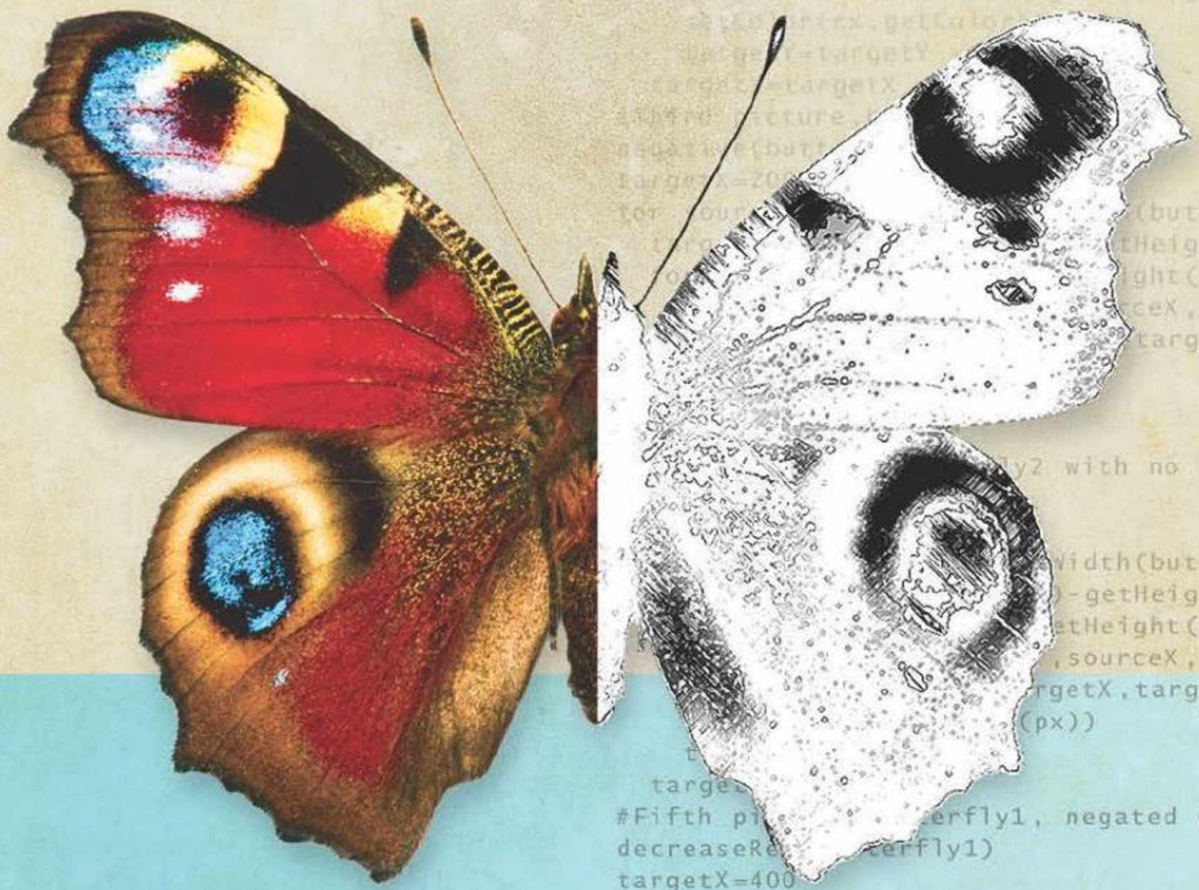
# INTRODUCCIÓN A LA computación y programación con Python®

TERCERA EDICIÓN

UN ENFOQUE MULTIMEDIA



Mark J. Guzdial | Barbara Ericson

ALWAYS LEARNING

PEARSON

# Introducción a la computación y programación con Python®

Tercera Edición

Mark J. Guzdial y Barbara Ericson

*College of Computing/GVU*
*Georgia Institute of Technology*

Traducción
Alfonso Vidal Romero Elizondo
*Ingeniero en Sistemas Electrónicos*
*ITESM, Campus Monterrey*

Revisión técnica
Roberto Martínez Román
*Departamento de Tecnologías de Información y Computación*
*ITESM, Campus Estado de México*

PEARSON

Datos de catalogación bibliográfica

**GUZDIAL, MARK J. y ERICSON, BARBARA**

**Introducción a la computación y programación con Python**

Tercera edición

PEARSON EDUCACIÓN, México, 2013

ISBN: 978-607-32-2049-1

Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 448

Authorized translation from the English language edition, entitled *INTRODUCTION TO COMPUTING AND PROGRAMMING IN PYTHON 3rd* Edition, by *MARK GUZDIAL and BARBARA ERICSON*, published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2013. All rights reserved.

ISBN **9780132923514**

Traducción autorizada de la edición en idioma inglés, titulada *INTRODUCTION TO COMPUTING AND PROGRAMMING IN PYTHON 3ª* edición, por *MARK GUZDIAL y BARBARA ERICSON,* publicada por Pearson Education, Inc., publicada como Prentice Hall, Copyright © 2013. Todos los derechos reservados.

Esta edición en español es la única autorizada.

**Edición en español**

| | |
|---|---|
| Dirección general: | Philip de la Vega |
| Dirección Educación Superior: | Mario Contreras |
| Editor Sponsor: | Luis M. Cruz Castillo |
| | e-mail: luis.cruz@pearson.com |
| Editor de Desarrollo: | Bernardino Gutiérrez Hernández |
| Supervisor de Producción: | Juan José García Guzmán |

Gerencia Editorial

Educación Superior Latinoamérica: Marisa de Anta

TERCERA EDICIÓN, 2013

D.R. © 2013 por Pearson Educación de México, S.A. de C.V.
Atlacomulco No. 500, 5° piso
Col. Industrial Atoto
C.P. 53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

**ISBN VERSIÓN IMPRESA: 978-607-32-2049-1**
**ISBN VERSIÓN E-BOOK: 978-607-32-2050-7**
**ISBN E-CHAPTER: 978-607-32-2051-4**

Impreso en México. Printed in Mexico.
1 2 3 4 5 6 7 8 9 0 – 16 15 14 13

PEARSON

**www.pearsonenespañol.com**

Dedicado a nuestros hijos:
Matthew, Katherine y Jennifer

# Contenido


Novedades en la tercera edición xiii

Prefacio xv

Acerca de los autores xxi

## 1 INTRODUCCIÓN 1

### 1 Introducción a las ciencias computacionales y la computación multimedia 3

1.1 ¿De qué tratan las ciencias computacionales? 3
1.2 Lenguajes de programación 6
1.3 Lo que las computadoras entienden 8
1.4 Computación multimedia: ¿por qué digitalizar los medios? 11
1.5 Ciencias computacionales para todos 12
1.5.1 Es acerca de la comunicación 12
1.5.2 Es acerca del proceso 13

### 2 Introducción a la programación 16

2.1 La programación se refiere a la asignación de nombres 16
2.1.1 Los archivos y sus nombres 18
2.2 La programación en Python 19
2.3 La programación en JES 20
2.4 La computación multimedia y JES 21
2.4.1 Mostrar una imagen 25
2.4.2 Reproducir un sonido 27
2.4.3 Nombrar valores 28
2.5 Creación de un programa 31
2.5.1 Recetas de variables: funciones reales similares a las matemáticas que reciben entradas 34

### 3 Modificación de imágenes mediante el uso de ciclos 40

3.1 Cómo se codifican las imágenes 41

3.2 Manipulación de imágenes 46
3.2.1 Exploración de imágenes 51
3.3 Modificación de los valores de los colores 53
3.3.1 Uso de ciclos en las imágenes 53
3.3.2 Aumentar/reducir el color rojo (verde, azul) 55
3.3.3 Prueba del programa: ¿en verdad funcionó? 60
3.3.4 Cambiar un color a la vez 61
3.4 Creación de un atardecer 62
3.4.1 Comprensión de las funciones 62
3.5 Iluminación y oscurecimiento 67
3.6 Creación de un negativo 68
3.7 Conversión a escala de grises 69

4 Modificación de píxeles en un rango 75
4.1 Copia de píxeles 75
4.1.1 Iterar a través de los píxeles mediante range 76
4.2 Reflejo de una imagen 78
4.3 Copia y transformación de imágenes 86
4.3.1 Copiado 86
4.3.2 Creación de un collage 94
4.3.3 Copiado general 96
4.3.4 Rotación 98
4.3.5 Cambio de escala 100

5 Técnicas de imágenes con selección y combinación 107
5.1 Sustitución de colores: ojo rojo, tonos sepia y posterización 108
5.1.1 Reducción del ojo rojo 111
5.1.2 Imágenes con tono sepia y posterización: uso de condicionales para elegir el color 113
5.2 Combinación de píxeles: difuminado 118
5.3 Comparación de píxeles: detección de bordes 120
5.4 Mezcla de imágenes 122
5.5 Sustracción de fondo 125
5.6 Chromakey 127
5.7 Dibujar sobre imágenes 131
5.7.1 Dibujar mediante comandos de dibujo 132
5.7.2 Representaciones vectoriales y de mapas de bits 134
5.8 Seleccionar sin volver a probar 135
5.9 Programas para especificar el proceso de dibujo 136
5.9.1 ¿Por qué escribimos programas? 139

# 2 SONIDO 143

## 6 Modificación de sonidos mediante ciclos 145

6.1 Cómo se codifica el sonido 145
6.1.1 La física del sonido 145
6.1.2 Exploración de la apariencia de los sonidos 149
6.1.3 Codificación del sonido 151
6.1.4 Números binarios y complemento a dos 153
6.1.5 Almacenamiento de sonidos digitalizados 154
6.2 Manipulación de sonidos 156
6.2.1 Abrir sonidos y manipular muestras 156
6.2.2 Uso de MediaTools de JES 159
6.2.3 Ciclos 161
6.3 Cambiar el volumen de los sonidos 161
6.3.1 Aumentar el volumen 161
6.3.2 ¿De verdad funcionó eso que hicimos? 162
6.3.3 Reducir el volumen 166
6.3.4 Interpretación de las funciones en los sonidos 167
6.4 Normalización de sonidos 167
6.4.1 Generación del recorte 169

## 7 Modificación de muestras en un rango 174

7.1 Manipulación de secciones diferentes del sonido en forma distinta 174
7.2 Empalme de sonidos 177
7.3 Recorte y copia en general 183
7.4 Inversión de sonidos 186
7.5 Reflejo 187
7.6 Sobre las funciones y el alcance 188

## 8 Creación de sonidos mediante la combinación de piezas 193

8.1 Composición de sonidos por medio de la adición 193
8.2 Mezcla de sonidos 195
8.3 Creación de un eco 196
8.3.1 Creación de múltiples ecos 197
8.3.2 Creación de acordes 198
8.4 Cómo funcionan los teclados de muestreo 198
8.4.1 El muestreo como un algoritmo 202
8.5 Síntesis aditiva 203
8.5.1 Creación de ondas senoidales 203

8.5.2 Sumar ondas senoidales 205
8.5.3 Comprobación de nuestro resultado 206
8.5.4 Ondas cuadradas 207
8.5.5 Ondas triangulares 210
8.6 Síntesis de música moderna 211
8.6.1 MP3 212
8.6.2 MIDI 212

9 Creación de programas más grandes 216
9.1 Diseño de programas de arriba hacia abajo (Top-Down) 217
9.1.1 Un ejemplo de diseño arriba-abajo 218
9.1.2 Diseño de la función de nivel superior 219
9.1.3 Escritura de las subfunciones 221
9.2 Diseño de programas de abajo hacia arriba (Bottom-Up) 225
9.2.1 Ejemplo de un proceso abajo-arriba 225
9.3 Prueba de su programa 226
9.3.1 Prueba de las condiciones límite 228
9.4 Tips sobre depuración 228
9.4.1 Averiguar cuál es la instrucción que nos debe preocupar 229
9.4.2 Ver las variables 230
9.4.3 Depuración del juego de aventuras 232
9.5 Algoritmos y diseño 235
9.6 Ejecución de programas fuera de Jes 236

3 TEXTO, ARCHIVOS, REDES, BASES DE DATOS Y UNIMEDIA 242

10 Creación y modificación de texto 244
10.1 Texto como unimedia 244
10.2 Cadenas: creación y manipulación de cadenas 245
10.3 Manipulación de partes de cadenas 248
10.3.1 Métodos de cadenas: introducción a los objetos y la notación punto 248
10.3.2 Listas: texto poderoso y estructurado 251
10.3.3 Las cadenas no tienen fuente 253
10.4 Archivos: lugares en donde puede colocar sus cadenas y otras cosas 253
10.4.1 Apertura y manipulación de archivos 255
10.4.2 Generación de cartas modelo 257
10.4.3 Escritura de programas 257
10.5 La biblioteca estándar de Python 261
10.5.1 Más sobre importaciones y sus propios módulos 262
10.5.2 Otro módulo divertido: random 263
10.5.3 Una muestra de las bibliotecas estándar de Python 265

# 11 Técnicas avanzadas de texto: Web e información 270

11.1 Redes: cómo obtener nuestro texto de la Web 270
11.2 Uso de texto para cambiar de un medio a otro 277
11.3 Cómo mover información entre medios 280
11.4 Uso de listas como texto estructurado para representaciones de medios 282
11.5 Cómo ocultar información en una imagen 284

# 12 Creación de texto para la Web 289

12.1 HTML: la notación de la Web 289
12.2 Escritura de programas para generar HTML 294
12.3 Bases de datos: un lugar para almacenar nuestro texto 299
12.3.1 Bases de datos relacionales 301
12.3.2 Ejemplo de una base de datos relacional mediante el uso de tablas hash 302
12.3.3 Trabajar con SQL 304
12.3.4 Uso de una base de datos para crear páginas Web 307

# 4 PELÍCULAS 311

# 13 Creación y modificación de películas 313

13.1 Generación de animaciones 314
13.2 Trabajar con una fuente de video 322
13.2.1 Ejemplos de manipulación de video 323
13.3 Creación de un efecto de video desde abajo hacia arriba 326

# 5 TEMAS EN CIENCIAS DE LA COMPUTACIÓN 333

# 14 Velocidad 335

14.1 Enfoque en ciencias de la computación 335
14.2 ¿Por qué los programas son veloces? 335
14.2.1 Lo que las computadoras entienden en realidad 336
14.2.2 Compiladores e intérpretes 337
14.2.3 ¿Qué es lo que limita la velocidad de una computadora? 341
14.2.4 ¿En realidad existe una diferencia? 343
14.2.5 Realizar búsquedas más rápidas 345
14.2.6 Algoritmos que nunca terminan o que no pueden escribirse 347
14.2.7 ¿Por qué Photoshop es más veloz que JES? 349
14.3 ¿Por qué las computadoras son veloces? 349
14.3.1 Velocidades de reloj y cálculos reales 350

14.3.2 Almacenamiento: ¿qué hace a una computadora lenta? 351
14.3.3 Pantalla 352

15 Programación funcional 355
15.1 Uso de funciones para facilitar la programación 355
15.2 Programación funcional mediante asociación (map) y reducción (reduce) 358
15.3 Programación funcional para multimedia 362
15.3.1 Manipulación de medios sin cambiar de estado 363
15.4 Recursividad: una idea poderosa 364
15.4.1 Recorridos recursivos de directorios 369
15.4.2 Funciones multimedia recursivas 371

16 Programación orientada a objetos 376
16.1 Historia de los objetos 376
16.2 Trabajo con tortugas 378
16.2.1 Clases y objetos 378
16.2.2 Envío de mensajes a los objetos 379
16.2.3 Los objetos controlan su estado 381
16.3 Enseñar nuevos trucos a las tortugas 383
16.3.1 Redefinición de un método existente de la clase tortuga 385
16.3.2 Uso de tortugas para realizar más acciones 386
16.4 Una presentación orientada a objetos 390
16.4.1 Cómo hacer la clase Diapositiva más orientada a objetos 393
16.5 Medios orientados a objetos 394
16.6 La caja Joe 399
16.7 ¿Por qué objetos? 401

APÉNDICE 407

A Referencia rápida de Python 407
A.1 Variables 407
A.2 Creación de funciones 408
A.3 Ciclos y condicionales 408
A.4 Operadores y funciones de representación 409
A.5 Funciones numéricas 410
A.6 Operaciones de secuencia 410
A.7 Cadenas de escape 410
A.8 Métodos de cadena útiles 410
A.9 Archivos 411
A.10 Listas 411
A.11 Diccionarios, tablas Hash o arreglos asociativos 411

A.12 Módulos externos 411
A.13 Clases 412
A.14 Métodos funcionales 412

Bibliografía 413

Índice 417

# Novedades en la tercera edición

En esta tercera edición, nos enfocamos en realizar correcciones, actualizaciones y mejoras que los profesores y varios revisores identificaron al usar el libro. Sobresalen las siguientes.

1. Una actualización en general de las referencias; por ejemplo, Netscape Navigator y los discos duros de 40 Gb son conceptos que ya no se usan.
2. Introdujimos más términos de ciencias de la computación (brevemente) en los primeros capítulos, como *algoritmo*, *identificador*, *alcance local* y *alcance global*.
3. Una presentación más detallada de las instrucciones de condición en los primeros capítulos, incluyendo `else` y `elif`.
4. Una sección sobre funciones y parámetros, con la explicación de cuándo usar retorno y cuándo no es necesario.
5. Una explicación más profunda sobre la forma como funcionan las variables, en especial respecto a los objetos.
6. Más información sobre el reflejo de imágenes, con un ejemplo más generalizado.
7. Actualizamos los ejemplos Web con referencias para acceder a sitios interesantes y modernos.
8. Más información sobre las diferencias entre los formatos de imágenes.
9. Eliminamos algunos de los ejemplos más elementales de la tortuga en el capítulo 16 y agregamos un par de ejemplos más sofisticados sobre tortugas.
10. Actualizamos la sección sobre hardware y redes para referirnos al hardware más reciente, incluyendo los procesadores multinúcleo y los teléfonos móviles.
11. Una definición más clara de lo que puede hacerse en Jython y CPython, en comparación con JES.
12. Agregamos otro ejemplo relacionado con la esteganografía.
13. Agregamos figuras y explicaciones adicionales en las áreas que los revisores consideraron confusas para los estudiantes.

MARK GUZDIAL Y BARBARA ERICSON
*Georgia Institute of Technology*

## AGRADECIMIENTOS

Ofrecemos nuestros más sinceros agradecimientos a los siguientes revisores:

- Joseph Oldham, Centre College
- Lukasz Ziarek, Purdue University
- Joseph O'Rourke, Smith College
- Atul Prakash, University of Michigan
- Noah D. Barnette, Virginia Tech
- Adelaida A. Medlock, Drexel University
- Susan E. Fox, Macalester College
- Daniel G. Brown, University of Waterloo
- Brian A. Malloy, Clemson University
- Renee Renner, California State University, Chico

# Prefacio

La investigación en la educación computacional deja ver en claro que no sólo se "aprende a programar", sino que también se aprende a programar *algo* [5, 22], y la motivación para hacer ese algo puede marcar la diferencia entre aprender y no aprender a programar [8]. El desafío para cualquier profesor es elegir *algo* que sea un motivador con el suficiente poder.

Las personas quieren comunicarse. Somos seres sociales y el deseo de comunicarnos es una de nuestras principales motivaciones. La computadora se utiliza cada vez más como herramienta de comunicación; incluso más que como herramienta de cálculo. Casi todo el texto, las imágenes, los sonidos, la música y las películas que se publican actualmente se preparan usando tecnología de computadora.

Este libro trata sobre cómo enseñar a las personas a programar para comunicarse a través de medios digitales. El libro se enfoca en cómo manipular imágenes, sonido, texto y películas tal como lo harían los profesionales, pero con programas escritos por estudiantes. Sabemos que la mayoría de las personas utilizará aplicaciones de nivel profesional para realizar este tipo de manipulaciones. Pero, saber *cómo* escribir sus propios programas significa que *puede* hacer más de lo que su aplicación actual le permite; de esta manera su poder de expresión no estará limitado por su software de aplicación.

También puede darse el caso de que al saber cómo funcionan los algoritmos en las aplicaciones de medios pueda utilizarlas mejor; o pasar de una aplicación a la siguiente con mayor facilidad. Si su enfoque en una aplicación está en saber lo que hace cada menú, todas las aplicaciones son distintas. Pero si su enfoque está en mover o colorear los píxeles en la forma que desea, tal vez sea más sencillo ir más allá de los elementos de menú y enfocarse en lo que desea.

Este libro no trata sólo de la programación en los medios. Los programas de manipulación de medios pueden ser difíciles de escribir o tal vez se comporten en formas inesperadas. Surgen preguntas como: "¿Por qué el mismo filtro de imágenes es más rápido en Photoshop?", y "Eso fue difícil de depurar. ¿Acaso hay formas de escribir programas que sean *más fáciles* de depurar?". Responder a cuestionamientos como los anteriores es lo que hacen los científicos computacionales. Al final del libro hay varios capítulos que tratan sobre *computación* y no sólo sobre programación. Estos capítulos van más allá de la manipulación de medios para tratar temas más generales.

La computadora es el dispositivo más sorprendentemente creativo que los humanos hayan concebido jamás. Está compuesta en su totalidad por genialidad. La noción "No sólo lo pienses, hazlo" es realmente posible con una computadora. Si puedes imaginarlo, puedes hacerlo "real" en una computadora. Jugar con la programación puede y *debería* ser una gran diversión.

## OBJETIVOS, METODOLOGÍA Y ORGANIZACIÓN

El contenido curricular de este libro cumple con los requisitos de la metodología "imperative-first" (imperativo primero) descrita en el documento de normas *Computing Curriculum 2001* de la ACM/IEEE [4]. El libro empieza con un enfoque en las construcciones de programación fundamentales: asignaciones, operaciones secuenciales, iteración, condicionales

y funciones de definición. Se hace un énfasis posterior en las abstracciones (por ejemplo: complejidad algorítmica, eficiencia de los programas, organización computacional, descomposición jerárquica, recursividad y programación orientada a objetos), después de que los estudiantes tengan un contexto para comprenderlas.

Este orden inusual se basa en los hallazgos de la investigación en las ciencias del aprendizaje. La memoria es asociativa. Recordamos nuevas cosas dependiendo de con qué las asociemos. Las personas pueden aprender conceptos y habilidades con base en la premisa de que serán de utilidad algún día, pero esos conceptos y habilidades estarán relacionados sólo con las premisas. El resultado se describe como "conocimiento frágil" [9]: el tipo de conocimiento que nos ayuda a aprobar el examen pero que olvidamos muy pronto, ya que no se relaciona con nada más que lo que vimos en esa clase.

Los conceptos y las habilidades se recuerdan mejor si es posible relacionarlos con muchas ideas diferentes, o con ideas que aparecen en nuestra vida diaria. Si queremos que los estudiantes obtengan un conocimiento *transferible* (conocimiento que puede aplicarse en situaciones nuevas), tenemos que ayudarlos a relacionar el nuevo conocimiento con problemas más generales, de modo que las memorias se ordenen en formas que se asocien con esos tipos de problemas [26]. En este libro enseñamos con experiencias concretas que los estudiantes pueden explorar y con las que pueden relacionarse (por ejemplo, condicionales para eliminar los ojos rojos en las imágenes), para posteriormente colocar abstracciones sobre éstas (por ejemplo, obtener el mismo objetivo usando recursividad o filtros y mapas funcionales).

Sabemos que el hecho de partir de las abstracciones en realidad no funciona para los estudiantes de computación. Ann Fleury ha demostrado en cursos introductorios de computación que los estudiantes simplemente no aceptan lo que les decimos sobre el encapsulamiento y la reutilización (por ejemplo [13]). Ellos prefieren código más simple que puedan rastrear con facilidad, y en realidad piensan que dicho código es *mejor*. Se requiere tiempo y experiencia para que se den cuenta de que los sistemas bien diseñados tienen cierto valor. Sin la experiencia es muy difícil para ellos aprender las abstracciones.

La metodología de **computación en los medios** que se utiliza en este libro parte de las razones por las que muchas personas utilizan las computadoras: manipulación de imágenes, exploración de música digital, ver y crear páginas Web, y realizar videos. Después explicamos la programación y la computación en términos de estas actividades. Queremos que los estudiantes visiten Amazon, por ejemplo, y piensen, "Aquí está un sitio Web de catálogo, y sé que este tipo de sitios se implementan con una base de datos y un conjunto de programas que dan formato a las entradas de la base de datos como páginas Web". Queremos que los estudiantes usen Adobe Photoshop® y GIMP, y piensen cómo es que sus filtros de imágenes están manipulando realmente los componentes rojo, verde y azul de los píxeles. Al partir de un contexto relevante es más probable que se realice la transferencia de conocimiento y de habilidades. Esto también hace a los ejemplos más interesantes y motivadores, lo que ayuda a mantener a los estudiantes en la clase.

