 250-253
encontrar una imagen, 244
limitar el alcance, 255
Clase de `buque`, 245
alien.py, 96-100
aliens.py, 109-111
americas.py, 367
parque_de_ atracciones.py, 84-87
y palabra clave, 79-80
API (programación de aplicaciones interfaz), 377
llamada, 378
respuesta de procesamiento, 379-383
solicitud de datos mediante, 378 resumen de resultados de, 382
para Hacker News, 390-393
límites de tarifa, 383
apóstrofe.py, 28

método `append()`, 41-42
interfaz de programación de aplicaciones. *Véase* API (interfaz de programación de aplicaciones)
argumentos, 135-141. *Véase también* funciones aritméticas, 30
`como` palabra clave, 156, 157
métodos assert, 218, 222-223 operador asterisco (*), 157 atributos, 163
    acceso, 164
    valores por defecto, 168
    modificación de los valores, 168-171

B

banned_users.py, 81
Bash, 475
bicicletas.py, 38-40
cumpleaños.py, 31
archivos de imagen de mapa de bits (*.bmp)*, 244
cuerpo
    de una función, 134
    de un archivo HTML, 458
Valores booleanos, 81
Bootstrap, 457
funciones incorporadas, 490

C

llamando a
    funciones, 133, 134, 139-140
    métodos, 165
CamelCaps, 186
car.py, 167-171
módulo *car.py*, 179-181, 183-184
coches.py, 47-49, 76
clases hijo, 172. *Véase también* clases: herencia
ciudades.py, 125
clases, 161-162
    atributos, 163
        acceso, 164
        valores por defecto, 168
        modificación de los valores, 168-171
    creación, 162-164

importando
todas las clases de un módulo, 182-183
módulo completo, 182
módulo en un módulo, 183-184
clases múltiples, 181-182
clase única, 179-180
herencia, 172-178 atributos y métodos, 174-175
método `init()`, 172-173
instancias como atributos, 175-177
métodos de anulación, 175
en Python 2.7, 173
función `super()`, 173
métodos
llamada, 165
`init()`, 163
modelado de objetos del mundo real, 177-178
instancias múltiples, creación, 165 directrices de estilo, 186-187
colormap_colors.py, 330
colores
formato hexadecimal, 373
Temas pygal, 374
archivos de valores separados por comas. *Véase* Archivos CSV (valores separados por comas)
comment.py, 33
comentarios, 33-34
operadores de comparación, 78-80
operador de concatenación (`+`), 101-102
pruebas condicionales, 76-81
confirmed_users.py, 128
counting.py, 122, 126
países.py, 365
códigos de país, 365-367
códigos_país.py, 365
Archivos CSV (valores separados por comas), 350-356
comprobación de errores, 359
análisis de cabeceras, 350
lectura de datos, 352

D

bases de datos. *Ver* Django: `bases de datos`; Heroku: extracción de datos de bases de datos, 321
visualización de datos, 321-322. *Véase también* matplotlib; módulo `datetime` de Pygal, 354-355
Death Valley, datos meteorológicos, 350, 359-362
consejos de depuración, 499-500
decoradores, 447
valores por defecto
atributos de clase, 168
parámetros de función, 138
palabra clave `def`, 134
declaración `del`, 100
dice_visual.py, 343-346
diccionarios
definición, 96
vacío, 98
formato más grande, 101 pares clave-valor
añadiendo, 97
quitando, 100
en `bucle`, 102
llaves, 104-106
llaves en orden, 106-107
pares clave-valor, 103-104
valores, 107-108 valores
acceso, 97
modificación, 99-100
Clase de `troquel`, 340
die.py, 340
die_visual.py, 341-343
dimensiones.py, 70-71
`div` (HTML), 460
división (`/`), 30
division.py, 200-203
Django, 397. *Véase también* Heroku; Sitio de administración del proyecto
Learning Log, 406-408
Bootstrap, 457
directorio de trabajo actual, 469