La metodología de la computación de medios invierte cerca de dos terceras partes del tiempo en dar a los estudiantes experiencias con una variedad de medios en contextos que los motiven. No obstante, pasado ese tiempo empiezan a hacer cuestionamientos sobre *computación*: "¿Por qué Photoshop es más rápido que mi programa?"; "El código de una película es lento. ¿Qué tan lentos se hacen los programas?", son preguntas comunes. Por eso introdujimos las abstracciones y las perspectivas valiosas de la ciencia computacional que responden a *sus* preguntas. Lo que se trata en la última parte de este libro.

Un organismo distinto de investigación en la educación computacional explora por qué son tan elevadas las tasas de abandono o fracaso en los cursos introductorios de computación. Una respuesta común es que esos cursos parecen "irrelevantes" y se enfocan inne-

cesariamente en "detalles tediosos", como la eficiencia [31, 1]. Los estudiantes perciben el contexto de comunicaciones como relevante (como nos lo indican en las encuestas y entrevistas [15, 27]). El contexto relevante forma parte de la explicación del éxito que hemos tenido con la retención en el curso de Georgia Tech, para el que se escribió este libro.

La entrada tardía de la abstracción no es el único orden inusual en esta metodología. Empezamos a usar arreglos y matrices en el capítulo 3, en nuestros primeros programas importantes. Por lo general, los cursos introductorios de computación presentan los arreglos en capítulos posteriores, ya que obviamente son más complicados que las variables con valores simples. Un contexto relevante y concreto es muy poderoso [22]. Encontramos que los estudiantes no tienen dificultades para manipular matrices de píxeles en una imagen.

La tasa de estudiantes que abandona los cursos introductorios de computación o que reciben una calificación de D o F (comúnmente conocida como *tasa WDF*) se reporta en el rango de 30 a 50%, o incluso mayor. Una reciente encuesta internacional de tasas de fracasos en los cursos introductorios de computación reportó que la tasa de fracaso promedio en 54 instituciones estadounidenses fue de 33%, y en 17 instituciones internacionales fue de 17% [6]. En Georgia Tech, de 2000 a 2002 tuvimos una tasa WDF promedio de 28% en el curso introductorio requerido para todas las especialidades. Utilizamos la primera edición de este texto en nuestro curso *Introducción a la computación en los medios*. Nuestra primera oferta piloto de ese curso tuvo 121 estudiantes, sin especialidades de computación o ingeniería, y dos terceras partes de los estudiantes eran mujeres. Nuestra tasa WDF fue de 11.5%.

Durante los siguientes dos años (de primavera de 2003 a otoño de 2005), la tasa WDF promedio en Georgia Tech (entre varios instructores y, literalmente, miles de estudiantes) fue de 15% [21]. En realidad, la tasa WDF anterior de 28% y la tasa WDF actual de 15% son incomparables, ya que todas las especialidades tomaron el primer curso y sólo las especialidades de artes liberales, arquitectura y administración tomaron el nuevo curso. Las especialidades individuales tienen cambios mucho más sustanciales. Por ejemplo, las especialidades en administración tuvieron una tasa de 51.5%, de 1999 a 2003, con el primer curso, y una tasa de fracaso de 11.2% en los primeros dos años del nuevo curso [21]. Desde que se publicó la primera edición de este libro, varias escuelas han adoptado y adaptado esta metodología, además de evaluar su resultado. Todas ellas han reportado mejoras impresionantes en las tasas de éxito [37, 36].

## Cómo utilizar este libro

Este libro representa lo que enseñamos en Georgia Tech casi en el mismo orden. Es probable que algunos maestros omitan ciertas secciones (por ejemplo, las referentes a la síntesis aditiva, a MIDI y a MP3), pero probamos todo el contenido aquí mostrado con nuestros estudiantes.

Sin embargo, este material se ha utilizado de muchas otras formas.

- Podría enseñarse una breve introducción a la computación sólo con los capítulos 2 (introducción a la programación) y 3 (introducción al procesamiento de imágenes), tal vez con algo de material de los capítulos 4 y 5. En algunos casos hemos dado talleres de un solo día sobre computación en los medios, usando sólo este material.
- En esencia, los capítulos 6 a 8 replican los conceptos de ciencias computacionales de los capítulos 3 a 5, pero en el contexto de sonidos en lugar de imágenes. La replicación nos parece útil: algunos estudiantes parecen relacionarse mejor con los conceptos de iteración y las condicionales al trabajar con un medio en vez del otro. Además, esto nos brinda la oportunidad de señalar que el mismo **algoritmo** puede tener efectos similares en distintos medios (por ejemplo, ampliar o reducir una imagen y aumentar o disminuir

el tono de un sonido son el mismo algoritmo). Pero sin duda podríamos omitirlo para ahorrar tiempo.

- El capítulo 12 (sobre películas) no introduce nuevos conceptos de programación o de computación. Aunque es motivacional, podríamos omitir el procesamiento de películas para ahorrar tiempo.
- Recomendamos que se estudien al menos algunos de los capítulos de la última unidad para llevar a los estudiantes a pensar sobre computación y programación de una manera más abstracta, pero sin duda no es necesario cubrir *todos* los capítulos.

### Python y Jython

El lenguaje de programación que utilizamos en este libro es Python. Este lenguaje se ha descrito como "seudocódigo ejecutable". Hemos descubierto que tanto los que se especializan en ciencias computacionales como los que no se especializan pueden aprender Python. Puesto que este lenguaje se utiliza en realidad para tareas de comunicaciones (por ejemplo, desarrollo de sitios Web), es un lenguaje relevante para un curso introductorio de computación. Por ejemplo, los anuncios de empleos que se publican en el sitio Web de Python (`http://www.python.org`) indican que compañías como Google e Industrial Light & Magic contratan programadores de Python.

El dialecto específico de Python que se utiliza en este libro es *Jython* (`http://www.jython.org`). Jython *es* Python. Las diferencias entre Python (que por lo general se implementa en C) y Jython (que se implementa en Java) son similares a las diferencias entre dos implementaciones de lenguajes cualesquiera (por ejemplo, las implementaciones de Microsoft y de GNU con respecto a C++): el lenguaje básico es *exactamente* el mismo, con algunas diferencias en las bibliotecas y detalles que la mayoría de los estudiantes nunca observará.

## NOTACIONES TIPOGRÁFICAS

Algunos ejemplos de código de Python se ven así: `x = x + 1`. Algunos ejemplos más extensos se ven así:

```
def holaProgramador():
  print "Hola, programador"
```

Al mostrar algo que el usuario escribe con la respuesta de Python, tendrá una fuente y estilo similares, sólo que la escritura del usuario aparecerá después de un indicador de Python (»>):

```
>>> print 3 + 4
7
```

Los componentes de la interfaz de usuario de JES (Entorno de Jython para estudiantes, por sus siglas en inglés) se especificarán usando una fuente de letras mayúsculas pequeñas, como el elemento de menú GUARDAR y el botón CARGAR.

Hay varios tipos especiales de barras laterales que encontrará en el libro.

**Idea de ciencias computacionales: una idea de ejemplo**

Los conceptos de ciencias computacionales aparecen así.

■

**Error común: un error común de ejemplo**

Las cosas comunes que pueden hacer que su programa falle aparecen así.

■

**Tip de depuración: un tip de depuración de ejemplo**

Si hay una buena forma de evitar que un error se introduzca en sus programas, se resaltará aquí.

■

**Tip de funcionamiento: un tip de funcionamiento de ejemplo**

Las mejores prácticas o técnicas que son de verdadera utilidad se resaltan de esta forma.

■

## RECURSOS PARA EL PROFESOR

Los recursos para el profesor están disponibles, en inglés, en el sitio Web de este libro: `www.pearsonenespañol.com/guzdial`.

- Diapositivas de presentaciones en PowerPoint®.

## RECONOCIMIENTOS

Nuestros más sinceros reconocimientos para las siguientes personas:

- Jason Ergle, Claire Bailey, David Raines y Joshua Sklare, que elaboraron la versión inicial de JES con sorprendente calidad en un tiempo increíblemente corto. A través de los años, Adam Wilson, Larry Olson, Yu Cheung (Toby) Ho, Eric Mickley, Keith McDermott, Ellie Harmon, Timmy Douglas, Alex Rudnick, Brian O'Neill y William Fredrick (Buck) Scharfnorth III han convertido a JES en la útil y aún comprensible herramienta que es en la actualidad.
- Adam Wilson creó las herramientas MediaTools que son tan útiles para explorar sonidos e imágenes, además de procesar video.
- Andrea Forte, Mark Richman, Matt Wallace, Alisa Bandlow, Derek Chambless, Larry Olson y David Rennie ayudaron a crear los materiales del curso. Derek, Mark y Matt crearon muchos de los programas de ejemplo.
- Hubo varias personas que realmente se esforzaron en Georgia Tech por que saliera este libro. Bob McMath, viceproboste en Georgia Tech, y Jim Foley, decano auxiliar de educación en el College of Computing, invirtieron mucho tiempo durante las primeras etapas. Kurt Eiselt trabajó duro para convertir este esfuerzo en algo real, convenciendo

a otros de que lo tomaran en serio. Janet Kolodner y Aaron Bobick estaban emocionados y fomentaron la idea de la computación en los medios para los estudiantes que incursionaban en las ciencias computacionales. Jeff Pierce revisó y nos apoyó en el diseño de las funciones de medios utilizadas en el libro. Aaron Lanterman me proporcionó muchos consejos sobre cómo transmitir el contenido del material digital en forma precisa. Joan Morton, Chrissy Hendricks, David White y todo el personal del GVU Center se aseguraron de que tuviéramos lo necesario y que se cuidaran los detalles para poder terminar este gran esfuerzo. Amy Bruckman y Eugene Guzdial "compraron" tiempo para que Mark pudiera completar la versión final.

- Agradecemos a Colin Potts y a Monica Sweat, quienes impartieron esta clase en Georgia Tech y nos brindaron muchas perspectivas sobre el curso.
- Charles Fowler fue la primera persona fuera de Georgia Tech dispuesta a arriesgarse y probar el curso en su propia institución (Gainesville College), por lo cual le estamos muy agradecidos.
- El curso piloto que se ofreció en la primavera de 2003 en Georgia Tech fue muy importante para hacer mejoras. Andrea Forte, Rachel Fithian y Lauren Rich realizaron la evaluación correspondiente, lo cual fue en extremo valioso para ayudarnos a comprender lo que funcionaba bien y lo que no. Los primeros asistentes de enseñanza (Jim Gruen, Angela Liang, Larry Olson, Matt Wallace, Adam Wilson y Jose Zagal) hicieron mucho para crear esta metodología. Blair MacIntyre, Colin Potts y Monica Sweat ayudaron a que los materiales fueran fáciles de adoptar. Jochen Rick hizo de CoWeb/Swiki un excelente lugar para que los estudiantes de CS1315 pasaran el tiempo.
- Muchos estudiantes señalaron errores y realizaron sugerencias para mejorar el libro. Agradecemos a Catherine Billiris, Jennifer Blake, Karin Bowman, Maryam Doroudi, Suzannah Gill, Baillie Homire, Jonathan Laing, Mireille Murad, Michael Shaw, Summar Shoaib, y en especial a Jonathan Longhitano, quien posee un verdadero instinto para la corrección de textos.
- Agradecemos a nuestros estudiantes de cursos anteriores de *Computación en los medios*: Constantino Kombosch, Joseph Clark y Shannon Joiner por darnos permiso de usar sus instantáneas de la clase en los ejemplos.
- El trabajo de investigación que condujo a este texto se apoyó en concesiones de la National Science Foundation: de la División de educación universitaria, programa CCLI, y del programa Innovaciones educativas del CISE. Agradecemos todo su apoyo.
- También agradecemos a los estudiantes de computación Anthony Thomas, Celines Rivera y Carolina Gomez por permitirnos usar sus imágenes.
- Por último, no por ello menos importante, agradecemos a nuestros hijos Matthew, Katherine y Jennifer Guzdial, que nos permitieron fotografiarlos y grabarlos para el proyecto de medios de mamá y papá, y quienes nos brindaron todo su apoyo y entusiasmo por la clase.

Mark Guzdial y Barbara Ericson
*Georgia Institute of Technology*

# Acerca de los autores

**Mark Guzdial** es profesor en la School of Interactive Computing del College of Computing en el Georgia Institute of Technology. Es uno de los fundadores de la serie de talleres International Computing Education Research de la ACM. La investigación del Dr. Guzdial se enfoca en las ciencias del aprendizaje y la tecnología; específicamente en la investigación sobre educación computacional. Sus primeros libros fueron sobre el lenguaje de programación Squeak y su uso en la educación. Fue el desarrollador original de "Swiki" (Squeak Wiki), el primer wiki desarrollado de manera explícita para usarse en escuelas. Mark ha publicado varios libros sobre el uso de los medios como un contexto para aprender computación, lo cual ha influido en los planes de estudios universitarios de computación alrededor del mundo. Es miembro del Consejo de Educación de la ACM y del Consejo SIGCSE (Grupo de interés especial para la educación en ciencias computacionales) de la ACM. Pertenece a los consejos editoriales de las publicaciones *Journal of the Learning Sciences* y *Communications of the ACM*.

**Barbara Ericson** es científica investigadora y directora de Divulgación informática en el College of Computing de Georgia Tech. Desde 2004 ha estado trabajando para mejorar la educación sobre computación a nivel introductorio.

Ha participado como representante de educación de los maestros en el consejo de la Computer Science Teachers Association, como copresidente de la Alianza K-12 para el National Center for Women in Information Technology, y como lectora para los exámenes avanzados de colocación de Ciencias computacionales. Disfruta la diversidad de problemas con los que ha trabajado a través de sus años en la computación, entre los que se encuentran gráficos por computadora, inteligencia artificial, medicina y programación orientada a objetos. Mark y Barbara recibieron el premio Karl V. Karlstrom de la ACM para educadores sobresalientes de 2010 por su trabajo de la computación en los medios, incluyendo este libro.

PARTE 1

# INTRODUCCIÓN


**Capítulo 1** Introducción a las ciencias computacionales y la computación multimedia

**Capítulo 2** Introducción a la programación

**Capítulo 3** Modificación de imágenes mediante el uso de ciclos

**Capítulo 4** Modificación de píxeles en un rango

**Capítulo 5** Técnicas de imágenes con selección y combinación

CAPÍTULO 1

# Introducción a las ciencias computacionales y la computación multimedia


1.1 ¿DE QUÉ TRATAN LAS CIENCIAS COMPUTACIONALES?

1.2 LENGUAJES DE PROGRAMACIÓN

1.3 LO QUE LAS COMPUTADORAS ENTIENDEN

1.4 COMPUTACIÓN MULTIMEDIA: ¿POR QUÉ DIGITALIZAR LOS MEDIOS?

1.5 CIENCIAS COMPUTACIONALES PARA TODOS


**Objetivos de aprendizaje del capítulo**

- De qué tratan las ciencias computacionales y qué es lo que les preocupa a los científicos computacionales.
- Por qué digitalizamos los medios.
- Por qué es valioso estudiar computación.
- Cuál es el concepto de una **codificación**.
- Cuáles son los componentes básicos de una computadora.

## 1.1 ¿DE QUÉ TRATAN LAS CIENCIAS COMPUTACIONALES?

Las ciencias computacionales comprenden el estudio del **proceso**: la forma en que nosotros o las computadoras hacemos las cosas, cómo especificamos lo que hacemos y cómo especificamos qué es lo que estamos procesando. Ésta es una definición bastante escasa. Veamos ahora una definición metafórica.

Idea de ciencias computacionales: **las ciencias computacionales comprenden el estudio de las recetas**

En este caso, las "recetas" son de un tipo especial: las que pueden ejecutarse mediante un dispositivo computacional, pero este punto sólo es de importancia para los científicos computacionales. El punto importante en general es que una receta de ciencias computacionales define *con exactitud* lo que debe hacerse.

Si lo vemos de una manera más formal, los científicos computacionales estudian *algoritmos*, los cuales son procedimientos paso a paso para realizar una tarea. Cada paso en un algoritmo es algo que una computadora ya sabe cómo hacer (por ejemplo, sumar dos números enteros pequeños) o algo que se puede enseñar a una computadora (por ejemplo, sumar números más grandes, incluso con un punto decimal). A una receta que se puede ejecutar en una computadora se le conoce como *programa*. Un programa es una codificación de un algoritmo en una representación que una computadora pueda leer.

■

Para usar nuestra metáfora un poco más: piense en un algoritmo como la forma paso a paso en la que su abuela hizo su receta secreta. Ella siempre la hizo de la misma forma y tuvo un resultado excelente y muy confiable. Anotar esa receta para poder leerla y hacerla después es como convertir su algoritmo en un programa para usted.

Si usted es un biólogo que desea describir la manera en que funciona la migración o cómo se replica el ADN, entonces es *muy* útil poder escribir una receta que especifique *con exactitud* lo que ocurre, en términos que puedan definirse y comprenderse por completo. Lo mismo aplica si usted es un químico que desea explicar cómo se llega al equilibrio en una reacción. El gerente de una fábrica puede definir un esquema de máquina y banda, e incluso probar su funcionamiento (antes de mover físicamente las cosas pesadas a sus posiciones) mediante el uso de **programas** de computadora. El hecho de poder definir tareas y/o simular eventos con exactitud es una razón importante del porqué las computadoras han cambiado de manera radical, cómo es que la ciencia se lleva a cabo y se comprende.

De hecho, si usted *no puede* escribir una receta para cierto proceso, tal vez en realidad no quiere comprender el proceso, o tal vez el proceso no puede funcionar realmente de la forma en que usted lo tiene pensado. Algunas veces, la acción de tratar de escribir la receta es una prueba por sí sola. Ahora, algunas veces no es posible escribir la receta debido a que el proceso es uno de los pocos que no pueden ejecutarse en una computadora. Hablaremos sobre esos procesos en el capítulo 14.

Tal vez suene gracioso decir que los *programas* son recetas, pero la analogía es contundente. Gran parte de lo que estudian los científicos computacionales puede definirse en términos de recetas.

- Algunos científicos computacionales estudian la forma en que se escriben las recetas. ¿Acaso hay mejores o peores formas de realizar algo? Si alguna vez ha tenido que separar claras de huevo de las yemas, se habrá dado cuenta de que saber cómo hacerlo de la forma correcta marca una enorme diferencia. Los teóricos de ciencias computacionales piensan en las recetas más rápidas y cortas, y las que ocupan la menor cantidad de espacio (puede considerarlas como el espacio en un mostrador: la analogía funciona), o incluso que usan la menor cantidad de energía (lo que es importante cuando operan en dispositivos de baja energía, como los teléfonos celulares). A *la manera* en que funciona una receta, algo completamente distinto a la forma en que está escrita (por ejemplo, en un programa), se le conoce como el estudio de los algoritmos. Los ingenieros de software piensan sobre cómo es que los grupos grandes pueden reunir recetas que todavía funcionan (las recetas para algunos programas, como el que lleva la cuenta de los registros de Visa/MasterCard, ¡tienen literalmente millones de pasos!). El término **software** significa una colección de programas de computadora (recetas) que logran una tarea.
- Otros científicos computacionales estudian las unidades que se utilizan en las recetas. ¿Acaso importa si una receta usa medidas métricas o inglesas? La receta puede funcionar en cualquier caso, pero si no sabemos lo que es una libra o una taza, nos será difícil entenderla. También hay unidades que tienen sentido para ciertas tareas y para otras no, pero si podemos ajustar las unidades a las tareas será más fácil explicarlas y podremos realizar las cosas con mayor rapidez, además de evitar errores. ¿Alguna vez se ha preguntado por qué los barcos en el mar miden su velocidad en *nudos*? ¿Por qué no usar algo como metros por segundo? Algunas veces, en ciertas situaciones especiales —por ejemplo, en un barco en alta mar— los términos más comunes no son apropiados o no funcionan tan bien como deberían. O podemos inventar nuevos tipos de unidades, como una unidad que represente a todo un programa o una computadora, o una red como la de sus amigos y los amigos de sus amigos en Facebook. Al estudio

de las unidades de ciencias computacionales se le conoce como **estructuras de datos**. Los científicos computacionales que estudian formas de llevar el registro de muchos datos en muchos tipos distintos de unidades estudian las **bases de datos**.

- ¿Se pueden escribir recetas para cualquier cosa? ¿Acaso hay recetas que *no* puedan escribirse? Los científicos computacionales saben que hay recetas que no pueden escribirse. Por ejemplo, no podemos escribir una receta que pueda indicar con absoluta certeza si alguna otra receta funcionará o no. ¿Qué hay sobre la *inteligencia*? ¿Podemos escribir una receta de tal forma que una computadora que la siga pudiera realmente estar *pensando* (y cómo podríamos saber si salió bien)? Los científicos computacionales en **teoría**, **sistemas inteligentes**, **inteligencia artificial** y **sistemas** se preocupan por este tipo de cosas.
- Incluso hay científicos computacionales que se enfocan en saber si a las personas les *gusta* lo que producen las recetas, algo parecido a las críticas de restaurantes en un periódico. Algunos de éstos son especialistas en **interfaces humano-computadora** que se preocupan por saber si las personas pueden comprender y hacer uso de las recetas ("recetas" que producen una *interfaz* que las personas utilizan, como ventanas, botones, barras de desplazamiento y otros elementos de lo que consideramos un programa en ejecución).
- Así como algunos chefs se especializan en ciertos tipos de recetas, como las crepas o la carne asada, los científicos computacionales también se especializan en ciertos tipos de recetas. Los científicos computacionales que trabajan en los *gráficos* están enfocados principalmente en las recetas que producen imágenes, animaciones e incluso películas. Los científicos computacionales que trabajan en la *música por computadora* se concentran en las recetas que producen sonidos (que a menudo son melódicos, pero no siempre).
- Hay además otros científicos computacionales que estudian las *propiedades emergentes* de las recetas. Piense en World Wide Web. Esta red es en realidad una colección de *millones* de recetas (programas) que se comunican entre sí. ¿Por qué una sección de la Web se volvería más lenta en cierto punto? Es un fenómeno que emerge de estos millones de programas, sin duda algo que no se tenía planeado. Esto es algo que los científicos computacionales de **redes** estudian. Lo verdaderamente sorprendente es que estas propiedades emergentes (el hecho de que empiezan a ocurrir cosas cuando se tienen muchas, muchas recetas interactuando al mismo tiempo) también pueden usarse para explicar cosas no relacionadas con la computación. Por ejemplo, la forma en que las hormigas buscan alimento o el hecho de que las termitas fabriquen montículos también puede describirse como algo que simplemente ocurre cuando se tienen muchos programas pequeños haciendo algo simple e interactuando. En la actualidad existen científicos computacionales que estudian la forma en que la Web permite nuevos tipos de interacciones, en especial en grupos extensos (como Facebook o Twitter). Los científicos computacionales que estudian la *computación social* están interesados en cómo funcionan estos tipos de interacciones y en las características del software que son más exitosas para promover interacciones sociales de utilidad.

La metáfora de la receta también funciona en otro nivel. Todos sabemos que es posible cambiar algunas cosas en una receta sin que el resultado cambie de manera considerable. Siempre podemos aumentar todas las unidades por medio de un multiplicador (por decir, el doble) para crear más. Siempre es posible agregar más ajo u orégano a la salsa de espagueti. Pero hay algunas cosas que no podemos cambiar en una receta. Si la receta requiere polvo para hornear, no podemos sustituirlo por bicarbonato de sodio. Si se supone que debemos hervir las empanadillas y luego sofreírlas, es probable que invertir el orden de estas acciones no dé un buen resultado (figura 1.1).