Django, *continuación*
datos
asociarse a un usuario, 453
conectarse a usuarios, 448
restringir el acceso a, 446-453
bases de datos
creación, 401
claves externas, 408 muchos-a-uno
relaciones, 408
migración, 401, 409, 450-451
consultas, 423
conjuntos de consulta, 410
formularios, 428-438
cross-site request falsificación, 431
visualización, 431 GET y POST
solicitudes, 430
ModeloFormulario, 428, 432
tratamiento, 430, 434
validación, 428
widgets, 432
hashes (para contraseñas), 406
Error HTTP 404, 452
`INSTALLED_APPS`, modificación, 405
jQuery, 457
localhost, 402
`@login_required`, 447
vista de `acceso`, 440
manage.py, 400
modelos, 403-405
privilegios, 406
comando `runserver`, 401
archivos estáticos, 467
superusuario, configuración, 406
plantillas
etiquetas de anclaje, 417
etiquetas de `bloque`, 417
contexto, 420
filtros, 423
filtro de `saltos de línea`, 423
etiquetas de plantilla, 417
aplicaciones de terceros, 456
URLs
espacios de nombres, 417
patrones, 414

expresiones regulares, 414, 422
valores de ID de usuario, 449 vistas, 412
servidor web gateway interfaz, 401
aplicación django-boostrap3, 456
docstrings, 134
perro.py, 162-166
notación por puntos, 155, 164

E

Einstein, Albert, 29
coche_eléctrico.py, 172-178
módulo *electric_car.py*, 183
declaración `elif`, 84-88
sentencia `else`, 83-84
Emacs, 497
tiempo de época, 384
operador de igualdad (==), 77
números_pares.py, 62
even_or_odd.py, 121
bucles de eventos, 241-242
excepciones, manipulación, 200 decidir qué errores informe, 207
`otros` bloques, 202-203
fracasar silenciosamente, 206-207
`FileNotFoundError`, 203-204 para evitar
fallos, 201-202 bloques `try-except`, 200-201
`ErrorDivisiónCero`, 200

F

favorite_languages.py, 100-102, 104-108, 112
`FileNotFoundError`, 203-204
file_reader.py, 190-194 archivos
cierre, 191
grande, 195-196
apertura, 190
caminos, 191-192 lectura
archivos completos, 190-191 línea por línea, 193
haciendo una lista de líneas, 194

modos lectura/escritura, 198
trabajar con contenidos, 194
escribir en
añadiendo, 199
archivos vacíos, 197
líneas múltiples, 198
banderas, 124
flotadores, 30-31
foods.py, 67-69
bucles `for`, 53-60
funciones, 133
argumentos, 135-141
arbitraria, 151-152 evitar errores con, 140-141
palabra clave, 137-138
listas, 147-150
opcional, 142-144
orden de, 137
argumentos posicionales, 136-137, 152
incorporado, 490
llamada, 133, 134, 139-140
varias veces, 136-137
definir, 134
diccionarios, volver, 144-145
sentencias `import`, 154
listas en
modificar, 147-149 impedir modificar, 149-150
módulos, almacenamiento en, 154-157. *Véase también* módulos
parámetros, valores por defecto para, 138
pasar información a, 134
valores de retorno, 141-146
estilo, 158

G

juegos. *Véase el* proyecto Alien Invasion; Pygame
Geany, 492-494
comentar bloques, 494
personalizar la configuración, 493 sangrar y sangrar bloques, 494

instalación de
en Linux, 492
en Windows, 492-493 ejecución de
programas Python, 493
get-pip.py, 238. *Véase también* pip Peticiones
GET, 430
Git, 378, 505-506
ramas, 508
compromisos, 378, 508
comprobación previa, 512 elaboración, 509
HEAD separado, 512 archivos
añadir, 508
ignorar, 472, 507
CABEZA, 512
instalación, 506
registro, comprobación, 509 proyectos
comprobación del estado de, 508
realización de, 507
eliminación de repositorios, 513
inicialización, 507
revertir cambios, 511-512 GitHub, 378.
Véase también Git mayor que (>), 79
mayor o igual que (>=), 79
greeter.py, 118-119, 134-135
greet_users.py, 147
gunicornio, 469, 470-471