**POLLO CACCIATORE**

3 pechugas de pollo enteras con hueso
1 cebolla mediana, picada
1 cucharada de ajo picado
2 cucharaditas y después ¼ de taza de aceite de oliva
1 ½ cucharaditas de harina
¼ de cucharadita de sal para sazonar Lawry's
1 pimiento picado (opcional) de cualquier color
1 lata (28 oz) de tomates picados
1 lata (15 oz) de salsa de tomate
1 lata (6.5 oz) de champiñones
1 lata (6 oz) de pasta de tomate
½ tarro (26 oz) de salsa de espagueti
3 cucharaditas de sazonador italiano
1 cucharada de polvo de ajo (opcional)

Corte el pollo en cuadros de aproximadamente 1 pulgada. Sofría la cebolla y el ajo hasta que la cebolla esté transparente. Mezcle la harina y la sal de Lawry's. Lo ideal es una proporción entre 1:4 y 1:5 de sal para sazonar y de harina, y una cantidad suficiente de esta mezcla para cubrir el pollo. Coloque el pollo cortado y la harina sazonada en una bolsa, y agite para cubrir. Agregue el pollo cubierto a la cebolla y el ajo. Mueva con frecuencia la mezcla hasta que el pollo esté dorado.

Tendrá que agregar aceite para evitar que se pegue o se queme: algunas veces yo agrego ¼ de taza de aceite de oliva. Agregue los tomates, la salsa, los champiñones y la pasta (junto con los pimientos opcionales). Mezcle bien. Agregue el sazonador italiano. Como también me gusta el ajo, por lo general también agrego polvo de ajo. Mezcle bien. Debido a toda la harina, la salsa puede hacerse muy espesa. Por lo general la adelgazo con la salsa de espagueti, hasta ½ tarro. Hierva a fuego lento de 20 a 30 minutos.

**FIGURA 1.1**
Una receta de cocina: en todo momento podemos duplicar los ingredientes, pero si agregamos una taza adicional de harina no se adelgazará, y ¡no intente dorar el pollo *después* de agregar la salsa de tomate!

Lo mismo aplica para las recetas de software. Por lo general hay cosas que podemos cambiar con facilidad: los nombres de las cosas (aunque es conveniente cambiar los nombres de manera consistente), algunas de las **constantes** (números que aparecen como simples números y no como variables) y tal vez hasta algunos de los **rangos** de datos (secciones de éstos) que se van a manipular. Sin embargo, el orden de los comandos que se envían a la computadora siempre tiene que quedar exactamente como se indica. A medida que avancemos, aprenderá qué es lo que podemos cambiar con seguridad y qué es lo que no debemos modificar.

## 1.2 LENGUAJES DE PROGRAMACIÓN

Los científicos computacionales escriben las recetas en un **lenguaje de programación** (figura 1.2). Se utilizan distintos lenguajes de programación para diferentes propósitos. Algunos de ellos son muy populares, como Java y C++. Otros son menos conocidos, como Squeak y T. Algunos otros están diseñados para facilitar en gran medida el aprendizaje de las ideas de ciencias computacionales, como Scheme o Python, pero el hecho de que sean fáciles de aprender no siempre los hace muy populares ni tampoco la mejor opción para los expertos que construyen recetas más grandes o complicadas. Al enseñar ciencias computacionales es difícil balancear el hecho de elegir un lenguaje fácil de aprender *y* que sea popular, además de ser lo suficientemente útil para los expertos como para que los estudiantes se vean motivados a aprenderlo.

¿Por qué los científicos computacionales simplemente no usan los lenguajes humanos naturales, como inglés y español? El problema es que los lenguajes naturales evolucionaron de la forma en que lo hicieron para mejorar las comunicaciones entre seres muy inteligentes: los humanos. Como explicaremos con más detalle en la siguiente sección, las computadoras

**Python/Jython**

```
def hello():
  print "Hola mundo"
```

**Java**

```
class HolaMundo (
  static public void main( String args[] ) {
    System.out.println( "Hola mundo" );
  }
}
```

**C++**

```
#include <iostream.h>
main() {
    cout << "Hola mundo" << endl;
    return 0;
}
```

**Scheme**

```
(define holamundo
  (lambda ()
          (display "Hola mundo")
          (newline)))
```

**FIGURA 1.2**
Comparación de lenguajes de programación: una tarea de programación simple y común es imprimir las palabras "Hola mundo" en la pantalla.

son excepcionalmente tontas. Necesitan un nivel de especificidad que el lenguaje natural no puede ofrecer. Además, lo que decimos a otra persona en una comunicación natural no es exactamente lo que decimos en una receta computacional. ¿Cuándo fue la última vez que dijo a alguien cómo funcionaba un videojuego como *Halo*, *Quake* o *Call of Duty* con tantos detalles minuciosos como para que pudiera replicar el juego (por decir, en papel)? Un idioma "humano" como el inglés o el español no es bueno para ese tipo de tarea.

Existen una gran variedad de lenguajes de programación debido a que hay una gran variedad de recetas para escribir. Los programas escritos en el lenguaje de programación *C* tienden a ser muy rápidos y eficientes, pero también tienden a ser difíciles de leer y de escribir; además requieren unidades que tratan más sobre las computadoras que sobre las migraciones de aves, o ADN, o cualquier otra cosa sobre la que quiera usted escribir su receta. El lenguaje de programación *Lisp* (y los lenguajes relacionados como Scheme, T y Common Lisp) es muy flexible y adecuado para explorar la forma de escribir recetas que no se hayan escrito antes, pero Lisp tiene una *apariencia* muy extraña en comparación con lenguajes como C, por lo que muchas personas lo evitan y, como consecuencia natural, hay pocas personas que lo conocen. Si desea contratar a cien programadores para que trabajen en su proyecto, será más sencillo encontrar cien programadores que conozcan un lenguaje popular que uno menos popular, aunque esto no significa que el lenguaje popular sea el mejor para su tarea.

El lenguaje de programación que usaremos en este libro es **Python** (visite `http://www.python.org` para obtener más información). Python es un lenguaje bastante popular que se utiliza con mucha frecuencia para la programación Web y multimedia. El motor de búsqueda

Web *Google* usa Python. La compañía de medios *Industrial Light & Magic* también usa Python. Hay una lista de compañías que usan Python disponible en `http://wiki.python.org/moin/OrganizationsUsingPython`. Este lenguaje es muy sencillo de aprender, fácil de leer y muy flexible, pero no muy eficiente. Es probable que el mismo algoritmo codificado en C y en Python sea más rápido en C. Python es un buen lenguaje para escribir programas que funcionan dentro de una aplicación, como el lenguaje de manipulación de imágenes GIMP (`http://www.gimp.org`) o la herramienta de creación de contenido 3D llamada Blender (`http://www.blender.org`).

La versión de Python que usamos en este libro se llama **Jython** (`http://www.jython.org`). Por lo general, Python se implementa en el lenguaje de programación C. Jython es Python implementado en *Java*: esto significa que en realidad Jython es un programa escrito en Java. Jython nos permite hacer multimedia que funciona en múltiples plataformas computacionales. Jython es un lenguaje de programación real que se utiliza en muchos proyectos. Es una forma de Python. Puede descargar una versión de Jython para su computadora desde el sitio Web de Jython, que funcionará para todo tipo de propósitos.

Usaremos Jython en este libro por medio de un *entorno* de programación especial conocido como **JES** (*Jython Environment for Students: Entorno de Jython para estudiantes*), el cual se desarrolló para facilitar la programación en Jython. JES cuenta con algunas características para trabajar con multimedia, como visores de sonidos e imágenes. JES también tiene integradas ciertas funciones especiales para manipular medios digitales, las cuales están disponibles para el programador sin necesidad de hacer nada especial. Todo lo que puede hacer en JES también puede hacerse en Jython normal, aunque tendría que incluir de manera explícita las bibliotecas especiales. La mayoría de los programas escritos en Jython funcionan también en Python.

Ahora revisemos dos de los términos más importantes que utilizaremos en este libro:

- Un **programa** es una descripción, en un lenguaje de programación, de un proceso que obtiene cierto resultado útil para alguien. Un programa puede ser pequeño (como el que implementa una calculadora) o enorme (como el que usa su banco para rastrear todas sus cuentas).
- Un **algoritmo** (en contraste) es la descripción de un proceso paso a paso que no está enlazado con ningún lenguaje de programación. El mismo algoritmo puede implementarse en muchos lenguajes distintos y de muchas formas diferentes en muchos programas distintos, aunque todos serían el mismo **proceso** si hablamos sobre el mismo algoritmo.

El término **receta**, según la forma en que se utiliza en este libro, describe programas o porciones de programas que hacen algo. Utilizaremos el término *receta* para enfatizar las *piezas de un programa que realizan una tarea útil* **relacionada con multimedia**.

## 1.3 LO QUE LAS COMPUTADORAS ENTIENDEN

Las recetas computacionales se escriben para ejecutarse en computadoras. ¿Cómo sabe la computadora qué hacer? ¿Qué podemos decirle a la computadora que haga en la receta? La respuesta es, "muy, muy poco". Las computadoras son exageradamente estúpidas. En realidad sólo conocen los números.

En realidad, incluso decir que las computadoras *conocen* los números no es del todo correcto. Las computadoras usan **codificaciones** de números. Son dispositivos electrónicos que reaccionan a los voltajes en los cables. Cada cable se denomina **bit**. Si un cable tiene voltaje, decimos que codifica un 1. Si no tiene voltaje, decimos que codifica un 0. Agrupamos estos cables (bits)

en conjuntos. Un conjunto de 8 bits se conoce como **byte**. Así, partiendo de un conjunto de ocho cables (un byte), tenemos un patrón de ocho ceros y unos, por ejemplo: 01001010. Si usamos el sistema numérico **binario** podemos interpretar este byte como un **número** (figura 1.3). Aquí es de donde sacamos la afirmación de que una computadora conoce los números.[1]

Una computadora tiene una **memoria** llena de bytes. Todo con lo que una computadora trabaja en un instante dado se almacena en su memoria. Esto significa que todo con lo que trabaja una computadora se *codifica* en sus bytes: imágenes JPEG, hojas de cálculo de Excel, documentos de Word, molestos anuncios emergentes en Web, y el correo electrónico basura más reciente.

Una computadora puede hacer muchas cosas con los números. Puede sumarlos, restarlos, multiplicarlos, dividirlos, ordenarlos, recolectarlos, duplicarlos, filtrarlos (por ejemplo, "Haz una copia de estos números, pero sólo los pares"), compararlos y hacer cosas con base en la comparación. Por ejemplo, podemos decir lo siguiente a una computadora en una receta: "Compara estos dos números. Si el primero es menor que el segundo, salta hasta el paso 5 en esta receta. De lo contrario, continúa con el siguiente paso".

Hasta ahora parece que la computadora es un tipo de calculadora elegante, y sin duda ésa es la razón por la que se inventó. Uno de los primeros usos de una computadora fue para calcular las trayectorias de los proyectiles durante la Segunda Guerra Mundial ("Si el viento proviene del SE a 15 mph y queremos golpear un blanco a 0.5 millas de distancia, a un ángulo de 30 grados al este del norte, entonces incline su lanzador a..."). Las computadoras modernas pueden realizar miles de millones de cálculos por segundo. Pero lo que hace a la computadora útil para recetas generales es el concepto de las *codificaciones*.

**Idea de ciencias computacionales: las computadoras pueden extender capas de codificaciones**

Las computadoras pueden extender capas de codificaciones a casi cualquier nivel de complejidad. Los números pueden interpretarse como caracteres, que a su vez pueden interpretarse en conjuntos como páginas Web, lo cual puede interpretarse que aparece como múltiples fuentes y estilos. Pero en el nivel más inferior, la computadora *sólo* "conoce" de voltajes, los cuales interpretamos como números. Las codificaciones nos permiten olvidarnos sobre los detalles de nivel inferior. Las codificaciones son un ejemplo de *abstracción*, con lo cual obtenemos nuevos conceptos para usar que nos permiten ignorar otros detalles.

■

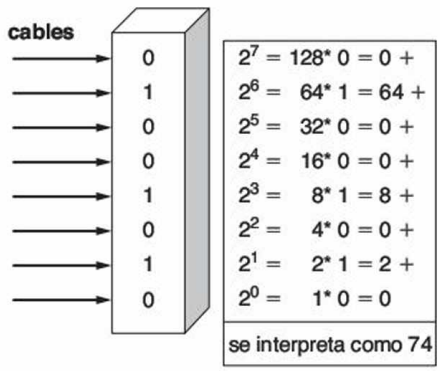



**FIGURA 1.3**
Ocho cables con un patrón de voltajes forman un byte, el cual se interpreta como un patrón de ocho ceros y unos, lo que a su vez se interpreta como un número decimal.

[1]Hablaremos más sobre este nivel de la computadora en el capítulo 14.

Si uno de estos bytes se interpreta como el número 65, podría tan sólo ser el número 65. O podría ser la letra *A* si usamos una codificación estándar de números a letras, conocida como *Código estándar estadounidense para el intercambio de información* (**ASCII**, por sus siglas en inglés). Si el 65 aparece en una colección de otros números que interpretamos como texto, y se encuentra en un archivo que termina en ".html", podría ser parte de algo parecido a `<a href...`, que un navegador Web interpretará como la definición de un vínculo. Si bajamos al nivel de la computadora, esa *A* es sólo un patrón de voltajes. Si subimos varias capas hasta el nivel de un navegador Web, esa A define algo en lo que podemos hacer clic para obtener más información.

Si la computadora sólo entiende números (y eso ya es bastante exigencia), ¿cómo manipula estas codificaciones? Seguro, sabe cómo comparar números, pero ¿cómo es que a partir de esto es posible ordenar la lista de una clase? Por lo general, cada capa de codificación se implementa como una pieza o capa en el software. Hay software que entiende cómo manipular los caracteres. El software de caracteres sabe cómo hacer cosas tales como comparar nombres, ya que ha codificado que la *a* va antes que la *b* y así en lo sucesivo, y que la comparación numérica del orden de los números en la codificación de las letras conduce a comparaciones alfabéticas. El software de caracteres es utilizado por otro software que manipula el texto en los archivos. Ésa es la capa que algunos programas como Microsoft Word, el Bloc de notas o TextEdit usarían. Otra pieza de software sabe cómo interpretar *HTML* (el lenguaje de Web) y otra capa del mismo software sabe cómo tomar el HTML y mostrar el texto, las fuentes, los estilos y los colores correctos.

De manera similar podemos crear capas de codificaciones en la computadora para nuestras tareas específicas. Podemos enseñar a una computadora cuáles son las células que contienen mitocondrias y ADN, que el ADN tiene cuatro tipos de nucleótidos y que las fábricas tienen ciertos tipos de prensas y estampillas. La creación de capas de codificación e interpretación, de modo que la computadora trabaje con las unidades correctas (recordemos nuestra analogía de las recetas) para un problema dado, corresponde a la **representación de datos**, o al proceso de definir las **estructuras de datos** correctas.

Si esto suena como mucho software, lo es. Cuando el software se distribuye en capas de esta forma, hace que la computadora se vuelva un poco lenta. Pero lo poderoso sobre las computadoras es que son *sorprendentemente* rápidas, ¡y se están volviendo cada vez más rápidas todo el tiempo!

**Idea de ciencias computacionales: la ley de Moore**

Gordon Moore, uno de los fundadores de Intel (fabricante de chips de procesamiento de computadoras), sostenía que el número de transistores (un componente clave de las computadoras) se duplicaría al mismo precio cada 18 meses, lo que en efecto significaba que la misma cantidad de dinero compraría hasta el doble del poder de cómputo cada 18 meses. Esto significa que las computadoras se vuelven cada vez más pequeñas, rápidas y económicas. Esta ley se ha cumplido durante décadas.

■

En la actualidad, las computadoras pueden ejecutar literalmente *miles de millones* de pasos de recetas por segundo. Pueden contener en un su memoria enciclopedias de datos, en sentido literal. Nunca se cansan ni se aburren. ¿Necesita buscar un tarjetahabiente de entre un millón de clientes? ¡No hay problema! ¿Busca el conjunto correcto de números para obtener el mejor valor de una ecuación? ¡Pan comido!

¿Procesa millones de elementos de imágenes, fragmentos de sonido o cuadros de película? A esto se le conoce como **computación multimedia**. En este libro escribirá recetas que mani-

pulan imágenes, sonidos, texto e incluso otras recetas. Esto es posible debido a que todo en la computadora se representa en forma digital, incluso las recetas. No podríamos realizar computación en los medios si éstos no se representaran en forma digital. Al terminar de leer el libro habrá escrito recetas para implementar efectos especiales de video digital, que crean páginas Web de la misma forma que lo hacen Amazon y eBay, y que filtran imágenes como Photoshop.

## 1.4 COMPUTACIÓN MULTIMEDIA: ¿POR QUÉ DIGITALIZAR LOS MEDIOS?

Consideremos una codificación que sería apropiada para las imágenes. Imagine que las imágenes están compuestas de pequeños puntos. Eso no es difícil de imaginar: analice de cerca su monitor o una pantalla de TV, y verá que sus imágenes *ya están* compuestas de pequeños puntos. Cada uno de estos puntos es de un color distinto. La física nos dice que los colores pueden describirse como la suma de *rojo*, *verde* y *azul*. Al sumar el rojo y el verde se obtiene amarillo. Mezcle los tres y obtendrá blanco. Desactívelos todos y tendrá un punto negro.

¿Qué pasaría si codificáramos cada punto en una imagen como una colección de tres bytes, uno para cada cantidad de rojo, verde y azul en ese punto en la pantalla? ¿Y si recolectamos varios de estos conjuntos de tres bytes para determinar todos los puntos de una imagen dada? Ésta es una forma bastante razonable de representar las imágenes, y en esencia es como lo haremos en el capítulo 3.

Para manipular estos puntos (a cada uno se le conoce como **pixel** o *elemento de imagen*) se puede requerir mucho procesamiento. Existen miles, o incluso millones de estos puntos en una imagen con la que podríamos trabajar en la computadora o en la Web. Pero la computadora no se aburre, además de que es muy rápida.

La codificación que usaremos para el sonido involucra 44 100 conjuntos de dos bytes (lo que se conoce como muestra) para cada *segundo* de tiempo. Una canción de tres minutos requiere 158 760 000 bytes (y el doble para sonido estéreo). Para realizar cualquier procesamiento con estos datos se necesitan *muchas* operaciones. Pero a mil millones de operaciones por segundo, podemos hacer muchas operaciones con cada uno de esos bytes en sólo unos cuantos momentos.

Para crear codificaciones de este tipo para los medios se requiere un cambio en los medios. Observe el mundo real: no está compuesto por miles de pequeños puntos que pueda ver. Escuche un sonido: ¿acaso puede oír miles de pequeños bits de sonido por segundo? El hecho de que *no pueda* escuchar pequeños bits de sonido por segundo es lo que hace posible crear estas codificaciones. Nuestros ojos y oídos están limitados: sólo podemos percibir hasta cierto punto, y sólo cosas hasta cierto tamaño. Si descompone una imagen en puntos que sean lo bastante pequeños, sus ojos no podrán detectar el hecho de que no sea un flujo continuo de color. Si descompone un sonido en piezas lo bastante pequeñas, sus oídos no podrán detectar que el sonido no es un flujo continuo de energía auditiva.

El proceso de codificar medios en pequeñas piezas se denomina **digitalización**, lo que algunas veces se conoce como "*volverse digital*". *Digital* significa (de acuerdo con los *Diccionarios Longman*) "Proporcionar información en forma de números; en relación con los dedos de las manos y los pies".[2] El proceso de convertir las cosas a digitales trata sobre convertir cosas de lo continuo e incontable en algo que podamos contar, como si fuera con nuestros dedos.

Los *medios digitales*, cuando se hacen bien, se sienten igual para nuestro aparato sensorial humano limitado, como si fueran el original. Las grabaciones fonográficas (¿alguna

[2]Definición de "digital" de LONGMAN DICTIONARY OF CONTEMPORARY ENGLISH. Copyright © 2009. Pearson Education. Reimpreso con permiso.

vez ha visto una?) capturan el sonido en forma continua, como una señal **análoga**. Las fotografías (en película) capturan la luz como un flujo continuo. Algunas personas dicen que pueden escuchar la diferencia entre las grabaciones fonográficas y las grabaciones en CD; pero para nuestros oídos y la mayoría de las mediciones, un CD (que *es* sonido digitalizado) suena justo igual, o tal vez más claro. Las cámaras digitales con resoluciones bastante altas producen imágenes con calidad fotográfica.

¿Por qué querríamos digitalizar los medios? Porque entonces serían más fáciles de manipular, de replicar con exactitud, de comprimir, de buscar, de indexar y clasificar (por ejemplo, para agrupar imágenes o sonidos similares), de comparar y de transmitir. Es decir, es difícil manipular imágenes que están en fotografías, pero es muy sencillo cuando las mismas imágenes están digitalizadas. Este libro trata sobre cómo usar el mundo cada vez más digital de los medios y manipularlos, aprendiendo computación en el proceso.

Gracias a la Ley de Moore la computación multimedia es algo factible como un tema introductorio. La computación multimedia se basa en que la computadora realiza muchas y muchas operaciones en muchos, pero muchos bytes. Las computadoras modernas pueden hacer esto con facilidad. Incluso con lenguajes lentos (pero fáciles de comprender), aun con recetas ineficientes (pero fáciles de leer y escribir), podemos aprender sobre la computación mediante la manipulación de medios.

A la hora de manipular medios, necesitamos respetar los derechos digitales del autor. Está permitido modificar imágenes y sonidos para fines educativos bajo la ley de uso justo (que limita o crea excepciones en cuanto al copyright del propietario). Sin embargo, compartir o publicar imágenes o sonidos manipulados podría representar una infracción del copyright del propietario.

## 1.5 CIENCIAS COMPUTACIONALES PARA TODOS

¿Por qué debe aprender ciencias computacionales escribiendo programas que manipulan medios? ¿Por qué alguien que no desea ser un científico computacional debe aprender acerca de las ciencias computacionales? ¿Por qué debería usted estar interesado en aprender sobre la computación mediante la manipulación de medios?

En la actualidad, la mayoría de los profesionales manipulan medios: papeles, videos, grabaciones de cinta, fotografías y dibujos. Esta manipulación se realiza cada vez con más frecuencia con una computadora. Los medios se encuentran por lo general en un formato digitalizado.

Usamos software para manipular estos medios. Utilizamos Adobe Photoshop para manipular nuestras imágenes y Audacity para manipular nuestros sonidos, y tal vez Microsoft PowerPoint para ensamblar nuestros medios en presentaciones. Usamos Microsoft Word para manipular nuestro texto y Google Chrome o Microsoft Internet Explorer para navegar por los medios en Internet.

Entonces, ¿por qué alguien que *no* desea ser un científico computacional debería estudiar ciencias computacionales? ¿Por qué debe aprender a programar? ¿Acaso no basta con aprender a *usar* todo este grandioso software? Las siguientes secciones proporcionan las respuestas a estas preguntas.

### 1.5.1 Es acerca de la comunicación

Los medios digitales se manipulan con software. *Si sólo puede manipular medios con software que **alguien más** creó para usted, está limitando su habilidad de comunicarse.* ¿Qué pasaría si desea decir algo que no puede decirse en software de Adobe, Microsoft, Apple y

el resto? ¿O qué tal si desea decir algo de una manera que esos programas no soportan? Si sabe cómo programar, incluso aunque tarde *más tiempo* en hacerlo por su cuenta, tendrá la libertad de manipular los medios a su manera.

¿Qué hay sobre aprender a usar estas herramientas a primera instancia? En todos nuestros años trabajando con computadoras, hemos visto muchos tipos de software ir y venir como *el* paquete de dibujo, pintura, procesamiento de palabras, edición de video, etcétera. No podemos aprender tan sólo una herramienta individual y esperar poder usarla durante toda nuestra carrera. Si sabemos *cómo* funcionan las herramientas, tenemos un entendimiento base que puede transferirse de una herramienta a otra. Puede pensar sobre su trabajo con los medios en términos de los *algoritmos* y no de las *herramientas*.

Por último, si va a preparar medios para Web, ya sea para marketing, impresión, difusión o para cualquier otro uso, vale la pena que tenga una idea de lo que es posible y puede hacerse con los medios. Es aún más importante como consumidor de medios que sepa cómo pueden manipularse los medios, que el hecho de saber qué es verdad y qué puede ser sólo un truco. Si conoce los fundamentos de la computación multimedia, entiende lo que va más allá de lo que provee cualquier herramienta individual.

### 1.5.2 Es acerca del proceso

En 1961, Alan Perlis dio una conferencia en el MIT, en donde afirmó que las ciencias computacionales, y en específico la programación, deberían ser parte de una educación liberal [17]. Perlis es una figura importante en el campo de las ciencias computacionales. El mayor premio disponible en ciencias computacionales es el premio Turing de la ACM. Perlis fue el primero en recibir ese premio. Es una figura importante en la ingeniería de software e inició varios de los primeros departamentos de ciencias computacionales en Estados Unidos.

El argumento de Perlis puede hacerse en comparación con el cálculo. Por lo general, el cálculo se considera parte de una educación liberal: no *todos* toman cálculo, pero si desea estar bien educado, por lo general debe tomar un semestre de cálculo. El cálculo es el estudio de los *cambios*, lo cual es importante en muchos campos. Como dijimos antes en este capítulo, las ciencias computacionales constituyen el estudio del **proceso**. Éste es importante para casi cualquier campo, desde negocios pasando por ciencias, hasta medicina pasando por leyes. Para todos es importante conocer el proceso de manera formal. El uso de una computadora para automatizar procesos ha cambiado a todas las profesiones.

Hace poco, Jeannette Wing afirmó que todos deberían aprender el **pensamiento computacional** [34]. Ella considera que los tipos de habilidades que se enseñan en la computación son herramientas imprescindibles para todos los estudiantes. Esto es lo que predijo Alan Perlis: que la automatización de la computación cambiaría la forma en que aprendemos sobre nuestro mundo.

La realidad es que *muchas* personas programan en la actualidad. Los científicos y los ingenieros escriben programas para crear modelos y probarlos en simulaciones, o para analizar datos. Los diseñadores de gráficos escriben programas para realizar tareas del programa en Photoshop o GIMP para ahorrar tiempo, o para mover sus diseños a Web. Los contadores programan cuando crean hojas de cálculo complejas. Muchos profesionales necesitan almacenar y manipular procesos y, por ende, aprenden a programar, justo como Alan Perlis predijo.

## PROBLEMAS

1.1 *Todas* las profesiones usan las computadoras en la actualidad. Use un navegador Web y un motor de búsqueda como Google para buscar sitios que relacionen su campo de

estudio con las ciencias computacionales, las computadoras o la computación. Por ejemplo, busque "biología ciencias computacionales" o "computación gerencial".

1.2 Busque una tabla ASCII en Web: una tabla con una lista de todos los caracteres y su correspondiente representación numérica. Anote la secuencia de números cuyos valores ASCII forman su nombre.

1.3 Busque una tabla Unicode en Web. ¿Cuál es la diferencia entre ASCII y Unicode? ¿Por qué necesitaríamos Unicode si ya tenemos ASCII?

1.4 Considere la representación para las imágenes descrita en la sección 1.4, en donde cada punto (pixel) en la imagen se representa mediante tres bytes para los componentes rojo, verde y azul del color en ese punto. ¿Cuántos bytes se requieren para representar una imagen de 640 por 480, que viene siendo un tamaño de imagen común en Web? ¿Cuántos bytes se requieren para representar una imagen de 1024 por 768, un tamaño común de pantalla? (¿Qué es lo que entiende ahora por el término cámara de "tres megapíxeles"?).

1.5 Un bit puede representar 0 o 1. Con dos bits tenemos cuatro combinaciones posibles: 00, 01, 10 y 11. ¿Cuántas combinaciones diferentes podemos lograr con cuatro bits u ocho bits (un byte)? Cada combinación puede usarse para representar un número binario. ¿Cuántos números puede representar con 2 bytes (16 bits)? ¿Cuántos números puede representar con cuatro bytes?

1.6 ¿Cómo puede representar un *número de punto flotante* en términos de bytes? Busque el término "punto flotante" en Web y vea lo que encuentra.

1.7 Busque a Alan Kay y el *Dynabook* en Web. Encuentre un sitio que le parezca *creíble* para su información e incluya el URL en su respuesta, junto con sus motivos para creer que la fuente es creíble. ¿Qué tiene que hacer él con la computación multimedia?

1.8 Busque ¿cómo contribuyó Grace Hopper a los lenguajes de programación?

1.9 Busque ¿qué departamento de ciencias computacionales dirigió Andrea Lawrence?

1.10 Busque ¿qué tiene que ver Alan Turing con nuestra noción de lo que una computadora puede hacer y cómo funcionan las codificaciones?

1.11 Busque ¿qué computadoras de Harvard contribuyeron a la astronomía?

1.12 Busque ¿cómo contribuyó Adele Goldberg a los lenguajes de programación?

1.13 Busque ¿qué cosas sorprendentes hizo Kurt Gödel con las codificaciones?

1.14 Busque ¿qué cosas sorprendentes hizo Ada Lovelace antes de construir la primera computadora mecánica?

1.15 Busque ¿qué hizo Claude Shannon por su tesis de maestría?

1.16 Busque ¿qué ha hecho Richard Tapia para fomentar la diversidad en la computación?

1.17 Busque ¿qué herramienta de computadora (que usted probablemente usa en forma regular) ayudó a crear Marissa Mayer?

1.18 Busque ¿qué premio de computación recibió Frances Allen?

1.19 Busque ¿en qué nueva tecnología está trabajando Mary Lou Jepsen?

1.20 Busque ¿qué creó Ashley Qualls que vale un millón de dólares?

1.21 Busque ¿qué fue lo que inventó Tim Berners-Lee?

## PARA PROFUNDIZAR

Puede aprender más sobre la ley de copyright en Estados Unidos, incluyendo temas como el uso justo, en `http://www.copyright.gov/`.

El libro *Chaos* de James Gleick describe más sobre las propiedades emergentes: cómo pueden los cambios pequeños conducir a efectos dramáticos, y los impactos no previstos de los diseños debido a las interacciones difíciles de pronosticar.

El libro *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds* [33] de Mitchel Resnick describe la forma en que pueden describirse las hormigas, termitas e incluso los embotellamientos de tráfico y los moldes de cieno de una manera muy precisa, con cientos o miles de procesos muy pequeños (programas) en ejecución, interactuando todos a la vez.

*Exploring The Digital Domain* [3] es un maravilloso libro introductorio sobre la computación, con excelente información sobre los medios digitales.

CAPÍTULO 2

# Introducción a la programación


2.1 LA PROGRAMACIÓN SE REFIERE A LA ASIGNACIÓN DE NOMBRES

2.2 LA PROGRAMACIÓN EN PYTHON

2.3 LA PROGRAMACIÓN EN JES

2.4 LA COMPUTACIÓN MULTIMEDIA Y JES

2.5 CREACIÓN DE UN PROGRAMA


### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Crear y mostrar imágenes.
- Crear y reproducir sonidos.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar JES para introducir y ejecutar programas.
- Crear y usar variables para almacenar valores y objetos, como imágenes y sonidos.
- Crear funciones.
- Reconocer distintos tipos (codificaciones) de datos, como enteros, números de punto flotante y objetos de medios.
- Secuenciar las operaciones en una función.

## 2.1 LA PROGRAMACIÓN SE REFIERE A LA ASIGNACIÓN DE NOMBRES

Idea de ciencias computacionales: **gran parte de la programación trata sobre la asignación de nombres**

Una computadora puede asociar nombres o *símbolos* con casi cualquier cosa: con un byte específico; con una colección de bytes que conforman una variable numérica o un grupo de letras; con un elemento de medios como un archivo, sonido o imagen; o incluso con conceptos más abstractos, como una receta con nombre (un *programa*) o una codificación con nombre (un *tipo*). Al igual que un filósofo o un matemático, un científico computacional considera que una elección de nombres es de alta calidad: el esquema de nombramiento (los nombres y qué es lo que nombran) debería ser elegante, parsimonioso y utilizable. El nombramiento es una forma de abstracción. El nombre se usa para hacer referencia a lo que estamos nombrando.

■

Es obvio que a la computadora en sí no le *importan* los nombres. Éstos son para los humanos. Si la computadora fuera sólo una calculadora, entonces el proceso de recordar palabras y su asociación con valores sería tan sólo un desperdicio de la memoria. Pero para los humanos esto es *muy* poderoso, ya que nos permite trabajar con la computadora de una manera natural, incluso de una forma que extiende la manera en que pensamos sobre las recetas (procesos) en general.

En realidad, un **lenguaje de programación** es un conjunto de nombres para el que una computadora tiene codificaciones, de tal forma que los nombres indican a la computadora que realice acciones esperadas e interprete nuestros datos de ciertas formas esperadas. Algunos de los nombres de los lenguajes de programación nos permiten definir *nuevos* nombres, que a su vez nos permiten crear nuestros propios niveles de codificación. Asignar una variable a un valor es una forma de definir un nombre para la computadora. Definir una función es dar un nombre a una receta.

Un **programa** está formado por un conjunto de nombres y sus valores, en donde algunos de estos nombres tienen valores de instrucciones para la computadora ("código"). Nuestras instrucciones estarán en el lenguaje de programación Python. Combinar estas dos definiciones significa que el lenguaje de programación Python nos proporciona un conjunto de nombres útiles que tienen un significado para la computadora, y que nuestros programas son entonces una selección de nombres útiles de Python, además de los nombres que definamos nosotros, que en conjunto nos permiten indicar a la computadora lo que queremos hacer.

**Idea de ciencias computacionales: los programas son para las personas, no para las computadoras**

Recuerde que los nombres sólo son significativos para las personas, no para las computadoras. Éstas sólo reciben instrucciones. Un buen programa es significativo (comprensible y útil) para los humanos. Las computadoras están hechas para servir a las personas.

■

Hay nombres buenos y malos. Esto no tiene nada que ver con las malas palabras o los TLA (acrónimos de tres letras). Un buen conjunto de codificaciones y nombres nos permite describir las recetas de tal forma que sea algo natural, sin tener que decir demasiado. Podemos considerar a la diversidad de lenguajes de programación como una colección de conjuntos de nombramientos y codificaciones. Algunos son mejores para ciertas tareas que otros. Algunos lenguajes requieren que escribamos más para describir la misma receta que otros, pero algunas veces eso que está "de más" nos conduce a una receta mucho más (humanamente) legible que ayuda a otros a comprender lo que estamos diciendo.

Los filósofos y matemáticos buscan sensaciones muy similares de calidad. Tratan de describir al mundo en unas cuantas palabras, buscando una selección elegante de éstas que cubran muchas situaciones, pero a la vez puedan ser comprendidas por sus compañeros filósofos y matemáticos. Esto es exactamente lo que hacen los científicos computacionales.

Con frecuencia, también se asigna un nombre a la forma en que pueden interpretarse las unidades y valores (**datos**) de una receta. ¿Recuerda que en la sección 1.3 vimos que todo está en bytes, pero que los bytes pueden interpretarse como números? En algunos lenguajes de programación, podemos decir de manera explícita que cierto valor es un *byte*, y más tarde decirle al lenguaje que lo trate como número, un *entero* (o algunas veces *int*). De manera similar, podemos decir a la computadora que esta serie específica de bytes es una colección de números (un **arreglo de enteros**), una colección de caracteres (una **cadena**) o incluso una

codificación más compleja de un solo número de punto flotante (un **float**: cualquier número que contenga un punto decimal).

En Python indicaremos de manera explícita a la computadora cómo interpretar nuestros valores, pero muy rara vez le indicaremos que ciertos nombres sólo se asocian con ciertas codificaciones. Los lenguajes como Java y C++ están *fuertemente tipificados*: esto quiere decir que, en estos lenguajes, los nombres están fuertemente asociados con ciertos tipos o codificaciones. Requieren que usted diga que este nombre sólo se asociará con enteros y que ese otro será solamente un número de punto flotante. De todas formas Python cuenta con *tipos* (codificaciones a las que podemos hacer referencia por nombre), pero no son tan explícitos. Python también tiene **palabras reservadas**. Estas son palabras que no podemos usar para nombrar cosas, ya que tienen un significado predeterminado en el lenguaje.

**Idea de ciencias computacionales: los nombres son símbolos e identificadores**

Utilizamos la palabra "nombres", pero los lenguajes de programación usan términos más específicos. Algunos lenguajes de programación llaman *símbolo* al nombre de un valor, función u objeto. Python (y Java) lo llaman *identificador*. Podemos pensar en esto como la noción humana de un "nombre", pero los mensajes de error usarán los términos más específicos. Tal vez el mensaje de error más común en Java sea *Identifier expected* (*Se esperaba identificador*), lo cual por lo general significa que falta un nombre o está mal escrito.

■

### 2.1.1 Los archivos y sus nombres

Un lenguaje de programación no es el único lugar en donde las computadoras asocian nombres y valores. El **sistema operativo** de su computadora se encarga de los archivos en su disco y asocia nombres con ellos. Los sistemas operativos que usted usa o tal vez conozca son: Windows XP, Vista, Windows 7, MacOS y Linux. Un **archivo** es una colección de valores (bytes) en su **disco duro** (la parte de su computadora que almacena cosas después de apagar la computadora). Si conoce el nombre de un archivo y se lo indica al sistema operativo, recibirá los valores asociados con el nombre.

Tal vez piense: "He estado usando la computadora durante años y *nunca* le he proporcionado un nombre de archivo al sistema operativo". Tal vez no se dio cuenta que lo estaba haciendo, pero al elegir un archivo de un cuadro de diálogo para seleccionar archivos en Photoshop, o al hacer doble clic en un archivo en una ventana de *directorios* (o en el Explorador o Buscador), está pidiendo a algún software en alguna parte que proporcione al sistema operativo el nombre que usted está eligiendo o en el que está haciendo doble clic, para recibir los valores de vuelta. Pero cuando escribe sus propias recetas, está obteniendo de manera explícita los nombres de archivos y preguntando por sus valores.

En la actualidad nuestros discos duros son enormes. Es probable que tenga más de un archivo con el mismo nombre de archivo en ellos, en alguna parte. Quizás tenga un par de archivos llamados **reporte.doc** (tal vez uno para Historia y otro para Química, y posiblemente otro más para su pasantía) o `amigos.jpg` en su disco en este momento. Mientras éstos se encuentren en diferentes directorios, el mismo nombre puede hacer referencia a distintos archivos sin problemas. La computadora tiene un *sistema de archivos* que administra los directorios y los archivos. El nombre completo de un archivo se conoce como *ruta* y describe los directorios que debemos seguir para obtener un archivo específico. Usaremos esta idea en breve.

Los archivos son *muy* importantes para la computación multimedia. Los discos pueden almacenar acres y acres de información en ellos. ¿Recuerda nuestra discusión sobre la Ley

de Moore? ¡La capacidad de un disco por dólar está aumentando con *más rapidez* que la velocidad de la computadora por dólar! Hoy en día los discos de computadora pueden almacenar películas enteras, horas (¿o días?) de sonidos, y el equivalente de cientos de rollos de película de imágenes.

Estos medios no son pequeños. Incluso en un formato *comprimido*, las imágenes del tamaño de la pantalla pueden ser de más de un millón de bytes, y las canciones pueden ser de 3 millones de bytes o más. Necesita mantenerlos en alguna parte en donde puedan persistir después de que la computadora se apague y en donde haya mucho espacio.

Por el contrario, la **memoria** de su computadora no es permanente (desaparece al apagar la máquina) y es relativamente pequeña. La memoria de computadora está aumentando su tamaño todo el tiempo, pero aún es sólo una fracción de la cantidad de espacio en su disco. Cuando trabaja con medios, los carga desde el disco hacia la memoria, pero no sería conveniente que se quedaran en la memoria al terminar de trabajar con ellos. Son demasiado grandes.

Compare la memoria de la computadora con una habitación. En la habitación puede alcanzar las cosas con facilidad: están a la mano, se pueden alcanzar y usar. Pero no sería conveniente que pusiera todo lo que posee (o todo lo que espera poseer) en esa habitación. Todas sus pertenencias, es decir, sus esquís, su auto y su velero. Eso sería tonto. En vez de ello, debe almacenar las cosas grandes en lugares designados para ello. Usted sabe cómo obtenerlas cuando las necesita (y tal vez regresarlas a su habitación si lo cree necesario o si puede).

Cuando llevamos cosas a la memoria, se asigna un nombre al valor de modo que podamos recuperarlo y usarlo más tarde. En ese sentido, la programación es algo como el *álgebra*. Para escribir ecuaciones y funciones que sean generalizables (es decir, que funcionen para cualquier número o valor), se escriben con *variables* como $PV = nRT$ o $e = Mc^2$, o $f(x) = sen(x)$. Esas letras P, V, R, T, e, M, c y x son nombres para los valores. Al evaluar $f(30)$ sabemos que la $x$ es el nombre para el número 30 al calcular $f$. Nombraremos los medios (como valores) de la misma forma cuando los usemos en la programación.

## 2.2 LA PROGRAMACIÓN EN PYTHON

El lenguaje de programación que usaremos en este libro se llama **Python**. Es un lenguaje inventado por Guido van Rossum. Él nombró este lenguaje en honor al famoso grupo humorístico inglés *Monty Python*. Este lenguaje se ha utilizado durante años por personas sin una capacitación formal en ciencias computacionales; está orientado a ser fácil de usar. La forma específica de Python que usaremos aquí es **Jython**, ya que se puede utilizar en multimedia en múltiples plataformas.

En realidad programaremos mediante una herramienta conocida como **JES** (**Jython Environment for Students**, o **entorno Jython para estudiantes**). JES es un simple **editor** (herramienta para introducir el texto de un programa) y una herramienta de interacción, de modo que pueda probar cosas en JES y crear nuevas recetas dentro de él. Los nombres de los medios (funciones, variables, codificaciones) de los que hablaremos en este libro se desarrollaron para trabajar dentro de JES (es decir, no forman parte de una distribución normal de Jython, aunque el lenguaje básico que utilizaremos es Python normal).

Podrá leer las instrucciones sobre cómo instalar JES en http://mediacomputation.org. El proceso que se explica ahí lo guiará para instalar Java, Jython y JES, y le dará un icono agradable para que haga clic en él e inicie JES. Este entorno está disponible para Windows, Macintosh y Linux.

**Tip de depuración: consiga Java, si tiene que hacerlo**

Para la mayoría de las personas, arrastrar la carpeta de JES en su disco duro es todo lo que necesitan para empezar. No obstante, si tiene Java instalado y es una versión más antigua que no pueda ejecutar JES, tal vez tenga problemas para hacer que JES inicie. Si tiene problemas, consiga una nueva versión de Java del sitio de Sun en http://www.java.sun.com.

■

## 2.3 LA PROGRAMACIÓN EN JES

La forma de iniciar JES dependerá de su plataforma. En Windows y Macintosh hay un icono de JES en el que sólo tiene que hacer doble clic. En Linux, tal vez tenga que cambiar a su directorio Jython y escribir un comando como `./JES.sh`. Consulte las instrucciones en el sitio Web para ver lo que puede funcionar en su computadora.

**Error común: JES puede ser lento para arrancar**

JES puede tardar un poco en cargarse. No se preocupe: tal vez vea la pantalla inicial durante mucho tiempo, pero si puede ver esa pantalla, se cargará. A menudo iniciará con más rapidez después del primer uso.

■

**Error común: cómo hacer que JES se ejecute más rápido**

Como veremos más adelante, cuando ejecuta JES en realidad está ejecutando Java. Como Java necesita memoria, es probable que JES se ejecute con lentitud, por lo tanto, deberá asignarle más memoria. Para hacer esto, salga de las demás aplicaciones que esté ejecutando. Su programa de correo electrónico, su mensajero instantáneo y su reproductor de música digital ocupan memoria; ¡algunas veces mucha! Salga de esas aplicaciones y JES se ejecutará con más rapidez.

■

Una vez que inicie JES, verá en su pantalla algo como la figura 2.1. En JES hay dos áreas principales (la barra entre ellas se mueve, para que pueda cambiar el tamaño de las dos).

- La parte superior es el **área del programa**. Aquí es donde usted escribe *sus* recetas: los programas que va a crear y sus nombres. Esta área es tan sólo un editor de texto; piense en ella como Microsoft Word para sus programas. En realidad la computadora no trata de interpretar los nombres que usted escribe en el área del programa sino hasta que presiona el botón LOAD PROGRAM (CARGAR PROGRAMA), y no puede presionar LOAD PROGRAM hasta que haya guardado su programa (usando el elemento de menú SAVE [GUARDAR] que está debajo del menú FILE [ARCHIVO]).

  No se preocupe si presiona LOAD PROGRAM antes de acordarse de guardar el programa. JES no cargará el programa sino hasta que esté guardado, por lo que tendrá la oportunidad de guardarlo.

- La parte inferior es el **área de comandos**. Aquí es donde literalmente *ordenamos* a la computadora que haga algo. Usted escribe sus comandos en el indicador >>> y al presionar la tecla de RETORNO (Apple) o INTRO (Windows), la computadora interpre-

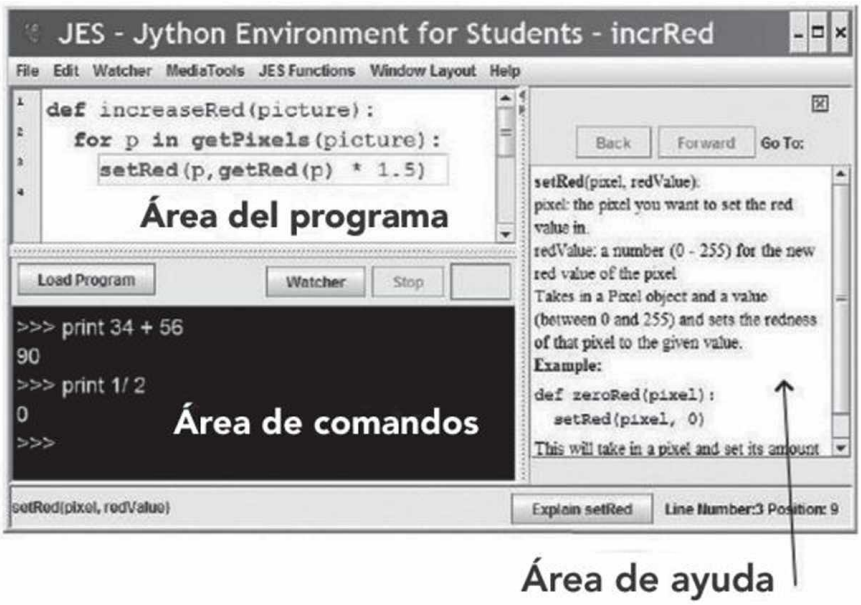



**FIGURA 2.1**
Las áreas de JES etiquetadas.

tará sus palabras (es decir, aplicará los significados y codificaciones del lenguaje de programación Python) y hará lo que usted le indicó. Esta interpretación incluirá todo lo que escribió y lo que cargó del área del programa también.

- El área de la derecha es el **área de ayuda**. Puede seleccionar algo y después hacer clic en el botón Explain (Explicar) para obtener ayuda sobre ese elemento.

En la figura 2.1 podemos ver otras características de JES, pero todavía no las usaremos mucho. El botón Watcher (Vigilante) abre el **vigilante** (**un depurador**), una ventana con herramientas para vigilar la forma en que la computadora ejecuta nuestro programa. El botón STOP (DETENER) nos permite detener un programa en ejecución (por ejemplo, si considera que se ha ejecutado por mucho tiempo, o si descubre que no está haciendo lo que usted quería).

**Tip de funcionamiento: ¡familiarícese con su ayuda!**

Una característica *importante* que debemos explorar es el menú HELP (AYUDA). Bajo este menú encontrará una gran cantidad de ayuda excelente relacionada con la programación y el uso de JES. Empiece a explorarla ahora para que tenga una idea de lo que tendrá disponible a la hora que empiece a escribir sus propios programas.

■

## 2.4 LA COMPUTACIÓN MULTIMEDIA Y JES

Comenzaremos por escribir comandos en el área de comandos. No definiremos nuevos nombres todavía; simplemente usaremos los nombres que la computadora ya conoce dentro de JES.

Uno de los nombres importantes es `print`; siempre se utiliza con algo que va después. El significado de `print` es: "Muestra una representación legible de lo que sigue a continuación". Después de `print` puede ir un nombre que la computadora conozca, o una *expresión* (literalmente, en el sentido algebraico). Pruebe escribir `print 34 + 56`, haga clic en el área de comandos, escriba el comando y presione INTRO, como se muestra a continuación:

```
>>> print 34 + 56
90
```

`34 + 56` es una expresión numérica que Python entiende. Como podemos ver, está compuesta de dos números y una operación (a nuestro entender, un *nombre*) que Python sabe cómo llevar a cabo, + significa "sumar". Python entiende otros tipos de expresiones; no todas ellas son numéricas.

```
>>> print 31.1/46.5
0.7333333333333334
>>> print 22 * 33
726
>>> print 14 - 15
-1
>>> print "Hola"
Hola
>>> print "Hola" + "Mark"
HolaMark
```

Python entiende bastantes operaciones matemáticas estándar. También sabe cómo reconocer distintas clases o **tipos** de números: enteros y de punto flotante. Un número de punto flotante contiene un punto decimal. Un número entero no tiene que contener un punto decimal. Python también sabe cómo reconocer *cadenas* (secuencias de caracteres) que empiezan y terminan con el signo " (comillas). Incluso sabe lo que significa "sumar" dos cadenas: simplemente coloca una justo después de la otra.