H

Hacker News, 390
hash mark (#), para comentarios, 33 head, de un
archivo HTML, 458 HEAD (Git), 512
Hola Mundo, 4
hola_mundo.py, 19-23
Heroku. *Véase también* Django; Proyecto Learning
Log
Bash, 475 bases de datos
migración, 474
puesta en marcha, 474
uso de Postgres, 469

Heroku, *continuación*
creación de una cuenta, 466
Procfile, 469
proyectos
supresión, 481
empujando a, 473
asegurar, 476
Python runtime, especificar, 468
settings.py, modificar para, 468
superusuario, crear, 475
Cinturón de herramientas, instalación, 466
URL de fácil uso, 476
wsgi.py, modificar para, 470
formato hexadecimal, para colores, 373 *highs_lows.py*, 350-361
hn_submissions.py, 391-393
Elaboración casera, 487

I

IDLE, 496-497
comentar el código, 497
personalizar la configuración, 497 instalar
en Linux, 496
en OS X, 496
en Windows, 497
declaraciones `if`
`y` palabra clave, 79-80
Expresiones booleanas, 81
comprobación de
listas vacías, 91
igualdad (`==`), 76-78
desigualdad (`!=`), 78
elementos en una lista, 80-81 elementos no en una lista, 81 elementos especiales, 90-91
declaración `elif`, 84-88
sentencia `else`, 83-84
cadenas `if-elif-else`, 84-93
listas y, 89-92
comparaciones numéricas, 78-80
`o` palabra clave, 80
simple, 82-83
directrices de estilo, 94
pruebas de condiciones
múltiples, 87-88
`importar *`, 157
`importa esto`, 34

errores de sangría, 57-60
errores de índice, 51
operador de desigualdad (!=), 78 bucles
infinitos, 126-127 herencia, 172. *Véase también*
clases:
herencia
función input()
entrada numérica, 119-120
indicaciones, 118-119
método insert(), 42
int, 31
IRC (Internet Relay Chat), 502-503
función itemgetter(), 391-392
método items(), 103

J

jQuery, 457
función json.dump(), 209 archivos JSON
conversión de cadenas a valores numéricos, 364
extracción de datos, 363
función json.load(), 210
jumbotron, 461

K

método keys(), 104
pares clave-valor, 96. *Véase también*
diccionarios
argumentos de palabras clave, 136, 137-138
palabras clave, 489

L

encuesta_idiomas.py, 224
Proyecto Learning Log, 397. *Véase también*
Django; despliegue en
Heroku
commits, uso de Git, 471-473 páginas de
error personalizadas, 478-480
get_object_or_404()
método, 480 ignorar archivos en
Git, 472 hacer un archivo estático
directorio, 470
desarrollo continuo, 480
Procfile, 469

requisitos.txt, 467
configuración de SECRET_KEY, 481 uso de gunicorn localmente, 470
cabeceras HTML, definición, 458 cierre de sesión, 442-443
página de inicio de sesión, 440-442 migrar la base de datos, 450-451
estilo, 456
Bootstrap, 457
navegación plegable, 459
jumbotron, 461
barra de navegación, 458-460
selectores, 459
página de registro, 443-446
usuarios app
vista de inicio de sesión por defecto, 440 mostrar mensajes a usuarios logueados, 441
loguear un usuario, 445
UserCreationForm, 445
función len(), 49
menor que (<), 79
menor o igual que (<=), 79
Linux
Geany (editor de texto), 6, 492-494 Python
comprobación de la versión instalada, 5, 486
instalación, 485-486
configuración, 5-8
ejecutar Hello World, 6-7
terminal
ejecutar comandos desde, 8
ejecutar programas desde, 16
solución de problemas de instalación, 15
listas, 37-38
método append(), 41
como argumentos, 147-150
copia, 67-69
del, 43
elementos
acceso, 38-39
añadir, 41-42
modificar, 40
vacío, 42
bucles for, 53-60