**Error común: los tipos de Python pueden producir resultados extraños**

Python toma muy en serio los tipos. Si ve que usted usa enteros, piensa que usted desea recibir un resultado entero de sus expresiones. Si ve que usa números de punto flotante, piensa que usted desea un resultado de punto flotante. Suena razonable, ¿o no? Pero qué hay sobre lo siguiente:

```
>>> print 1.0/2.0
0.5
>>> print 1/2
0
```

¿Acaso 1/2 es igual a 0? ¡Pues claro! 1 y 2 son enteros. Como no hay un entero igual a 1/2, ¡entonces la respuesta debe ser 0! La acción de sumar ".0" a un entero convence a Python de que estamos hablando sobre números de punto flotante, por lo que muestra el resultado en formato de punto flotante. Muchos otros lenguajes de programación funcionan de la misma manera. Python 3.0 *no* interpreta los números así, por lo que este resultado podría ser distinto a medida que Python 3.0 se haga más popular.

■

Python también conoce las **funciones**. ¿Recuerda las funciones de álgebra? Son una "caja" en la que colocamos un valor y de ahí sale otro. Una de las funciones que Python

conoce recibe un carácter como el valor de *entrada* (el valor que entra en la caja) y devuelve o envía (el valor que sale de la caja) el número que viene siendo la asignación ASCII para ese carácter. El nombre de esta función es `ord` (de *ordinal*) y podemos usar `print` para mostrar el valor que devuelve la función `ord`:

```
>>> print ord("A")
65
```

Hay otra función integrada en Python llamada `abs`: es una función que devuelve el valor absoluto del valor introducido.

```
>>> print abs(1)
1
>>> print abs(-1)
1
```

**Tip de depuración: errores comunes de escritura**

Si escribe algo que Python no pueda comprender, obtendrá un error de sintaxis.

```
>>> pint "Hola"
Your code contains at least one syntax
error, meaning it is not legal Jython.
```

Si trata de acceder a una palabra que Python no conoce, Python dirá que no conoce ese nombre.

```
>>> print a
A local or global name could not be found. You need to define
the function or variable before you try to use it in any way.
```

Un *nombre local* es un nombre definido dentro de una función, y un *nombre global* es un nombre disponible para todas las funciones (como `pickAFile`).

■

Otra función que JES conoce es la que nos permite elegir un archivo del disco. Tal vez se haya dado cuenta que cambiamos de decir "Python conoce" a "JES conoce". `print` es algo que todas las implementaciones de Python conocen. `pickAFile` es algo que creamos para JES. En general, puede ignorar la diferencia, pero si trata de usar otro tipo de Python, será importante saber lo que es común y lo que no es. No recibe ninguna entrada, a diferencia de `ord`, pero devuelve una cadena que representa el nombre del archivo en su disco. El nombre de la función es `pickAFile`. Python es muy "especial" en cuanto al uso de mayúsculas: ¡si usa `pickafile` o `Pickafile` no funcionarán! Pruebe de esta forma: `print pickAFile()`. Cuando lo haga, obtendrá algo similar a la figura 2.2.

Es probable que esté familiarizado con la forma de usar un selector de archivos o cuadro de diálogo de archivos:

- Haga doble clic en las carpetas/directorios para abrirlos.
- Haga clic para seleccionar y luego haga clic en ABRIR, o haga doble clic para seleccionar un archivo.

Una vez que seleccione un archivo, lo que se devuelve es el *nombre de archivo completo* como una cadena (secuencia de caracteres). (Si hace clic en CANCELAR, `pickAFile` devuelve la *cadena vacía*: una cadena de caracteres que no contiene ningún carácter, o `""`). Pruébelo: escriba `print pickAFile()` y después ABRA un archivo.

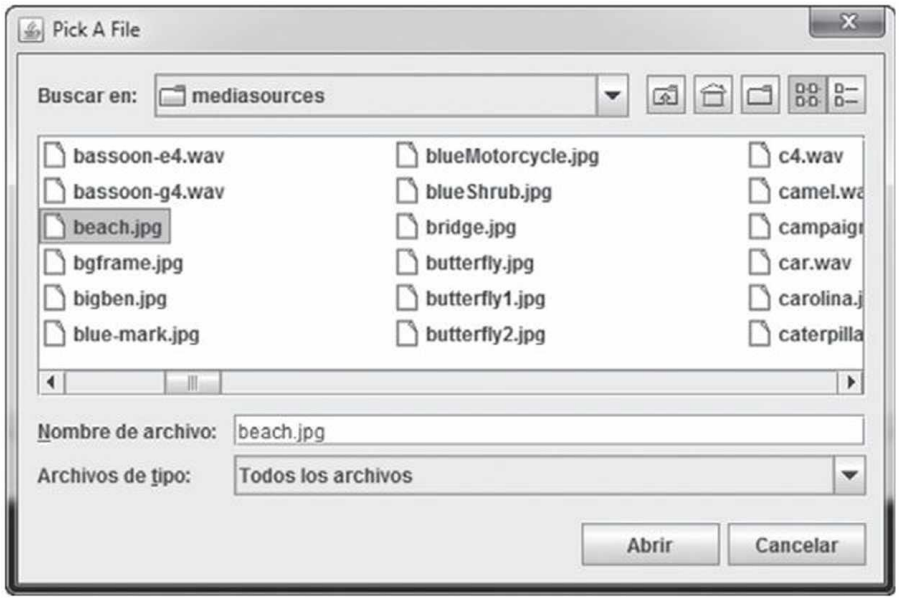



**FIGURA 2.2**
El selector de archivos.

```
>>> print pickAFile()
C:\ip-book\mediasources\beach.jpg
```

Lo que obtenga al momento de seleccionar un archivo dependerá de su sistema operativo. En Windows, el nombre de archivo casi siempre empieza con **C:** y contiene barras diagonales inversas (\). En Linux o MacOS es probable que se vea así: **/Users/guzdial/ip-book/mediasources/beach.jpg**. En realidad este nombre de archivo consta de dos partes:

- El carácter entre palabras (el / entre "Users" y "guzdial") se conoce como **delimitador de ruta** o **separador de ruta**. Todos desde el principio del nombre del archivo hasta el último delimitador de ruta es la *ruta* hacia el archivo. Eso describe con exactitud *en dónde* existe un archivo en el disco duro (en qué **directorio**).
- La última parte del archivo (en este caso, **beach.jpg**) se conoce como **nombre de archivo base**. Al observar el archivo en la ventana del Finder/Explorador/directorio (dependiendo de su sistema operativo), ésa es la parte que puede ver. Los últimos tres caracteres (después del punto) se conocen como **extensión del archivo**. Éstos identifican la codificación del archivo.

Los archivos con la extensión ".jpg" son archivos **JPEG**. Contienen imágenes (dicho en forma más literal, contienen datos que pueden *interpretarse* como la *representación* de una imagen, pero basta con decir que "contienen imágenes"). JPEG es una *codificación* (representación) estándar para cualquier tipo de imagen. Los otros archivos de medios que usaremos con frecuencia son los archivos ".wav" (figura 2.3). La extensión ".wav" significa que son archivos **WAV**; es decir, contienen sonidos. WAV es una codificación estándar para los sonidos. Existen muchos otros tipos de extensiones para los archivos, e incluso hay muchos tipos más de extensiones de medios. Por ejemplo, también hay archivos GIF (".gif")

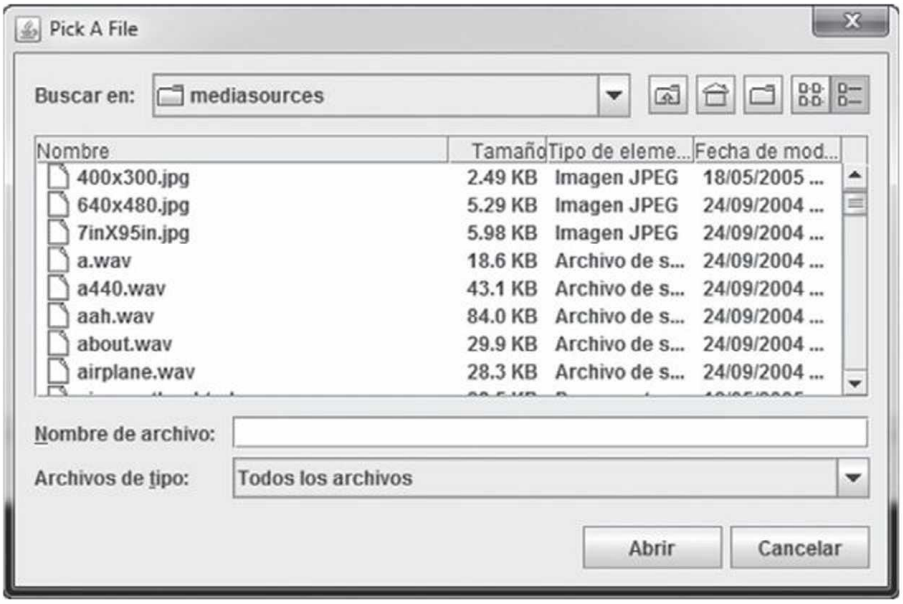



**FIGURA 2.3**
Selector de archivos con los tipos de medios identificados.

para imágenes y archivos AIFF (".aif" o ".aiff") para sonidos. Nos apegaremos a JPEG y WAV en este libro, tan sólo para evitar demasiada complejidad.

### 2.4.1 Mostrar una imagen

Ahora sabemos cómo obtener un nombre de archivo completo: ruta y nombre base. Esto *no* significa que el archivo en sí se carga en memoria. Para llevar el archivo a la memoria, tenemos que decir a JES cómo interpretarlo. *Sabemos* que los archivos JPEG son imágenes, pero tenemos que decir a JES de manera explícita que lea el archivo y cree una imagen a partir de él. Hay una función para eso también, llamada `makePicture`.

`makePicture` *requiere* un **parámetro**: cierta entrada para la función. Al igual que `ord`, la entrada se especifica dentro de paréntesis. Recibe un nombre de archivo. Por suerte, ahora sabemos cómo obtener uno.

```
>>> print makePicture(pickAFile())
Picture, filename C:\ip-book\mediasources\barbara.jpg
   height 294  width 222
```

El resultado de `print` sugiere que en realidad creamos una imagen, a partir de un nombre de archivo, una altura y una anchura dadas. ¡Bien hecho! Ah, pero ¿en verdad quería *ver* la imagen? ¡Necesitaremos otra función! (¿Mencionamos en alguna parte que las computadoras son estúpidas?) La función para mostrar la imagen se llama `show`. Esta función *también* recibe un parámetro: `Picture`.

Pero tenemos un problema. No nombramos la imagen que acabamos de crear, por lo que no tenemos ninguna forma de hacer referencia a ella de nuevo. No podemos simplemente decir a la computadora que muestre la imagen que creamos hace un segundo, pero que no le asigna-

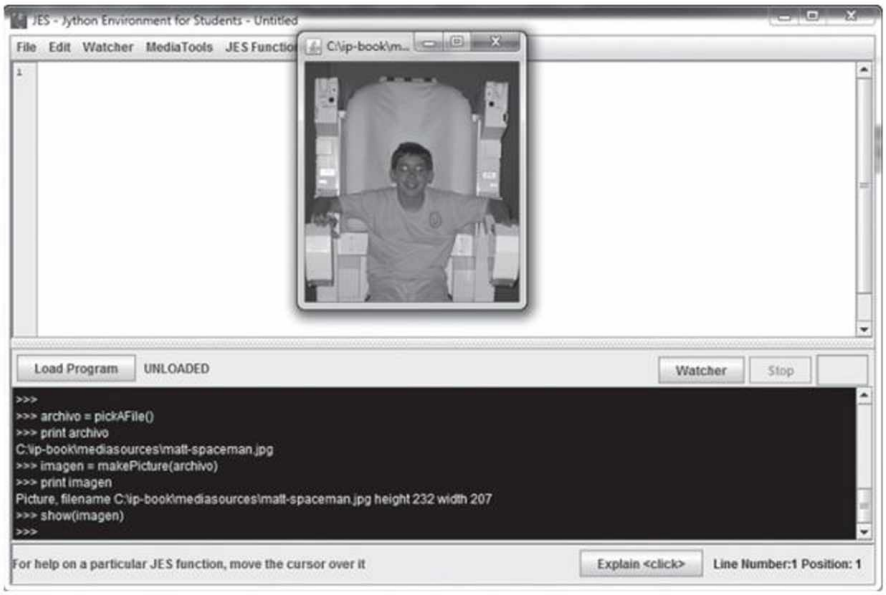



**FIGURA 2.4**
Seleccionar, crear y mostrar una imagen, además de nombrar las piezas.

mos nombre. Las computadoras no recuerdan cosas a menos que las nombremos. Al proceso de crear un nombre para un valor también se le conoce como *declaración de una variable.*

Empecemos de nuevo y esta vez primero asignaremos un nombre al archivo que seleccionemos. También nombraremos la imagen que vamos a crear. De esta forma podremos mostrar la imagen con nombre, como se muestra en la figura 2.4.

Puede elegir y nombrar un archivo usando `archivo = pickAFile()`. Esto significa crear el nombre del archivo y configurarlo de manera que haga referencia al valor devuelto por la función `pickAFile()`. Ya vimos que dicha función devuelve el nombre del archivo (incluyendo la ruta). Podemos nombrar y crear una imagen usando `imagen = makePicture(archivo)`. Esta instrucción pasa el nombre de archivo a la función `makePicture` y luego devuelve la imagen creada. El nombre `imagen` se refiere a la imagen creada. Después podemos mostrar la imagen creada al pasar el nombre `imagen` a la función `show`.

Otra manera es hacer todo esto de una sola vez, ya que la salida de una función puede usarse como la entrada de otra: `show(makePicture(pickAFile()))`. Esto es lo que vemos en la figura 2.5. Para empezar, usted podrá elegir un archivo y después se pasará el nombre de ese archivo a la función `makePicture`, que a su vez pasará la imagen resultante a la función `show`. Pero, de nuevo, como no asignamos un nombre a la imagen, no podremos hacer referencia a ella otra vez.

¿Ya lo probó usted mismo? Felicidades: acaba de realizar su primer cómputo multimedia.

Para resumir, diremos que el código

```
>>> archivo = pickAFile()
>>> imagen = makePicture(archivo)
>>> show(imagen)
```

hace lo mismo que este código (suponiendo que seleccione el mismo archivo):

```
>>> imagen = makePicture(pickAFile())
>>> show(imagen)
```

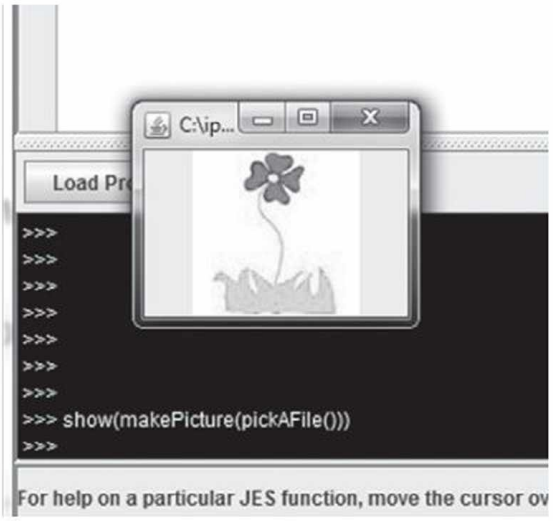



**FIGURA 2.5**
Seleccionar, crear y mostrar una imagen, usando cada función como entrada para la siguiente.

Esto incluso puede hacerse en una sola línea (sin usar nombres, lo que significa que no podremos mostrar o cambiar de nuevo la imagen después de esta línea).

```
>> show(makePicture(pickAFile()))
```

Si escribe `print show(imagen)`, observará que la salida de `show es None`. En Python las funciones no *tienen* que devolver un valor, a diferencia de las funciones matemáticas reales. Si una función *hace* algo (como abrir una imagen en una ventana), puede ser útil sin que además necesite devolver un valor. Los científicos computacionales usan el término **efecto secundario** para cuando una función realiza algo distinto al producido a través de su cómputo de la entrada al valor de retorno. Mostrar ventanas y hacer sonidos es un tipo de cómputo de efecto secundario.

## 2.4.2 Reproducir un sonido

Podemos replicar todo este proceso anterior con sonidos.

- Seguimos usando `pickAFile` para buscar el archivo que deseamos y obtener su nombre de archivo. Esta vez elegiremos un archivo que termine con .wav.
- Ahora usaremos `makeSound` para crear un sonido. Como podría imaginar, `makeSound` recibe un nombre de archivo como entrada.
- Usaremos `play` para reproducir el sonido. `play` recibe un sonido como entrada pero devuelve `None`.

He aquí los mismos pasos que vimos antes con las imágenes:

```
>>> archivo = pickAFile()
>>> print archivo
C:\ip-book\mediasources\helloWorld.wav
>>> sonido = makeSound(archivo)
>>> print sonido
```

```
Sound file: C:\ip-book\mediasources\helloWorld.wav number of samples: 44033
>>> print play(sonido)
None
```

(En el siguiente capítulo explicaremos lo que significa la longitud del sonido). Pruebe esto por su cuenta, usando archivos JPEG y archivos WAV que estén en su propia computadora, que cree usted mismo o que estén disponibles en http://www.mediacomputation.org. (Hablaremos más sobre dónde obtener los medios y cómo crearlos en los siguientes capítulos).

### 2.4.3 Nombrar valores

Como vimos en la última sección, para nombrar datos usamos el signo =. Podemos revisar nuestros nombramientos usando `print`, como lo hemos hecho hasta ahora.

```
>>> miVariable=12
>>> print miVariable
12
>>> otraVariable=34.5
>>> print otraVariable
34.5
>>> miNombre="Mark"
>>> print miNombre
Mark
```

No interprete el signo = como "es igual a". Esto es lo que significa en matemáticas, pero no se asemeja en nada a lo que estamos haciendo aquí. Interprete = como "se convierte en un nombre para". Así, `miVariable=12` significa "`miVariable` se convierte en un nombre para `12`". Por ende, lo inverso (colocar la *expresión* del lado izquierdo y el nombre a la derecha) no tiene sentido: `12=miVariable` significaría entonces "`12` se convierte en un nombre para `miVariable`".

```
>>> x = 2 * 8
>>> print x
16
>>> 2 * 8 = x
Your code contains at least one syntax error, meaning
it is not legal Jython.
```

Podemos usar nombres más de una vez.

```
>>> print miVariable
12
>>> miVariable="Hola"
>>> print miVariable
Hola
```

La *vinculación* (o asociación) entre el nombre y los datos sólo existe hasta que (a) el nombre se asigna a otra cosa, o (b) salga de JES. La relación entre nombres y datos (o incluso entre nombres y funciones) sólo existe durante una sesión de JES.

Recuerde que los datos tienen codificaciones o tipos. La forma en que los datos actúan en las expresiones depende en parte de sus tipos. Observe cómo el *entero* 12 y la *cadena*

"12" actúan de manera distinta para la multiplicación que se muestra a continuación. Ambos hacen algo razonable para su tipo, pero son acciones muy distintas.

```
>>> miVariable=12
>>> print miVariable*4
48
>>> miOtraVariable="12"
>>> print miOtraVariable*4
12121212
```

Podemos asignar nombres a los *resultados* de funciones. Si nombramos el resultado de `pickAFile`, obtendremos el mismo resultado cada vez que imprimamos el nombre. No volvemos a ejecutar `pickAFile`. Nombrar código para volver a ejecutarlo es lo que hacemos cuando definimos funciones, lo cual veremos un poco más adelante.

```
>>> archivo = pickAFile()
>>> print archivo
C:\ip-book\mediasources\640x480.jpg
>>> print archivo
C:\ip-book\mediasources\640x480.jpg
```

En el siguiente ejemplo, asignamos nombres al nombre de archivo y la imagen.

```
>>> miNombrearchivo = pickAFile()
>>> print miNombrearchivo
C:\ip-book\mediasources\barbara.jpg
>>> miImagen = makePicture(miNombrearchivo)
>>> print miImagen
Picture, filename C:/ip-book/mediasources/barbara.jpg
height 294 width 222
```

Observe que las nociones algebraicas de *sustitución* y *evaluación* funcionan aquí también. `miImagen = makePicture(miNombrearchivo)` ocasiona que se cree la misma imagen exacta como si hubiéramos ejecutado `makePicture(pickAFile())`,[1] ya que configuramos `miNombrearchivo` para que fuera igual al resultado de `pickAFile()`. Los valores se sustituyen por los nombres cuando se evalúa la expresión. `makePicture(miNombrearchivo)` es una expresión que se expande, en tiempo de evaluación, a `makePicture("C:\ip-book\mediasources\barbara.jpg")`, debido a que **C:\ip-book\mediasources\barbara.jpg** es el nombre del archivo que se seleccionó cuando se evaluó `pickAFile()` y el valor devuelto se nombró `miNombrearchivo`.

También podemos reemplazar las *invocaciones* (o *llamadas*) a la función con el *valor* devuelto. `pickAFile()` devuelve una *cadena*: un grupo de caracteres encerrados entre comillas. Podemos hacer que el último ejemplo trabaje así, también.

```
>>> miNombrearchivo = "C:/ip-book/mediasources/barbara.jpg"
>>> print miNombrearchivo
C:/ip-book/mediasources/barbara.jpg
>>> miImagen = makePicture(miNombrearchivo)
```

[1]Suponiendo, desde luego, que haya elegido el mismo archivo.

```
>>> print miImagen
Picture, filename C:/ip-book/mediasources/barbara.jpg
height 294 width 222
```

O incluso podemos sustituir el nombre.

```
>>> miImagen = makePicture("C:/ip-book/mediasources/barbara.jpg")
>>> print miImagen
Picture, filename C:/ip-book/mediasources/barbara.jpg
height 294 width 222
```

**Error común: los nombres de archivos de Windows y las barras diagonales inversas**

Windows usa barras diagonales inversas como delimitadores de archivo. Python proporciona significados especiales a ciertas combinaciones de barras diagonales inversas y caracteres, como veremos más adelante. Por ejemplo, '\n' significa lo mismo que la tecla INTRO o RETORNO. Estas combinaciones pueden ocurrir de manera natural en los nombres de archivos de Windows. Para evitar que Python malinterprete estos caracteres, puede usar barras diagonales normales, como en **C:/ip-book/mediasources/barbara.jpg**, o puede escribir sus nombres de archivo con una "r" al frente, como en el siguiente ejemplo:

```
>>> miArchivo=r"C:\ip-book\mediasources\barbara.jpg"
>>> print miArchivo
C:\ip-book\mediasources\barbara.jpg
>>> miArchivo
'C:\\ip-book\\mediasources\\barbara.jpg'
```

■

**Idea de ciencias computacionales: podemos sustituir nombres, valores y funciones**

Podemos sustituir un valor, un nombre asignado a ese valor y la función que devuelve el mismo valor *indistintamente*. A la computadora lo que le importa es el valor, no si proviene de una cadena, de un nombre o de la llamada a una función. La clave es que la computadora va a *evaluar* el valor, el nombre y la función. Mientras que estas *expresiones* se evalúen de la misma forma, pueden usarse de forma indistinta.

■

En realidad no necesitamos usar `print` cada vez que pedimos a la computadora que haga algo. Si queremos llamar a una función que no devuelve nada (y por ende, no sirve de nada usar `print`), tan sólo podemos llamar a la función escribiendo su nombre, su entrada (si la tiene) y presionando Intro.

```
>>> show(miImagen)
```

A estas instrucciones que indican a la computadora que haga cosas les decimos *comandos*. `print miImagen` es un comando, al igual que `miNombrearchivo = pickAFile()` y `show(miImagen)`. Éstas son más que expresiones: indican a la computadora que *haga* algo.

## 2.5 CREACIÓN DE UN PROGRAMA

En la sección anterior usamos nombres para representar valores. Los valores se sustituyen por los nombres al evaluar la expresión. Podemos hacer lo mismo para los programas. Podemos nombrar una serie de comandos y después sólo usar el nombre cada vez que queramos ejecutar los comandos. En Python, el nombre que definamos será una *función*. Entonces, un *programa* en Python es una colección de una o más funciones que realizan una tarea útil. Usaremos el término *receta* para describir programas (o porciones de programas) que realizan una operación de medios útil, incluso aunque lo que está cubierto por el término no sea suficiente como para formar un programa útil por sí solo.

¿Recuerda cuando dijimos que se puede asignar un nombre a casi cualquier cosa en las computadoras? Ya vimos cómo nombrar valores. Ahora veremos cómo nombrar recetas.

**Tip de funcionamiento: ¡pruebe *todas* las recetas!**

Para poder comprender realmente lo que está sucediendo, escriba, cargue y ejecute *todas* las recetas de este libro. *CADA* una de ellas. Todas son cortas y la práctica le ayudará en forma considerable para convencerlo de que los programas funcionan, a desarrollar sus habilidades de programación y a comprender *por qué* funcionan.

■

El nombre que Python entiende para *definir* el nombre de nuevas recetas es `def`. Este nombre no es una función: es un comando como `print`. `def` se usa para *definir* nuevas funciones. Ahora bien, hay ciertas cosas que deben ir después de la palabra `def`. La estructura de lo que va en la línea con el comando `def` se conoce como la **sintaxis** del comando: las palabras y caracteres que tienen que estar ahí para que Python comprenda lo que está sucediendo, además del orden de esas cosas.

Después de `def` deben ir tres cosas en la misma línea:

- El nombre de la receta que está definiendo, como `mostrarMiImagen`.
- Las *entradas* que recibirá esta receta. La receta puede ser una función que reciba entradas, como `abs` o `makePicture`. Las entradas se nombran y colocan entre paréntesis separados por comas. Si su receta no recibe entradas, simplemente escriba `()` para indicar que no hay entradas.
- La línea termina con un signo de dos puntos, `:`.

Lo que viene después son los comandos a ejecutar, uno después del otro, cada vez que se ejecute la receta. Para crear una colección de comandos, definimos un **bloque**. El bloque de comandos que va después de un comando (o *instrucción*) `def` es el que está asociado con el nombre de la función.

La mayoría de los programas reales que hacen cosas útiles, en especial los que crean interfaces de usuario, requieren la definición de más de una función. Imagine que tiene varios comandos `def` en el área del programa. ¿Cómo cree que Python averiguará que una función terminó y empieza una nueva? (En especial debido a que *es* posible definir funciones *dentro de* otras funciones). Python necesita alguna forma de averiguar en dónde termina el *cuerpo de la función*: qué instrucciones son parte de esta función y cuáles son parte de la siguiente.

La respuesta es **sangría**. Todas las instrucciones que forman parte de la definición tienen una ligera sangría después de la instrucción `def`. Recomendamos usar sólo dos espacios: es suficiente para ver, fácil de recordar y simple. (En JES podemos también usar un *tabulador*: presionar una sola vez la tecla de tabulación). Puede introducir la función en el área del programa de la siguiente forma (en donde ␣ indica un espacio, o presionar una sola vez la barra espaciadora):

```
def hola():
␣␣print "Hola"
```

Ahora podemos definir nuestro primer programa. Escriba esto en el área del programa de JES. Cuando termine, guarde el archivo: use la extensión ".py" para indicar un archivo de Python (nosotros lo guardamos como **seleccionarYMostrar.py**).

**Programa 1: seleccionar y mostrar una imagen**

```
def seleccionarYMostrar():
    miArchivo = pickAFile()
    miImagen = makePicture(miArchivo)
    show(miImagen)
```

■

Observará un cuadro delgado de color azul alrededor del cuerpo de la función mientras escribe. El cuadro azul indica los bloques en su programa (figura 2.6). Todos los comandos en el mismo bloque que la instrucción que contiene el cursor (la barra vertical en donde usted escribe) están encerrados en el mismo cuadro azul. Sabrá que tiene la sangría correcta cuando todos los comandos que *espera* que se encuentren en el bloque *estén* en el cuadro.

Una vez que escriba su receta y la guarde, podrá cargarla. Haga clic en el botón LOAD PROGRAM (CARGAR PROGRAMA).

**Tip de depuración: ¡no olvide cargar su programa!**

El error más común en JES es escribir la función, guardarla y después probar la función en el área de comandos antes de cargarla. Es necesario hacer clic en el botón LOAD PROGRAM para que la función esté disponible en el área de comandos.

■

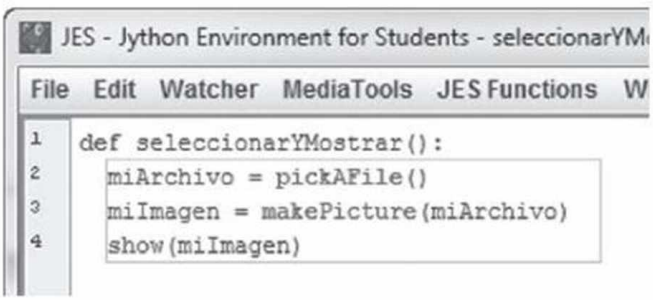



**FIGURA 2.6**
Visualización de los bloques en JES.

Ahora puede ejecutar su programa. Haga clic en el área de comandos. Como no va a recibir ninguna entrada y no va a devolver ningún valor (ésta no es una función matemática estricta), sólo escriba el nombre de su programa como un comando:

```
>>> seleccionarYMostrar()
>>>
```

Podemos definir de manera similar nuestro segundo programa para seleccionar y reproducir un sonido.

**Programa 2: seleccionar y reproducir un sonido**

```
def seleccionarYReproducir():
    miArchivo = pickAFile()
    miSonido = makeSound(miArchivo)
    play(miSonido)
```

■

**Tip de funcionamiento: use los nombres que prefiera**

En la última sección, usamos los nombres miNombrearchivo y milmagen. En este programa usamos miArchivo y milmagen. ¿Acaso importa? A la computadora no le importa en lo absoluto. Podríamos llamar a todas nuestras imágenes miGlifo, o incluso miCosa. A la computadora no le importa qué nombres utilice: usted es el único beneficiado. Elija nombres que sean (a) significativos para usted (de modo que pueda leer y entender su programa), (b) significativos para otros (de modo que cualquiera que vea su programa pueda comprenderlo) y (c) fáciles de escribir. Los nombres con muchos caracteres, como `laImagen QueVoyAAbrirDespuesDeEsto` son significativos y fáciles de leer, pero son muy molestos de escribir.

■

En realidad estos programas tal vez no sean útiles. Tener que seleccionar el archivo una y otra vez es molesto si deseamos que aparezca la misma imagen. Ahora que tenemos el poder de definir programas, podemos definir nuevos programas para que realicen las tareas que deseamos. Definiremos uno que abra una imagen específica y otro que abra un sonido específico.

Use `pickAFile` para obtener el nombre de archivo del sonido o imagen que desee. Necesitaremos el nombre al definir el programa para que reproduzca ese sonido específico o muestre esa imagen específica. Sólo tenemos que configurar el valor de `miArchivo` en forma directa, en vez de hacerlo como resultado de `pickAFile`, colocando la cadena entre comillas directamente en la receta.

**Programa 3: mostrar una imagen específica**

Asegúrese de reemplazar NOMBREARCHIVO en el siguiente programa con la ruta completa a su propio archivo de imagen; por ejemplo, **C:/ip-book/mediasources/barbara.jpg**.

```
def mostrarImagen():
  miArchivo = "NOMBREARCHIVO"
  miImagen = makePicture(miArchivo)
  show(miImagen)
```

■

### Cómo funciona

La variable `miArchivo` recibe el valor del nombre del archivo: el mismo que devolvería la función `pickAFile` si usted eligiera ese archivo. Después creamos una imagen a partir del archivo y la llamamos `miImagen`. Por último, mostramos la imagen en `miImagen`.

**Programa 4: reproducir un sonido específico**

Asegúrese de reemplazar NOMBREARCHIVO en el siguiente programa con la ruta completa a su propio archivo de sonido; por ejemplo, **C:/ip-book/mediasources/helloWorld.wav**.

```
def reproducirSonido():
  miArchivo = "NOMBREARCHIVO"
  miSonido = makeSound(miArchivo)
  play(miSonido)
```

■

**Tip de funcionamiento: copiar y pegar**

Es posible copiar y pegar texto entre las áreas de programa y de comandos. Puede usar `print pickAFile()` para imprimir el nombre de un archivo, luego seleccionarlo y copiarlo (mediante COPY del menú EDIT) para después hacer clic en el área de comandos y pegarlo (mediante PASTE del menú EDIT). De manera similar, puede copiar comandos completos desde el área de comandos hacia el área del programa. Esa es una manera fácil de probar los comandos individuales y después ponerlos todos en una receta, una vez que tenga el orden correcto y estén funcionando. También puede copiar texto dentro del área de comandos. En vez de volver a escribir un comando, selecciónelo, CÓPIELO, PÉGUELO en la línea inferior (¡asegúrese de que el cursor esté al final de la línea!) y presione INTRO para ejecutarlo.

■

## 2.5.1 Recetas de variables: funciones reales similares a las matemáticas que reciben entradas

¿Cómo creamos una verdadera función, como una función en matemáticas, que reciba entradas, tales como `ord` o `makePicture`? ¿Por qué *querríamos* hacerlo?

Una razón importante de usar una variable para especificar la entrada al programa es hacerlo más *general*. Considere el programa 3: `mostrarImagen`. Es para un nombre de archivo específico. ¿Sería útil tener una función que pudiera recibir *cualquier* nombre de archivo, para después crear y mostrar la imagen? Ese tipo de función maneja el caso *general* de crear y mostrar imágenes. A este tipo de generalización le llamaremos **abstracción**. La abstracción conduce a soluciones generales que funcionan en muchas situaciones.

Es muy fácil definir un programa que reciba entrada. Sigue siendo cuestión de *sustitución* y *evaluación*. Colocaremos un nombre dentro de esos paréntesis en la línea `def`. A ese nombre se le conoce algunas veces como **parámetro** o **variable de entrada**.

Cuando evaluamos la función al especificar su nombre con un **valor de entrada** (también conocido como **argumento**) dentro de paréntesis (como `makePicture(miNombrearchivo)` o `show(miImagen)`), el valor de entrada se *asigna* a la variable de entrada. Decimos que la

variable de entrada *asume* el valor de entrada. Durante la ejecución de la función (receta), el valor de entrada se *sustituirá* por la variable.

He aquí cómo se vería un programa que recibe el nombre de archivo como variable de entrada:

**Programa 5: mostrar el archivo de imagen cuyo nombre de archivo se recibe como entrada**

```
def mostrarConNombre(miArchivo):
  miImagen = makePicture(miArchivo)
  show(miImagen)
```

■

Haga clic en el botón LOAD PROGRAM (CARGAR PROGRAMA) para indicar a JES que lea la función o funciones en el área del programa. Si tiene errores en su función, tendrá que corregirlos y hacer clic en el botón LOAD PROGRAM de nuevo. Una vez que haya cargado con éxito sus funciones, podrá usarlas en el área de comandos.

Al escribir

```
mostrarConNombre("C:/ip-book/mediasources/barbara.jpg")
```

en el área de comandos y presionar INTRO, la variable `miArchivo` en la función `mostrarConNombre` asume el valor

```
"C:/ip-book/mediasources/barbara.jpg"
```

Entonces, `miImagen` se referirá a la imagen que resulte de leer e interpretar el archivo. Después se mostrará la imagen.

Podemos crear una función para reproducir un sonido de la misma forma. Podemos escribir más de una función en el área del programa. Sólo agregue la función que se muestra a continuación después de la anterior, y haga clic en el botón LOAD PROGRAM de nuevo.

**Programa 6: reproducir el archivo de sonido cuyo nombre de archivo se recibe como entrada**

```
def reproducirConNombre(miArchivo):
  miSonido = makeSound(miArchivo)
  play(miSonido)
```

■

Pruebe esta función escribiendo lo siguiente en el área de comandos.

```
reproducirConNombre("C:/ip-book/mediasources/croak.wav")
```

Puede crear funciones que reciban más de un parámetro. Sólo separe los parámetros con comas.

**Programa 7: reproducir un archivo mientras se muestra una imagen**

```
def reproducirYMostrar(sArchivo, iArchivo):
  miSonido = makeSound(sArchivo)
  miImagen = makePicture(iArchivo)
  play(miSonido)
  show(miImagen)
```

■

También podemos escribir programas que reciben imágenes o sonidos como valores de entrada. He aquí un programa que muestra una imagen pero recibe el objeto imagen como valor de entrada en vez del nombre de archivo.

**Programa 8: mostrar la imagen que se proporciona como entrada**

```
def mostrarImagen(miImagen):
  show(miImagen)
```

■

En este punto tal vez quiera guardar sus funciones en un archivo, de modo que pueda usarlas otra vez. Haga clic en FILE (ARCHIVO) y después en SAVE PROGRAM (GUARDAR PROGRAMA). A continuación aparecerá una ventana de diálogo de archivos, la cual le permitirá especificar en dónde desea guardar el archivo y cómo llamarlo. Si después sale de JES y vuelve a iniciarlo, podrá usar OPEN PROGRAM (ABRIR PROGRAMA) en el menú FILE para abrir el archivo de nuevo y hacer clic en LOAD PROGRAM para cargar las funciones y poder usarlas.

Ahora, ¿cuál es la diferencia entre la función `mostrarImagen` y la función `show` integrada en JES? Nada en lo absoluto. Sin duda podemos crear una función que proporcione un nuevo nombre a otra función. Si eso hace que su código sea más fácil de entender, entonces es una gran idea.

¿Cuál es el valor de entrada *correcto* para una función? ¿Acaso es mejor introducir un nombre de archivo, o una imagen? ¿Y qué significa "mejor" en este caso? Leerá más sobre todas estas cuestiones más adelante, pero por lo pronto tenemos una respuesta corta: escriba la función que sea más útil para usted. Si definir `mostrarImagen` es más comprensible para usted que usar `show`, entonces eso es útil. Si lo que en verdad desea es una función que se haga cargo de crear la imagen y mostrarla, entonces tal vez la función `mostrarConNombre` le sea más útil.

Lo que hemos hecho hasta ahora es indicar a la computadora que muestre y reproduzca archivos de medios, y crear funciones para facilitarnos la tarea de agrupar estos comandos. Apenas si hemos visto un poco del poder de las computadoras. En primer lugar, una computadora puede *repetir* operaciones, millones y millones de veces, sin siquiera cansarse o aburrirse. En segundo lugar, una computadora puede tomar *elecciones*; es decir, realizar comparaciones y tomar acciones dependiendo de los resultados de esas comparaciones. Usaremos la repetición para procesar todos los píxeles en una imagen, y la elección para procesar sólo algunos píxeles.

## RESUMEN DE PROGRAMACIÓN

En este capítulo hablamos sobre varios tipos de codificaciones de datos (u objetos).

| | |
|---|---|
| Enteros (por ejemplo, 3) | Números sin un punto decimal: no pueden representar fracciones. |
| Números de punto flotante (por ejemplo, 3.0, 3.01) | Números que pueden contener un punto decimal: pueden representar fracciones. |
| Cadenas (por ejemplo, "Hola") | Una secuencia de caracteres (incluyendo espacios, signos de puntuación, etc.) delimitados en ambos extremos por una comilla doble. |
| Nombre de archivo | Una cadena cuyos caracteres representan una ruta y un nombre de archivo base. |
| Imágenes | Codificaciones de imágenes, por lo general provenientes de un archivo JPEG. |
| Sonidos | Codificaciones de sonidos, por lo general provenientes de un archivo WAV. |

He aquí las piezas de programa que introdujimos en este capítulo:

| | |
|---|---|
| `print` | Muestra el valor de una expresión (variable, valor, fórmula, etc.) en su forma de texto. |
| `def` | Define a una función y sus variables de entrada (si tiene). |
| `ord` | Devuelve el valor numérico equivalente (del estándar ASCII) para el carácter de entrada. |
| `abs` | Recibe un número y devuelve su valor absoluto. |
| `pickAFile` | Deja que el usuario seleccione un archivo y devuelve el nombre de la ruta completa como una cadena. No recibe ninguna entrada. |
| `makePicture` | Recibe un nombre de ruta como entrada, lee el archivo y crea una imagen a partir de éste. Devuelve la nueva imagen. |
| `show` | Muestra una imagen proporcionada como entrada. No devuelve nada. |
| `makeSound` | Recibe un nombre de ruta como entrada, lee el archivo y crea un sonido a partir de él. Devuelve el nuevo sonido. |
| `play` | Recibe un sonido y lo reproduce. No devuelve nada. |

## PROBLEMAS

2.1 Preguntas de conceptos de ciencias computacionales:

- ¿Qué es un algoritmo?
- ¿Qué es una codificación?
- ¿Cómo pueden las computadoras representar imágenes como números?
- ¿Qué es la Ley de Moore?

2.2 ¿Qué significa `def`? ¿Qué hace la instrucción `def unaFuncion(x,y):`?

2.3 ¿Qué significa `print`? ¿Qué hace la instrucción `print a`?

2.4 ¿Cuál es la salida de `print 1 / 3`? ¿Por qué recibe esta salida?

2.5 ¿Cuál es la salida de `print 1.3 * 3`? ¿Por qué recibe esta salida?

2.6 ¿Cuál es la salida de `print 1.0 * 3`? ¿Por qué recibe esta salida?

2.7 ¿Cuál es la salida de `print 10 + 3 * 7`? ¿Por qué recibe esta salida?

2.8 ¿Cuál es la salida de `print (10 + 3) * 7`? ¿Por qué recibe esta salida?

2.9 ¿Cuál es la salida de `print "Hola" + "todos"`? ¿Por qué recibe esta salida?

2.10 ¿Cuál es la salida de lo siguiente?

```
>>> a = 3
>>> b = 4
>>> x = a * b
>>> print x
```

2.11 ¿Cuál es la salida de lo siguiente?

```
>>> a = 3
>>> b = -5
>>> x = a * b
>>> print x
```

2.12 ¿Cuál es la salida de lo siguiente?

```
>>> a = 4
>>> b = 2
>>> x = a / b
>>> print x
```

2.13 ¿Cuál es la salida de lo siguiente?

```
>>> a = 4
>>> b = 2
>>> x = b - a
>>> print x
```

2.14 ¿Cuál es la salida de lo siguiente?

```
>>> a = -4
>>> b = 2
>>> c = abs(a)
>>> x = a * c
>>> print x
```

2.15 ¿Cuál es la salida de lo siguiente?

```
>>> nombre = "Barb"
>>> nombre = "Mark"
>>> print nombre
```

2.16 ¿Cuál es la salida de lo siguiente?

```
>>> a = ord("A")
>>> b = 2
```

```
>>> x = a * b
>>> print x
```

2.17 El siguiente código proporciona el mensaje de error que se muestra a continuación. Corrija el código.

```
>>> pickafile()
The error was:pickafile
Name not found globally.
A local or global name could not be found. You need
to define the function or variable before you try
to use it in any way.
```

2.18 El siguiente código proporciona el mensaje de error que se muestra a continuación. Corrija el código.

```
>>> a = 3
>>> b = 4
>>> c = d * a
The error was:d
Name not found globally.
A local or global name could not be found. You need
to define the function or variable before you try
to use it in any way.
```

2.19 ¿Qué hace `show(p)`? (Sugerencia: hay más de una respuesta a esta pregunta).

2.20 Pruebe algunas otras operaciones con cadenas en JES. ¿Qué ocurre si multiplica un número por una cadena, como `3 * "Hola"`? ¿Qué ocurre si intenta multiplicar una cadena por una cadena, como `"a" * "b"`?

2.21 Evaluamos la expresión `pickAFile()` cuando queríamos ejecutar la función llamada `pickAFile`. ¿Pero qué es en sí el nombre `pickAFile`? ¿Qué obtiene al escribir `print pickAFile`? ¿Qué hay sobre `print makePicture`? ¿Qué se imprime y qué cree usted que significa?

## PARA PROFUNDIZAR

El mejor (más detallado, material y elegante) libro de texto de ciencias computacionales es *Structure and Interpretation of Computer Programs* de Abelson, Sussman y Sussman [2]. Es un libro desafiante para leer pero en definitiva vale la pena el esfuerzo. Un libro más reciente y orientado al principiante de programación pero de la misma línea es *How to Design Programs* [12].

Ninguno de estos libros está realmente orientado a los estudiantes que desean programar por simple entretenimiento o porque ofrezcan algo sencillo de hacer. Están orientados a los futuros desarrolladores de software profesionales. Los mejores libros para el estudiante que explora la computación son los de Brian Harvey. Su libro *Simply Scheme* usa el mismo lenguaje de programación que el libro de Abelson y colaboradores, pero es más accesible. Sin embargo, nuestro favorito de esta clase de libros es el conjunto de tres volúmenes de Harvey, *Computer Science Logo Style* [23], que combina las buenas ciencias computacionales con proyectos creativos y divertidos.

CAPÍTULO 3

# Modificación de imágenes mediante el uso de ciclos


3.1 CÓMO SE CODIFICAN LAS IMÁGENES

3.2 MANIPULACIÓN DE IMÁGENES

3.3 MODIFICACIÓN DE LOS VALORES DE LOS COLORES

3.4 CREACIÓN DE UN ATARDECER

3.5 ILUMINACIÓN Y OSCURECIMIENTO

3.6 CREACIÓN DE UN NEGATIVO

3.7 CONVERSIÓN A ESCALA DE GRISES


## Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Comprender cómo se digitalizan las imágenes, aprovechando los límites en la visión humana.
- Identificar diferentes modelos para el color, incluyendo RGB, el modelo más común para las computadoras.
- Manipular los valores de los colores en las imágenes, como aumentar o reducir los valores de rojo.
- Convertir una imagen a colores en escala de grises usando más de un método.
- Obtener el negativo de una imagen.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar una representación matricial al buscar píxeles en una imagen.
- Usar los objetos *imágenes* y *píxeles*.
- Usar la iteración (con un ciclo `for`) para cambiar los valores de los colores de los píxeles en una imagen.
- Anidar bloques de código unos dentro de otros.
- Elegir entre hacer que una función *devuelva* un valor y sólo proporcionar un *efecto secundario*.
- Determinar el *alcance* del nombre de una variable.

## 3.1 CÓMO SE CODIFICAN LAS IMÁGENES

Las imágenes (o gráficos) son una parte importante de cualquier comunicación de medios. En este capítulo veremos cómo se representan las imágenes en una computadora (en su mayoría como imágenes de *mapas de bits*: cada punto o *pixel* se representa por separado) y cómo pueden manipularse. Los capítulos siguientes introducirán representaciones alternativas de imágenes, tales como las imágenes *vectoriales*.

Las imágenes son un arreglo bidimensional de *píxeles*. Describiremos cada uno de estos términos en esta sección.

Para nuestros fines, una imagen es una ilustración almacenada en un archivo JPEG. **JPEG** es un estándar internacional para indicar cómo almacenar ilustraciones de alta calidad pero con poco espacio. JPEG es un formato de **compresión con pérdidas**. Esto significa que se *comprime*, o sea que se hace más pequeño, pero no con el 100% de la calidad del formato original. Aunque por lo general, lo que se pierde es la calidad o nitidez que no se ve o que no podemos ver de todas formas. Una imagen JPEG es adecuada para casi todos los propósitos. El formato BMP es *sin pérdida* pero no está comprimido. Un archivo BMP será mucho mayor que un archivo JPEG para la misma imagen. El formato PNG es sin pérdida y está comprimido.

Un arreglo unidimensional es una secuencia de elementos del mismo tipo. Podemos asignar un nombre a un arreglo y después usar números de índice para acceder a los elementos del arreglo. El primer elemento en el arreglo está en el índice 0. En la figura 3.1 el valor en el índice 0 es 15, el valor en el índice 1 es 12, el valor en el índice 2 es 13 y el valor en el índice 3 es 10.

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 15 | 12 | 13 | 10 |

**FIGURA 3.1**
Un arreglo de ejemplo.

A un arreglo bidimensional también se le conoce como **matriz**. Una matriz es una colección de elementos ordenados en filas y columnas. Esto significa que es posible especificar tanto el índice de fila como el de columna, para poder acceder a un valor en la matriz.