sentencias `if`, 89-92
errores de sangría, 57-60
índices, 39
errores, 50-51
negativo, 39
método `insert()`, 42
función `len()`, 49
comprensión de listas, 63-64
función `max()`, 63
función `min()`, 63
denominación, 38
numérico, 61-64
método `pop()`, 43-45
función `range()`, 62-63
`remove()`, 45 elimina todas las
apariciones de un método
valor, 129
método `reverse()`, 49
rodajas, 65-67
función `sorted()`, 48
método `sort()`, 47
función `sum()`, 63
localhost, 402
errores lógicos, 58
método `lower()`, 24
método `lstrip()`, 27

M

magos.py, 54-57
magic_number.py, 79
making_pizzas.py, 155-157 matplotlib
formato de los ejes de los
gráficos, 337
color, 329-330
etiquetas, 324-325
grosor de línea, 324-325
sombreado, 358
tamaño, 338
instalación, 322-323
función `plot()`, 324 trazado
fechas, 355
series de datos múltiples, 357
módulo `pyplot`, 324
guardar parcelas, 331
gráficos de dispersión, 326-329 gráficos
lineales simples, 324

métodos, 24, 165, 174-175. *Véase también*
clases
ModelForm, 428, 432
módulos, 154-157
alias para, 157
funciones
alias para, 156
importar todo, 157
importación específica, 156
importación de todo un módulo, 154
operador de módulo (%), 120-121 *motorcycles.py*, 41-46
encuesta_montaña.py, 130
mpl_cuadrados.py, 324-326
multiplicación (*), 30
mi_coche.py, 180
mis_coches.py, 181-184
mi_coche_electrico.py, 181

N

errores de nombre, 21-23
nombre_funcion.py, 216-221
nombre.py, 24-25
nombres.py, 216
na_populations.py, 368
anidamiento
diccionarios en diccionarios, 113-114
diccionarios en listas, 109-111 listas en diccionarios, 111-113
nueva línea (\n), 26
no (!), 78
number_reader.py, 209
números
aritmética, 30
evitar errores de tipo, 31-32
comparaciones, 78-80
exponentes, 30
flotadores, 30-31
números enteros, 30-32
orden de las operaciones, 30
función str(), 32
numbers.py, 61
number_writer.py, 209

O

programación orientada a objetos, 161. *Ver también las* clases
función open(), 190
módulo operador, 391-392
o palabra clave, 80 OS X
Python
comprobación de la versión instalada, 8, 486-487
instalación, 486-488
configuración, 8-10
ejecución de Hello World, 10
Sublime Text (editor de texto), 9, 494-496
terminal
ejecutar comandos desde, 9
ejecutar programas desde, 16
solución de problemas de instalación, 15

P

parámetros, 135, 138-139
clases padre, 172. *Véase también* clases: herencia
parrot.py, 118, 122-125
declaración de paso, 206
PEP 8, 72-73
persona.py, 144-146
Peters, Tim, 34
pets.py, 129, 136-140
pi, 190-196
pip, 236
comprobación, 237
get-pip.py, 238
instalación, 238
pi_string.py, 194-196
pizza.py, 111, 151-152
planificar un proyecto, 236
players.py, 65-67
método pop(), 43-45
argumentos posicionales, 135, 136-137, 152
Postgres, 469
Solicitudes POST, 430

imprimiendo_modelos.py, 147-150
instrucciones de `impresión`, 24
largo, 102
en Python 2, 29
privilegios, 406
Procfile, 469
Proyecto Gutenberg, 208
indicaciones, para la introducción de datos por el usuario, 118-119
extensión de archivo *.py*, 20 Pygal, 322
gráficos
histogramas, 342-343
enlaces, añadir, 390
estilo, 386-388
información sobre herramientas, añadir, 387
temas de color, 374
instalación, 340
diccionarios de trazado, 388
tirar los dados, 339
WorldMap, 367-374
agrupación de países, 371-372 trazado de datos numéricos, 368-372
estilo, 372-374 Pygame. *Véase también* Invasión alienígena
proyecto
archivos de imagen de mapa de bits (*.bmp)*, 244 colores, 242-243
crear una ventana vacía, 241
mostrar texto, 293
dibujar imágenes a la pantalla, 246 grupos
vaciado, 282
almacenar elementos en, 259 actualizar todos los elementos en, 260
instalar, 236-240 posicionar imágenes con `recto`, 245
respuesta a entradas, 241-242
pulsaciones de teclas, 249-250
clics del ratón, 295-296
coordenadas de pantalla, 246
superficies, 241
módulo `pyplot`, 324
Python
funciones incorporadas, 490
documentación, 501

instalación de
en Linux, 485-486 en OS X, 487
en Windows, 488-489
intérprete, 20
palabras clave, 489
PEP 8, 72-73
biblioteca estándar, 184-185
sesión terminal, 4
`--versión`, 486
Zen de, 34-36
Python 2.7
creación de clases en, 164 división de
enteros en, 32 sentencias de `impresión` en,
29 función `raw_input()`, 121
Propuesta de mejora de Python (PEP), 72
python_repos.py, 379-390

Q

consultas
en las bases de datos, 423
en GitHub (`q=`), 378-379 querysets, en
bases de datos, 410 quit values, 122-124

R

random_walk.py, 332
paseos aleatorios, 331-332
puntos de coloración, 335-336
método `fill_walk()`, 332 paseos
múltiples, generar,
334-335
trazado, 333-334
Clase `RandomWalk`, 332 puntos de inicio y
fin,
trazado, 336
función `range()` 61-63
método `readlines()`, 194
método `read()`, 191
`rect`, posicionamiento de imágenes con, 245 reddit,
502
refactorización, 212-214
remember_me.py, 210-214
paquete de solicitudes, 379

recursos
consejos de depuración, 499-500
documentación, 501-502
IRC (Internet relay chat), 502-503
canales, 503
cultura, 503
crear una cuenta, 502
reddit, 502
Stack Overflow, 501
valores de retorno, 141
rollercoaster.py, 120
dados rodantes
análisis de resultados, 341
diferentes tamaños, 345
dos dados, 343
depuración del patito de goma, 500
rw_visual.py, 333-338

S

scatter_squares.py, 327-330
Configuración de `SECRET_KEY`, 481
método `setUp()`, 227
Sitka, Alaska, datos meteorológicos, 350-362
rebanada, 65
función `sorted()`, 48
método `sort()`, 47
método `split()`, 204
SQLite, 401
corchetes (`[]`), 38
squares.py, 62, 64
Stack Overflow, 501
almacenamiento de datos
método `json.dump()`, 209
método `json.load()`, 209
guardar y leer datos, 210-211
Función `str()`, 32
cadenas
cambio de maletín, 24
concatenación, 25
nuevas líneas en, 26
comillas simples y dobles, 23
tabulaciones en, 26
espacios en blanco, 26-27
método `strip()`, 27

normas de estilo, 72
líneas en blanco, 73
CamelCaps, 186
clases, 186
funciones, 158
sentencias `if`, 94
sangría, 72
longitud de línea, 73
PEP 8, 72
subclases, 173. *Véase también* clases: herencia
Sublime Text, 494 comentar código, 496 configurar, 495
sangrar y sangrar bloques de código, 496
instalación de
en Linux, 494 en OS X, 494
en Windows, 495
ejecución de programas Python, 495
sustracción (`-`), 30
superclases, 173. *Véase también* clases: herencia
superusuario
en Django, 406
en Heroku, 475
survey.py, 223
errores de sintaxis, 28
resaltado de sintaxis, 20