En la figura 3.2 podemos ver el ejemplo de una matriz. Usamos dos números para hacer referencia a los elementos (los números) en esta matriz. Usaremos el primer número para hacer referencia a la columna deseada (porción vertical) y el segundo número para hacer

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 15 | 12 | 13 | 10 |
| 1 | 9 | 7 | 2 | 1 |
| 2 | 6 | 3 | 9 | 10 |

**FIGURA 3.2**
Una matriz de ejemplo.

referencia a la fila deseada (porción horizontal). En las *coordenadas* (0, 1) (horizontal, vertical) encontrará el elemento de la matriz cuyo valor es 9. (0, 0) es 15, (1, 0) es 12 y (2, 0) es 13. Con frecuencia nos referiremos a las coordenadas como (*x*, *y*) (*columna*, *fila*).

Lo que se almacena en cada elemento en la imagen es un **pixel**. La palabra "pixel" es la abreviación de "elemento de imagen". En sentido literal es un punto, y la imagen en general está compuesta de muchos de estos puntos. ¿Alguna vez ha colocado una lupa sobre las imágenes en un periódico o revista, en una televisión o incluso en su propio monitor? Cuando vemos la imagen en la revista o en la televisión no parece que estuviera dividida en millones de puntos discretos, pero lo está.

Podemos obtener una vista similar de los píxeles individuales usando la herramienta de imagen, como se muestra en la figura 3.3. Esta herramienta nos permite realizar un acercamiento en una imagen hasta en un 500%, de modo que podamos ver cada pixel individual. Una forma de abrir esta herramienta es explorar la imagen como se muestra a continuación.

```
>>> archivo = "c:/ip-book/mediasources/caterpillar.jpg"
>>> imagen = makePicture(archivo)
>>> explore(imagen)
```

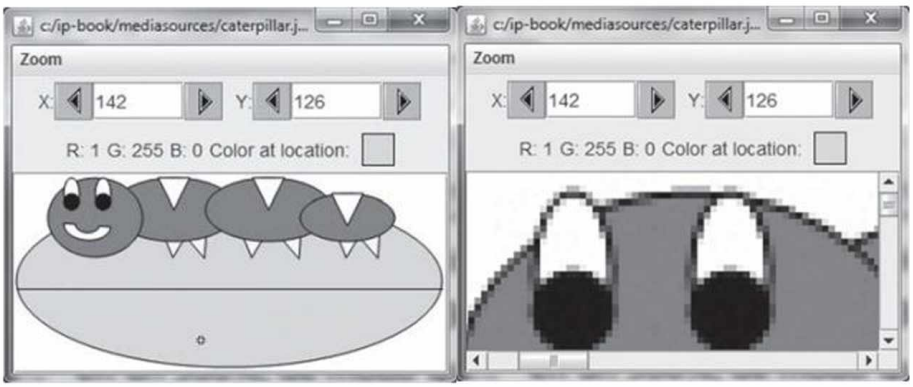



**FIGURA 3.3**
Imagen mostrada en la herramienta de imágenes de JES: imagen al 100% (*izquierda*) y al 500% (*derecha*).

Nuestro aparato sensorial humano no puede distinguir (sin ampliación u otro equipo especial) los pequeños bits en toda la imagen. Los humanos tienen una *agudeza* visual baja: no vemos tantos detalles como, por ejemplo, un águila. En realidad tenemos más de un tipo de sistema de visión en uso en nuestro cerebro y ojos. Nuestro sistema para procesar el color es diferente de nuestro sistema para procesar blanco y negro (o **luminancia**). Por ejemplo, usamos la luminancia para detectar el movimiento y los tamaños de los objetos. En realidad detectamos mejor los detalles de luminancia con los lados de nuestros ojos que con el centro. Ésa es una ventaja evolutiva, ya que nos permite detectar cuando el tigre dientes de sable se esconde a nuestra derecha detrás de algunos arbustos, por ejemplo.

La falta de resolución en la visión humana es lo que hace posible la digitalización de imágenes. Los animales que perciben un mayor detalle que los humanos (por ejemplo, las águilas o los gatos) podrían incluso ver los píxeles individuales. Descomponemos la imagen en elementos más pequeños (píxeles), pero hay suficientes de ellos y son lo bastante pequeños como para que la imagen no se vea entrecortada. Si *podemos* ver los efectos de la digitalización (es

decir, si podemos ver pequeños rectángulos en algunos puntos), entonces tenemos lo que se conoce como *pixelización*: el efecto cuando el proceso de digitalización se vuelve obvio.

La codificación de imágenes es más compleja en estructura que la codificación del sonido. Un sonido se codifica como una colección de números cuyo orden es intrínsecamente lineal: progresa hacia delante en el tiempo. Una imagen tiene dos dimensiones: anchura y altura. Cada posición en esas dimensiones tiene un color que es más complejo que un número individual.

La luz visible es continua: la luz visible tiene cualquier longitud de onda entre 370 y 730 nanómetros (0.00000037 y 0.00000073 metros). Pero nuestra percepción de la luz se limita en cuanto a la forma en que funcionan nuestros sensores de colores. Nuestros ojos tienen sensores que se activan (llegan a un máximo) cerca de los 425 nanómetros (azul), 550 nanómetros (verde) y 560 nanómetros (rojo). Nuestro cerebro determina que un color específico se basa en la retroalimentación de estos tres sensores en nuestros ojos. Hay algunos animales con sólo dos tipos de sensores, como los perros. Estos animales pueden percibir el color, pero no los mismos colores ni de la misma forma que los humanos. Una de las implicaciones interesantes de nuestro aparato sensorial visual limitado es que en realidad percibimos dos tipos de naranja. Existe un naranja *espectral*: una longitud de onda específica que es el color naranja natural. Existe también una mezcla de rojo y amarillo que choca con nuestros sensores de colores de tal forma que lo percibimos como el mismo color naranja.

Mientras sigamos codificando lo que llegue a nuestros tres tipos de sensores de colores, estaremos registrando nuestra percepción humana del color. Así, codificamos cada pixel como una tercia de números. El primer número representa la cantidad de rojo en el pixel, el segundo es la cantidad de verde y el tercero es la cantidad de azul. Podemos crear cualquier color visible por el humano si combinamos la luz roja, verde y azul (figura 3.4). Al combinar las tres luces obtenemos el color blanco puro. Si apagamos las tres obtenemos negro. A esto le llamamos el **modelo de colores RGB**.

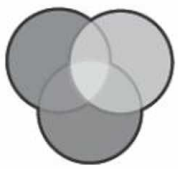


**FIGURA 3.4**
Al mezclar rojo, verde y azul se crean nuevos colores.

Existen otros modelos para definir y codificar colores además del modelo RGB. También está el *modelo de colores HSV*, el cual codifica matiz, saturación y valor (también se le conoce como el modelo de color *HSB* por matiz, saturación y brillo). La ventaja del modelo HSV es que si deseamos algunos cambios, como hacer que un color se vuelva más "claro" u "oscuro", hay que modificar una de esas tres dimensiones (figura 3.5). Hay otro modelo conocido como el *modelo de colores CMYK*, el cual codifica cian, magenta, amarillo y negro. El modelo CMYK es el que utilizan las impresoras. Cian, magenta, amarillo y negro son las tintas que las impresoras combinan para crear colores. Sin embargo, para codificar cuatro elementos en la computadora se requieren más recursos que para tres, por lo que el modelo de colores CMYK es menos popular para los medios digitales que el RGB. Este último es el modelo de color más popular en las computadoras.

Cada componente de color (algunas veces conocido como **canal**) en un pixel se representa por lo general con un solo byte, ocho bits. Estos ocho bits pueden representar 256

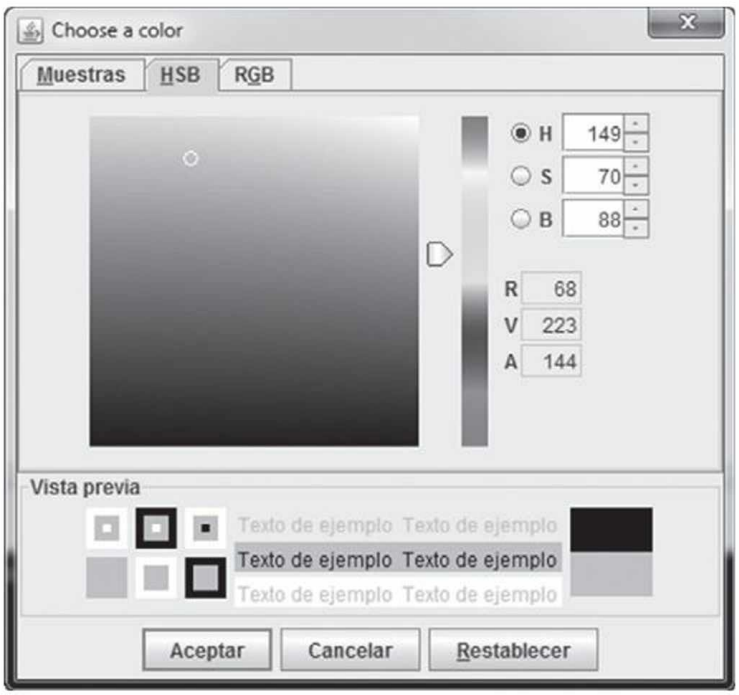



**FIGURA 3.5**
Seleccionar colores usando el modelo de colores HSB.

patrones ($2^8$): 00000000, 00000001, hasta 11111111. Casi siempre usamos estos patrones para representar los valores 0 a 255. Así, cada pixel usa 24 bits para representar los colores. Hay $2^{24}$ patrones posibles de ceros y unos en esos 24 bits, lo cual significa que la codificación estándar para los colores usando el modelo RGB puede representar 16 777 216 colores. En realidad podemos percibir más de 16 millones de colores, pero resulta que eso no importa. No hay ninguna tecnología que siquiera esté cerca de poder replicar todo el espacio de colores completo que podemos ver. Tenemos dispositivos que pueden representar 16 millones de colores distintos, pero esos 16 millones de colores no cubren todo el espacio de color (o luminancia) que podemos percibir. Por lo tanto, el modelo RGB de 24 bits será adecuado hasta que la tecnología avance.

Hay modelos de computadora que usan más bits por pixel. Por ejemplo, existen modelos de 32 bits que usan los ocho bits adicionales para representar *transparencia*; es decir, qué tanto del color "debajo" de la imagen dada debe mezclarse con este color. Algunas veces a esos ocho bits adicionales se les llama el **canal alfa**. Existen otros modelos que usan más de ocho bits para los canales rojo, verde y azul, pero son poco comunes.

En realidad percibimos los bordes de los objetos, el movimiento y la profundidad a través de un sistema de visión *separado*. Percibimos el color a través de un sistema y la *luminancia* (qué tan claros/oscuros son los objetos) a través de otro sistema. La luminancia no es en realidad la *cantidad* de luz, sino nuestra *percepción* de la cantidad de luz. Podemos medir la cantidad de luz (por ejemplo, el número de fotones que se reflejan del color) y mostrar que un punto rojo y un punto azul reflejan cada uno la misma cantidad de luz, pero percibimos el azul como más oscuro. Nuestro sentido de luminancia se basa en las comparaciones con los alrededores. La ilusión óptica en la figura 3.6 señala cómo es que percibimos los niveles de

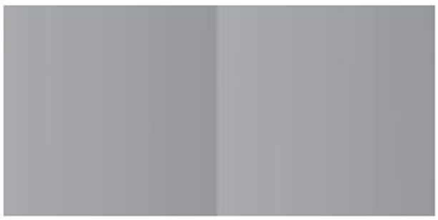


**FIGURA 3.6**
Los extremos de esta figura son los mismos colores de gris, pero los dos cuartos de en medio contrastan de manera pronunciada, por lo que el lado izquierdo se ve más oscuro que el derecho.

gris. Los dos cuartos de los extremos son en realidad el mismo nivel de gris, pero como los dos cuartos de en medio terminan en un contraste pronunciado de claro y oscuro, percibimos una mitad más oscura que la otra, cuando en realidad la diferencia está sólo en la parte de en medio.

La mayoría de las herramientas para permitir a los usuarios elegir colores dejan que éstos especifiquen el color en forma de componentes RGB. El selector de colores en JES (que viene siendo el selector de colores estándar de Java Swing) ofrece un conjunto de barras deslizables para controlar la cantidad de cada color (figura 3.7). Para elegir un color, escriba lo siguiente en el área de comandos.

```
>>> pickAColor()
```

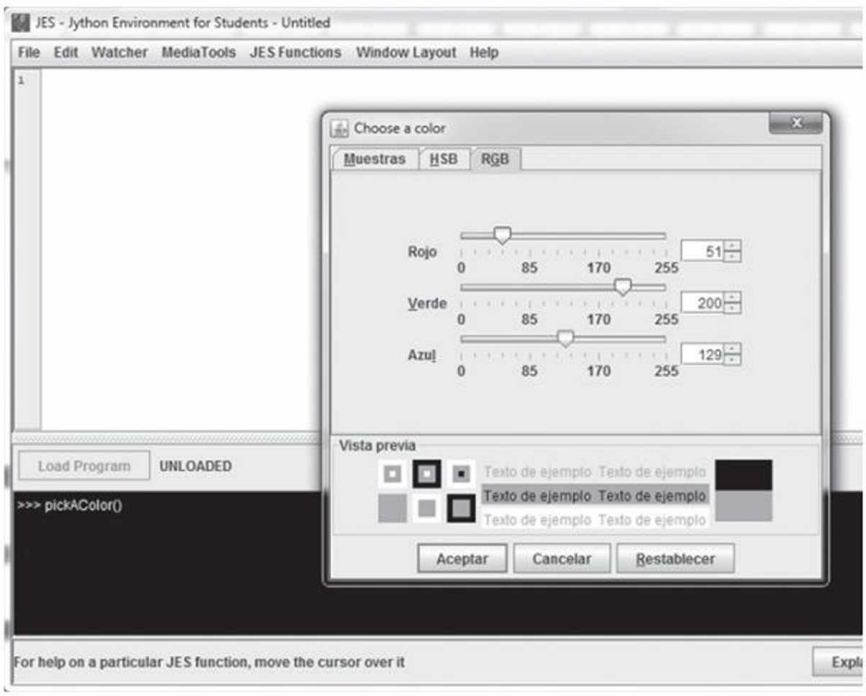



**FIGURA 3.7**
Seleccionar un color mediante controles deslizables de RGB en JES.

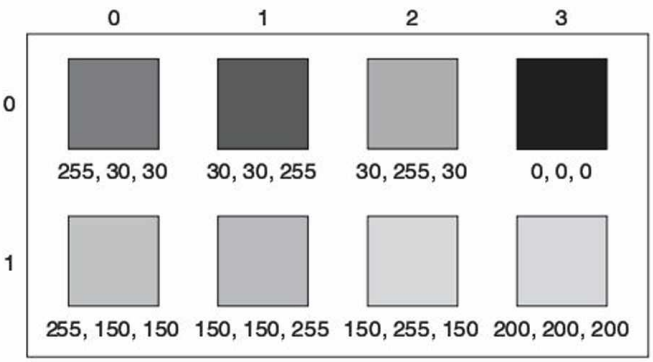



**FIGURA 3.8**
Tercias de RGB en una representación de matriz.

Como dijimos antes, una tercia de (0, 0, 0) (componentes rojo, verde, azul) es el color negro, y (255, 255, 255) es blanco. (255, 0, 0) es rojo puro, pero (100, 0, 0) es rojo también, sólo que más oscuro. (0, 100, 0) es verde mediano y (0, 0, 100) es azul mediano. Cuando el componente rojo es el mismo que el verde y el azul, el color resultante es gris. (50, 50, 50) sería un gris bastante oscuro y (100, 100, 100) es más claro.

La figura 3.8 es una representación de las tercias RGB de píxeles en forma matricial. Así, el pixel en (1, 0) tiene el color (30, 30, 255), lo cual significa que tiene un valor rojo de 30, un valor verde de 30 y un valor azul de 255; es en su mayor parte un color azul, pero no es azul puro. El pixel (2, 1) tiene verde puro, pero también más rojo y azul (150, 255, 150), por lo que es un verde bastante claro. 150 en binario es "10010110" y 255 es "11111111", de modo que un color verde bastante claro se codificaría en color de 24 bits como "100101101111111110010110".

Las imágenes en disco, e incluso en la memoria de la computadora, se guardan por lo general en formato *comprimido*. La cantidad de memoria necesaria para representar cada pixel, inclusive de pequeñas imágenes, es bastante grande (tabla 3.1). Una imagen bastante pequeña de 320 píxeles de largo por 240 píxeles de ancho, con 24 bits por pixel, ocupa 230 400 bytes: esto es aproximadamente 230 *kilobytes* (1000 bytes) o 1/4 *megabyte* (millón de bytes). Un monitor de computadora con 1024 píxeles de ancho y 768 píxeles de alto con 32 bits por pixel ocupa tres megabytes sólo para representar la pantalla.

TABLA 3.1 Número de bytes necesarios para almacenar píxeles en diversos tamaños y formatos

| | **Imagen de 320 × 240** | **Imagen de 640 × 480** | **Imagen de 1024 × 768** |
|---|---|---|---|
| *Color de 24 bits* | 230 400 bytes | 921 600 bytes | 2 359 296 bytes |
| *Color de 32 bits* | 307 200 bytes | 1 228 800 bytes | 3 145 728 bytes |

## 3.2 MANIPULACIÓN DE IMÁGENES

Para manipular imágenes en JES, creamos un objeto imagen a partir de un archivo JPEG y después cambiamos el color de los píxeles en la imagen. Para cambiar el color de un pixel, manipulamos los componentes rojo, verde y azul.

Para crear imágenes usamos `makePicture`. Hacemos que aparezca la imagen mediante `show`.

```
>>> archivo = pickAFile()
>>> print archivo
C:\ip-book\mediasources\beach.jpg
>>> miImagen = makePicture(archivo)
>>> show(miImagen)
>>> print miImagen
Picture, filename C:\ip-book\mediasources\beach.jpg
    height 480 width 640
```

Lo que hace `makePicture` es recoger todos los bytes en el nombre de archivo de entrada, llevarlos a la memoria, cambiar ligeramente su formato y colocar un mensaje sobre ellos para anunciar: "¡Ésta es una imagen!". Al ejecutar `miImagen = makePicture(nombreArchivo)` estamos diciendo: "El nombre de ese objeto imagen (observe el anuncio sobre él) es ahora `miImagen`".

Las imágenes conocen su anchura y su altura. Podemos preguntárselo con `getWidth` y `getHeight`.

```
>>> print getWidth(miImagen)
640
>>> print getHeight(miImagen)
480
```

Podemos obtener cualquier pixel específico de una imagen si usamos `getPixel` con la imagen, además de las coordenadas del pixel deseado. También podemos obtener un arreglo unidimensional de todos los píxeles con `getPixels`. El arreglo unidimensional empieza con todos los píxeles de la primera fila, seguidos de todos los píxeles de la segunda fila, y así en lo sucesivo. Para hacer referencia a los elementos del arreglo usamos `[indice]`.

```
>>> pixel = getPixel(miImagen,0,0)
>>> print pixel
Pixel red=2 green=4 blue=3
>>> pixeles = getPixels(miImagen)
>>> print pixeles[0]
Pixel red=2 green=4 blue=3
```

**Error común: no intente imprimir el arreglo de píxeles: ¡es demasiado grande!**

En sentido literal, `getPixels` devuelve un arreglo de todos los píxeles. Si intenta imprimir el valor de retorno que recibe de `getPixels`, obtendrá la impresión de cada pixel. ¿Cuántos píxeles hay? Bueno, **beach.jpg** tiene una anchura de 640 y una altura de 480. ¿Cuántas líneas se imprimirían? 640 * 480 = ¡307 200! Una impresión de 307 200 líneas es muy grande. Lo más probable es que usted no quiera esperar a que termine. Si lo hace por accidente, sólo salga de JES y vuelva a iniciarlo.

■

Los píxeles saben de dónde provienen. Podemos preguntar por sus coordenadas *x* y *y* con `getX` y `getY`.

```
>>> print getX(pixel)
0
>>> print getY(pixel)
0
```

Cada pixel sabe cómo obtener su color rojo (`getRed`) y establecer su color rojo (`setRed`). (Los colores verde y azul funcionan de manera similar).

```
>>> print getRed(pixel)
2
>>> setRed(pixel,255)
>>> print getRed(pixel)
255
```

También puede solicitar a un pixel su color con `getColor`, y puede establecer el color del pixel con `setColor`. Los objetos de tipo color conocen sus componentes rojo, verde y azul. Puede crear nuevos colores mediante la función `makeColor`.

```
>>> color = getColor(pixel)
>>> print color
color r=255 g=4 b=3
>>> nuevoColor = makeColor(0,100,0)
>>> print nuevoColor
color r=0 g=100 b=0
>>> setColor(pixel,nuevoColor)
>>> print getColor(pixel)
color r=0 g=100 b=0
```

Si cambia el color de un pixel, la imagen a la que pertenece el pixel también cambia.

```
>>> print getPixel(miImagen,0,0)
Pixel red=0 green=100 blue=0
```

**Error común: ver cambios en la imagen**

Si muestra su imagen y luego cambia los píxeles, tal vez se pregunte por qué no ve nada diferente. Las visualizaciones de las imágenes no se actualizan de manera automática. Si ejecuta `repaint` con la imagen, por ejemplo, `repaint(imagen)`, ésta se actualizará.

■

También puede crear colores a partir de `pickAColor`, que le ofrece varias formas de elegir un color.

```
>>> color2 = pickAColor()
>>> print color2
color r=255 g=51 b=51
```

Al terminar de manipular una imagen, puede guardarla con `writePictureTo`.

```
>>> writePictureTo(miImagen,"C:/temp/imagenModificada.jpg")
```

**Error común: la terminación debe ser .jpg**

Asegúrese de terminar su nombre de archivo con ".jpg" para que su sistema operativo pueda reconocerlo como archivo JPEG.

■

**Error común: guardar un archivo rápidamente, ¡y cómo encontrarlo de nuevo!**

¿Qué ocurre si no conoce la ruta completa para un directorio que desee utilizar? Lo único que tiene que especificar es el nombre base.

```
>>> writePictureTo(miImagen,"nueva-imagen.jpg")
```

El problema es encontrar el archivo de nuevo. ¿En qué directorio se guardó? Éste es un problema muy simple de resolver. El directorio predeterminado (el que se asigna si no especificamos una ruta) es en donde se encuentra JES. Si usó `pickAFile()` recientemente, el directorio predeterminado será el directorio del cual eligió el archivo. Si tiene una carpeta de medios estándar (por ejemplo, **mediasources**) en donde guarda sus medios y de donde selecciona sus archivo, ahí es donde se guardarán sus archivos si no especifica una ruta completa.

■

No tenemos que escribir nuevas funciones para manipular imágenes. Podemos hacerlo desde el área de comandos, usando las funciones que acabamos de describir. En el siguiente ejemplo usamos una referencia a un nombre predefinido, *black*. La lista de todos los nombres de colores predefinidos aparece al final de este capítulo, justo antes de los ejercicios.

```
>>> print black
color r=0 g=0 b=0
>>> archivo="C:/ip-book/mediasources/caterpillar.jpg"
>>> imagen=makePicture(archivo)
>>> show(imagen)
>>> pixeles = getPixels(imagen)
>>> setColor(pixeles[0],black)
>>> setColor(pixeles[1],black)
>>> setColor(pixeles[2],black)
>>> setColor(pixeles[3],black)
>>> setColor(pixeles[4],black)
>>> setColor(pixeles[5],black)
>>> setColor(pixeles[6],black)
>>> setColor(pixeles[7],black)
>>> setColor(pixeles[8],black)
>>> setColor(pixeles[9],black)
>>> repaint(imagen)
```

El resultado, que muestra una pequeña línea negra en la parte superior izquierda de la imagen, aparece en la figura 3.9. Esta línea negra tiene 10 píxeles de largo.

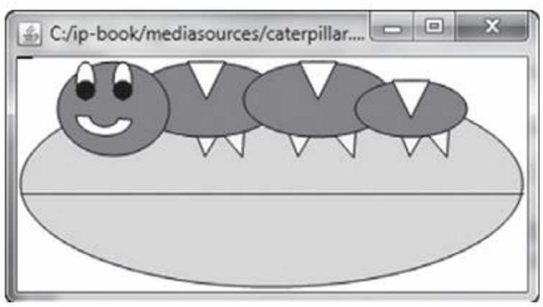



**FIGURA 3.9**
Modificación directa de los colores de los píxeles por medio de comandos: observe la pequeña línea negra en la esquina superior izquierda.