T

tab (`\t`), 26 código de comprobación
añadir pruebas, 221
métodos de afirmación, 218, 222-223
pruebas fallidas, 218-220
cobertura total, 217
funciones, 216-221
superación de pruebas, 217-218
caso de prueba, 217
clases de pruebas, 222-228
ejemplos, 223-226
método `setUp()`, 227
módulo `unittest`, 215, 217
pruebas unitarias, 217
test_nombre_funcion.py, 217-221

test_survey.py, 225-228
editores de texto, 491-497
método `title()`, 24
Cinturón de herramientas, Heroku, 466
toppings.py, 78, 87-92
rastreo, 22
seguimiento de expedientes de proyectos, 378, 471-473. *Véase también* Git
bloques `try-except`, 200-201. *Véase también* excepciones, gestión
tuplas
  definición, 69
  bucle `for`, 70
  escribir sobre, 71
errores tipográficos, 31-32, 70

U

módulo `unittest`, 215, 217
pruebas unitarias, 217
Tiempo Unix, 384
método `upper()`, 24
user_profile.py, 152
user.py, 103-104
usuarios.py, 113

V

valores
  pares clave-valor, 96. *Véase también* diccionarios
  de variables, 20
método `values()`, 107
variables
  definición, 20
  errores de nombre, 21-23
  convenciones de nomenclatura, 21 sistema de control de versiones, 378. *Véase también* Git
vim, 497
desactivación del entorno virtual, 399
  instalación de `virtualenv`, 399
voting.py, 83-84

W

datos meteorológicos, 350-362
marco web, 397
bucles `while`
  bandera `activa`, 124-125
  sentencia `break`, 125
  infinito, 126-127
  mover elementos entre listas, 128
  abandonar valores, 122-124
espacio en blanco
  añadiendo, 26
  decapado, 26-27
Ventanas
  Geany (editor de texto), 12-14, 492-494
  Python
    instalación, 11, 488-489
    configuración, 10-14
  ejecución de Hello World, 14 terminal
    ejecutar comandos desde, 11-12
    ejecución de programas desde, 16-17
  solución de problemas de instalación, 15
`con` palabra clave, 190
word_count.py, 205-207
población_mundo.py, 363-374
escribir_mensaje.py, 197-199
método `write()`, 198

Z

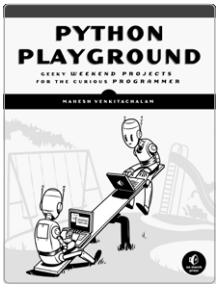
Zen de Python, 34-36
`ErrorDivisiónCero`, 200

Python Crash Course está escrito en New Baskerville, Futura, Dogma y TheSansMono Condensed.

RECURSOS

Visite *https://www.nostarch.com/pythoncrashcourse/* para obtener recursos, erratas y más información.

Más libros sensatos de NO STARCH PRESS



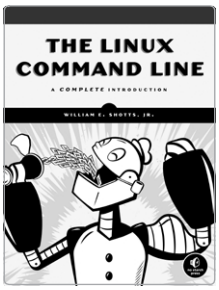
PATIO DE RECREO PYTHON
Proyectos frikis de fin de semana para programadores curiosos
por MAHESH VENKITACHALAM
OCTUBRE 2015, 352 PP., $29.95
ISBN 978-1-59327-604-1



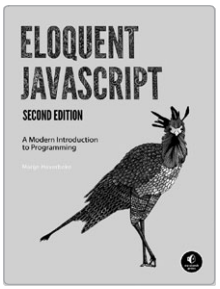
AUTOMATICE LAS COSAS ABURRIDAS CON PYTHON
Programación práctica para principiantes
por AL SWEIGART
ABRIL DE 2015, 504 PÁGS., 29,95 $.
ISBN 978-1-59327-599-0



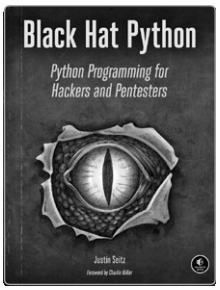
MATEMÁTICAS CON PYTHON
Utiliza la programación para explorar el álgebra, la estadística, el cálculo y mucho más.
por AMIT SAHA
AGOSTO DE 2015, 264 PÁGS., 29,95 €.
ISBN 978-1-59327-640-9



LA LÍNEA DE MANDO LINUx
Una introducción completa
por WILLIAM E. SHOTTS, JR.
ENERO 2012, 480 PP., $39.95
ISBN 978-1-59327-389-7



JAVASCRIPT ELOCUENTE, 2ª EDICIÓN
Introducción moderna a la programación
por MARIJN HAVERBEKE
DICIEMBRE DE 2014, 472 PÁGS., 39,95 $.
ISBN 978-1-59327-584-6



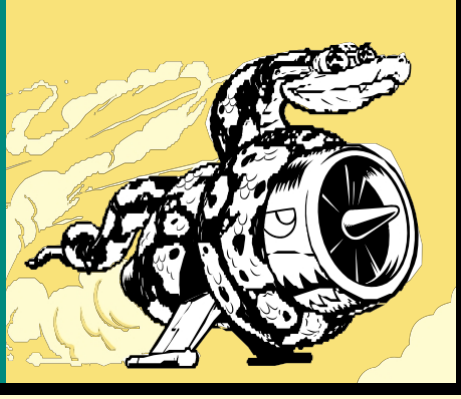
SOMBRERO NEGRO PYTHON
Programación en Python para hackers y pentesters
por JUSTIN SEITZ
DICIEMBRE DE 2014, 192 PÁGS., 34,95 $.
ISBN 978-1-59327-590-7

TELÉFONO:
800.420.7240 O
415.863.9900

CORREO ELECTRÓNICO:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

APRENDA PYTHON - ¡RÁPIDO!



Python Crash Course es una introducción rápida y completa a la programación con Python que le permitirá escribir programas, resolver problemas y crear cosas que funcionan en muy poco tiempo.

En la primera mitad del libro, aprenderá conceptos básicos de programación, como listas, diccionarios, clases y bucles, y practicará la escritura de código limpio y legible con ejercicios para cada tema. También aprenderá cómo hacer que sus programas sean interactivos y cómo probar su código de forma segura antes de añadirlo a un proyecto. En la segunda mitad del libro, pondrás en práctica
Pon en práctica tus nuevos conocimientos con tres proyectos sustanciales: un juego arcade inspirado en Space Invaders, visualizaciones de datos con las bibliotecas superútiles de Python y una sencilla aplicación web que puedes desplegar en línea.

A medida que avance en el *Curso acelerado de Python*, aprenderá a..:

- Utilizar potentes bibliotecas y herramientas de Python, como matplotlib, NumPy y Pygal.
- Crear juegos en 2D que respondan a las pulsaciones del teclado y del ratón, y que sean más difíciles a medida que avanza el juego.
- Trabajar con datos para generar visualizaciones interactivas
- Cree y personalice aplicaciones web sencillas e implántelas en línea de forma segura
- Afronte los errores y las equivocaciones para poder resolver sus propios problemas de programación

Si has estado pensando seriamente en adentrarte en la programación, *Python Crash Course* te pondrá al día y te hará escribir programas reales rápidamente. ¿Por qué esperar más? Arranca el motor y ¡a programar!

CONTRAELAUTOR

Eric Matthes es un profesor de ciencias y matemáticas de secundaria que vive en Alaska, donde imparte un curso introductorio de Python. Lleva escribiendo programas desde que tenía cinco años.

CURSO RÁPIDO DE PYTHON 2 Y 3

EL MEJOR ENTRETENIMIENTO GEEK™
www.nostarch.com

"I L I E FLAT."
Este libro utiliza RepKover, una encuadernación duradera que no se rompe.

ISBN: 978-1-59327-603-4
53995
9 781593 276034
$39.95 ($45.95 CDN)
6 89145 76036 1

ARMARIO EN: LENGUAJES DE PROGRAMACIÓN/ PYTHON