**Tip de funcionamiento: ¡use la ayuda de JES!**

JES cuenta con un excelente sistema de ayuda. ¿Olvidó cuál es la función que necesita? Sólo seleccione un elemento del menú HELP (AYUDA) de JES (figura 3.10: todo tiene hipervínculos, así que puede buscar lo que necesita). ¿Olvidó qué es lo que hace una función que ya está usando? Selecciónela y elija EXPLAIN (EXPLICAR) del menú HELP, para que reciba una explicación de lo que acaba de seleccionar (figura 3.11).

■

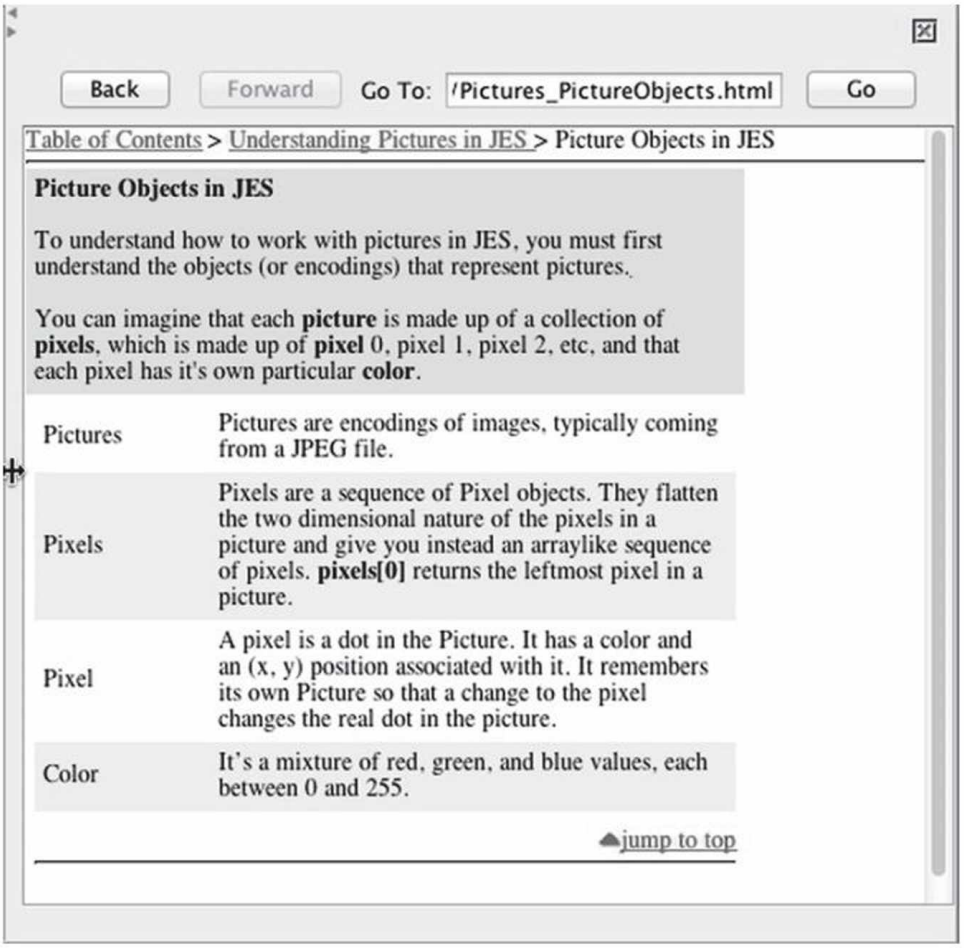



**FIGURA 3.10**
Ejemplo de una entrada del menú HELP (AYUDA) de JES.

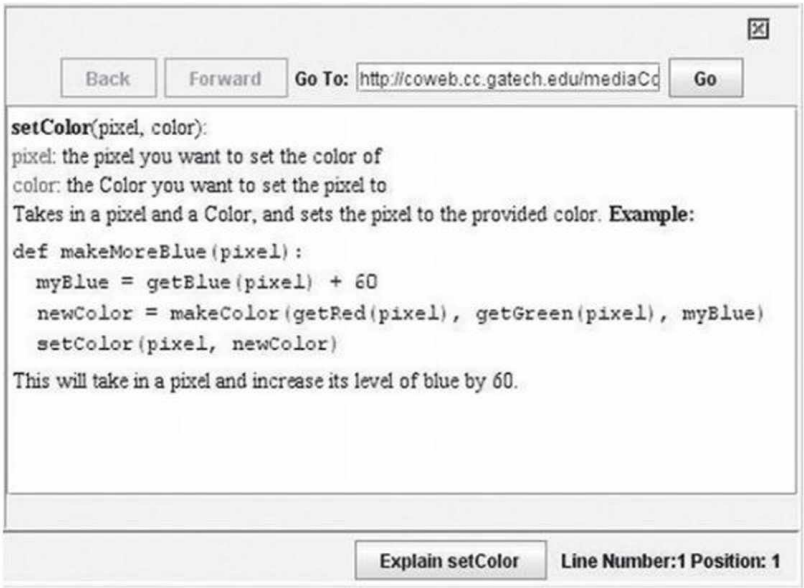



**FIGURA 3.11**
Ejemplo de una entrada de EXPLAIN (EXPLICAR) de JES.

### 3.2.1 Exploración de imágenes

En `http://mediacomputation.org` encontrará la aplicación MediaTools con documentación acerca de cómo empezar a utilizarla. También encontrará un menú MEDIATOOLS en JES. Ambos tipos de MediaTools tienen un conjunto de herramientas de exploración de imágenes, que son de verdadera utilidad para estudiar una imagen. La aplicación MediaTools aparece en la figura 3.12. Con MediaTools puede verificar los valores RGB en cualquier posición $(x, y)$.

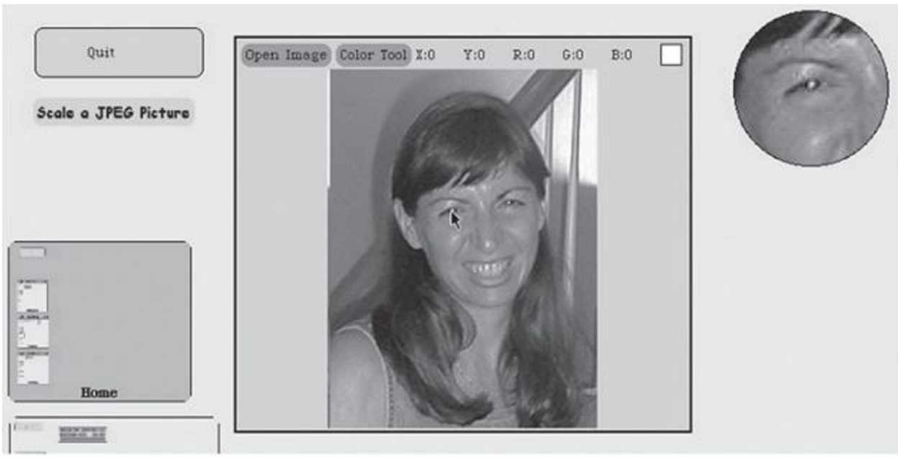



**FIGURA 3.12**
Uso de las herramientas de exploración de imágenes MediaTools.

En la figura 3.12 podemos ver un botón SCALE A JPEG PICTURE (ESCALAR UNA IMAGEN JPEG) del lado izquierdo, debajo del botón QUIT (SALIR). La mayoría de las cámaras digitales modernas capturan imágenes de alta resolución con un gran número de píxeles. Una imagen de 12 megapíxeles tiene literalmente 12 millones de píxeles: son muchos píxeles para procesar en JES. Podemos encoger esa imagen si la escalamos por algún valor menor de 1.0; por ejemplo, un factor de escala de 0.5 encoge la imagen a la mitad en cada dimensión, por lo que tenemos un 75% menos de píxeles. Es conveniente que encoja las imágenes grandes de su cámara digital antes de usarlas en JES.

La herramienta de imágenes de JES se muestra en la figura 3.13. Esta herramienta de imágenes funciona a partir de objetos imagen que hemos definido *y nombrado* en el área de comandos. Si no nombró la imagen, no podrá verla con la herramienta de imágenes de JES. `p = makePicture(pickAFile())` le permitirá definir una imagen y nombrarla como `p`. A continuación podrá explorar la imagen usando `explore(pict)`, o puede elegir PICTURE TOOL (HERRAMIENTA DE IMÁGENES) del menú MEDIATOOLS, con lo cual aparecerá un menú desplegable de los objetos imágenes disponibles (por sus nombres de variables) y tendrá la oportunidad de elegir uno de ellos, haciendo clic en Aceptar.

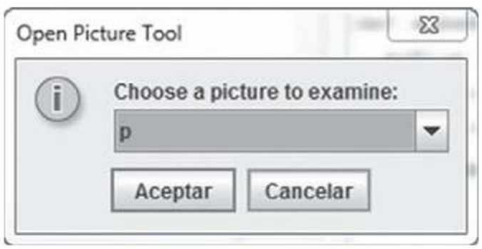



**FIGURA 3.13**
Selección de una imagen en la herramienta de imágenes de JES.

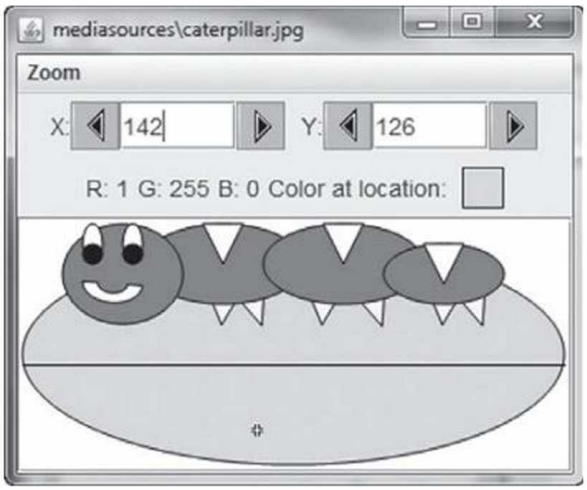



**FIGURA 3.14**
Elegir una imagen en la herramienta de imágenes en JES.

La herramienta de imágenes de JES nos permite explorar una imagen. Podemos realizar un acercamiento o alejamiento al seleccionar un nivel en el menú ZOOM. Si presiona el botón del ratón a medida que mueve el cursor alrededor de la imagen, aparecerán las coordenadas ($x$, $y$) (horizontal, vertical) y los valores RGB del pixel sobre el que se encuentra el cursor del ratón (figura 3.14).

La aplicación MediaTools funciona a partir de archivos en el disco, en vez de los objetos imagen que nombró en su área de comandos. Si desea revisar un archivo antes de cargarlo en JES, use la *aplicación* MediaTools. Haga clic en el cuadro PICTURE TOOLS (HERRAMIENTAS DE IMÁGENES) en MediaTools y se abrirán las herramientas. Use el botón OPEN (ABRIR) para que aparezca un cuadro de selección de archivo: haga clic en los directorios que desea explorar a la izquierda y en las imágenes que desea a la derecha; después haga clic en OK (Aceptar). Cuando aparezca la imagen, habrá varias herramientas distintas disponibles. Mueva el cursor sobre la imagen y mantenga presionado el botón del ratón.

- Aparecerán los valores de rojo, verde y azul para el pixel sobre el que esté colocado. Esto es útil cuando queremos saber cómo se asocian los colores en la imagen a los valores numéricos de rojo, verde y azul. También es útil si vamos a realizar algunos cálculos en los píxeles y queremos verificar los valores.
- Las posiciones $x$ y $y$ aparecerán para el pixel sobre el que estemos colocados. Esto es útil cuando queremos analizar regiones de la pantalla (por ejemplo, si quiere procesar sólo una parte de la imagen). Si conoce el rango de las coordenadas $x$ y $y$ que desea procesar, puede ajustar su ciclo `for` para llegar sólo a esas secciones.
- Por último, hay una lupa que le permite ver los píxeles aumentados. (Puede hacer clic sobre la lupa y moverla a su alrededor).

## 3.3 MODIFICACIÓN DE LOS VALORES DE LOS COLORES

Lo más fácil de hacer con las imágenes es cambiar los valores de color de sus píxeles, para lo cual se modifican los componentes rojo, verde y azul. Podemos obtener efectos radicalmente distintos con sólo ajustar estos valores. Algunos de los *filtros* de Adobe Photoshop hacen justo lo que vamos a hacer en esta sección.

La forma en que vamos a manipular los colores es mediante el cálculo de un *porcentaje* del color original. Si queremos el 50% de la cantidad de rojo en la imagen, vamos a establecer el canal rojo a 0.50 veces lo que tiene justo ahora. Si deseamos aumentar el rojo en un 25%, vamos a establecerlo a 1.25 veces lo que tiene justo ahora. Recuerde que el asterisco (*) es el operador de multiplicación en Python.

### 3.3.1 Uso de ciclos en las imágenes

Lo que podríamos hacer es acceder a cada pixel en la imagen y establecerlo en un nuevo valor de rojo, verde o azul. Digamos que queremos reducir el rojo en un 50%. Podemos escribir entonces el siguiente código:

```
>>> archivo="C:/ip-book/mediasources/barbara.jpg"
>>> imagen=makePicture(archivo)
>>> show(imagen)
>>> pixeles = getPixels(imagen)
>>> setRed(pixeles[0],getRed(pixeles[0]) * 0.5)
>>> setRed(pixeles[1],getRed(pixeles[1]) * 0.5)
```

```
>>> setRed(pixeles[2],getRed(pixeles[2]) * 0.5)
>>> setRed(pixeles[3],getRed(pixeles[3]) * 0.5)
>>> setRed(pixeles[4],getRed(pixeles[4]) * 0.5)
>>> setRed(pixeles[5],getRed(pixeles[5]) * 0.5)
>>> repaint(imagen)
```

Esto es bastante tedioso de escribir, en especial para todos los píxeles incluso en una pequeña imagen. Lo que necesitamos es una forma de decir a la computadora que haga lo mismo una y otra vez. Bueno, no exactamente lo mismo; queremos cambiar lo que sucede de una manera bien definida. Queremos realizar un paso a la vez, o procesar un pixel adicional.

Para ello podemos usar un ciclo `for`. Este ciclo ejecuta cierto bloque de comandos (que usted especifica) para cada elemento en un *arreglo* (que usted le proporciona), en donde cada vez que se ejecutan los comandos una variable específica (que usted nombrará) tendrá el valor de un elemento distinto del arreglo. Un arreglo es una colección ordenada de datos. `getPixels` devuelve un arreglo de todos los objetos píxeles en una imagen de entrada.

Vamos a escribir instrucciones como éstas:

```
for pixel in getPixels(imagen):
```

Analicemos con detalle la instrucción anterior.

- Primero viene el nombre del comando `for`.
- Después viene el nombre de la variable que queremos usar en el código para direccionar (y manipular) los elementos de la secuencia. En este caso usamos la palabra `pixel` porque queremos procesar cada uno de los píxeles de la imagen.
- La palabra `in` es **requerida**: ¡debe escribirla! Al escribir `in` el comando es más comprensible que si se omite esa palabra, por lo que hay un beneficio a cambio de las cuatro pulsaciones de tecla adicionales (espacio-i-n-espacio).
- Después necesitamos un *arreglo*. A la variable `pixel` se le va a asignar cada elemento del arreglo, cada vez que se recorra el ciclo: un elemento del arreglo, una iteración por el ciclo, el siguiente elemento del arreglo, la siguiente iteración por el ciclo. Usamos la función `getPixels` para que genere el arreglo por nosotros.
- Por último necesitamos un signo de dos puntos (":"). Este signo es importante: indica que lo que viene a continuación es un *bloque* (en el capítulo 2 vimos algo sobre los bloques).

A continuación vienen los comandos que vamos a ejecutar para cada pixel. Cada vez que se ejecuten los comandos, la variable (en nuestro ejemplo, `pixel`) será un elemento diferente del arreglo. Los comandos (conocidos como el *cuerpo*) se especifican como un bloque. Esto significa que deben ir después de la instrucción `for`, cada una en su propia línea, ¡y luego deben llevar una sangría de dos espacios más! Por ejemplo, he aquí el ciclo `for` que establece el canal rojo de cada pixel a la mitad del valor original.

```
for pixel in getPixels(imagen):
  valor = getRed(pixel)
  setRed(pixel,valor * 0.5)
```

Vamos a repasar ahora este código.

- La primera instrucción indica que vamos a tener un ciclo `for`, el cual establecerá la variable `pixel` en cada uno de los elementos del arreglo que se obtiene como salida de `getPixels(imagen)`.
- La siguiente instrucción tiene sangría, por lo que forma parte del cuerpo del ciclo `for`: una de las instrucciones que se ejecutarán cada vez que pixel tenga un nuevo valor (sea cual sea el siguiente pixel en la imagen). Obtiene el valor rojo actual en el pixel actual y lo coloca en la variable `valor`.
- La tercera instrucción todavía tiene sangría, por lo que también forma parte del cuerpo del ciclo. Aquí establecemos el valor del canal rojo (`setRed`) del pixel llamado `pixel`, en el valor de la variable `valor` multiplicado por 0.5. Esto reducirá el valor original a la mitad.

Recuerde que lo que viene después de la instrucción `def` de definición de función es *también* un bloque. Si tiene un ciclo `for` dentro de una función, entonces la instrucción `for` ya tiene dos espacios de sangría, por lo que el cuerpo del ciclo `for` (las instrucciones a ejecutar) debe tener una sangría de *cuatro* espacios. El bloque del ciclo `for` está dentro del bloque de la función. A esto se le conoce como **bloque anidado**: un bloque está anidado dentro del otro. He aquí un ejemplo de cómo convertir un ciclo en una función.

```
def reducirRojo(imagen):
  for pixel in getPixels(imagen):
    valor = getRed(pixel)
    setRed(pixel,valor * 0.5)
```

En realidad no tiene que colocar ciclos en las funciones para usarlos. Puede escribirlos en el área de comandos de JES. Este entorno es lo bastante inteligente como para descubrir que usted necesita escribir más de un comando si está especificando un ciclo, por lo que cambia el indicador de >>> a .... Desde luego que no puede averiguar cuándo va a terminar el ciclo, por lo que tendrá que oprimir INTRO sin escribir nada más para indicar a JES que ha terminado con el cuerpo de su ciclo. Tal vez se dé cuenta que en realidad no necesitamos la variable `valor`; podemos tan sólo reemplazar la variable con la función que contiene el mismo valor. He aquí cómo hacerlo en la línea de comandos:

```
>>> for pixel in getPixels(imagen):
...   setRed(pixel,getRed(pixel) * 0.5)
```

Ahora que vimos cómo hacer que la computadora ejecute miles de comandos sin tener que escribir miles de líneas individuales, vamos a probarlo.

### 3.3.2 Aumentar/reducir el color rojo (verde, azul)

Un deseo común al trabajar con imágenes digitales es cambiar el grado de *rojo* (o de verde o azul; casi siempre el rojo) de una imagen. Puede aumentarlo para "calentar" la imagen, o reducirlo para "enfriar" la imagen o lidiar con cámaras digitales con mucho rojo.

El programa que se indica a continuación reduce la cantidad de color al 50% en una imagen de entrada. Usa la variable `p` para representar el pixel actual. Usamos la variable `pixel` en la función anterior. Eso no importa: el nombre puede ser lo que nosotros queramos.

**Programa 9: reducir la cantidad de rojo en una imagen al 50%**

```
def reducirRojo(imagen):
  for p in getPixels(imagen):
    valor=getRed(p)
    setRed(p,valor*0.5)
```

■

Ahora escriba el código anterior en su área de programa de JES. Haga clic en LOAD PROGRAM (CARGAR PROGRAMA) para que Python procese la función (asegúrese de guardarla, con un nombre como **reducirRojo.py**) y así el nombre `reducirRojo` represente a esta función. Siga el ejemplo que se indica a continuación para tener una mejor idea de cómo funciona todo esto.

Esta receta recibe una imagen como entrada: la que usaremos para obtener los píxeles. Para obtener una imagen necesitamos un nombre de archivo y después tenemos que crear una imagen a partir de ese nombre. También podemos abrir una herramienta de imágenes de JES en una imagen, usando la función `explore(imagen)`. Esto creará una copia de la imagen actual y la mostrará en la herramienta de imágenes de JES. Después de aplicar la función `reducirRojo` a la imagen podemos explorarla de nuevo y comparar las dos imágenes, una al lado de la otra. Por lo tanto, la receta puede usarse así:

```
>>> archivo="C:/ip-book/mediasources/Katie-smaller.jpg"
>>> imagen=makePicture(archivo)
>>> explore(imagen)
>>> reducirRojo(imagen)
>>> explore(imagen)
```

**Error común: paciencia — los ciclos for siempre terminan**

El error más común con este tipo de código es darse por vencido y oprimir el botón STOP (DETENER) antes de que termine por su cuenta. Si utiliza un ciclo `for`, el programa *siempre* se detendrá. Pero tal vez tarde un minuto completo (¡o dos!) por algunas de las manipulaciones que vamos a realizar; en especial si su imagen de origen es grande.

■

La imagen original y su versión con el color rojo reducido aparecen en la figura 3.15. Sin duda, 50% es *mucho* rojo para reducir. La imagen se ve como si se hubiera tomado a través de un filtro de azul. Observe que el valor rojo del primer pixel era 133 y cambió a 66.

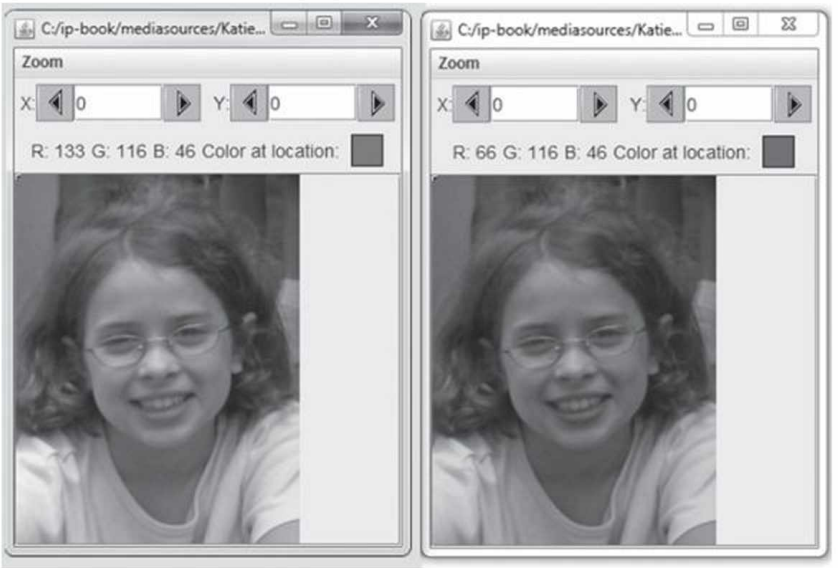



**FIGURA 3.15**
La imagen original (*izquierda*) y la versión con rojo reducido (*derecha*).

**Idea de ciencias computacionales: la habilidad más importante de aprender es rastrear**

La habilidad más importante que usted puede desarrollar en su primer curso de programación es la habilidad de *rastrear* su programa (algunas veces a esto se le conoce como *ejecución paso a paso* o *recorrido* de su programa). Rastrear su programa es recorrerlo línea por línea y averiguar lo que ocurre. Con sólo ver un programa, ¿puede *predecir* lo que hará? Debería ser capaz de hacerlo al reflexionar sobre lo que hace. Con el tiempo deseará ser capaz de *diseñar* nuevos programas también, pero tiene que entender muy bien cómo funcionan sus programas antes de poder crear nuevos de manera efectiva.

■

## Cómo funciona

Vamos a *rastrear* la función que reduce el rojo para ver cómo funciona. Queremos entrar en el punto en donde acabamos de llamar a `reducirRojo`:

```
>>> archivo="C:/ip-book/mediasources/Katie-smaller.jpg"
>>> imagen=makePicture(archivo)
>>> explore(imagen)
>>> reducirRojo(imagen)
>>> explore(imagen)
```

¿Qué ocurre ahora? En realidad, `reducirRojo` representa a la función que vimos antes, por lo que se empieza a ejecutar.

```
def reducirRojo(imagen):
  for p in getPixels(imagen):
    valor=getRed(p)
    setRed(p,valor*0.5)
```

La primera línea que ejecutamos es `def reducirRojo(imagen):`. Esta línea dice que la función debe esperar cierta entrada, la cual se llamará `imagen` durante la ejecución de la función.

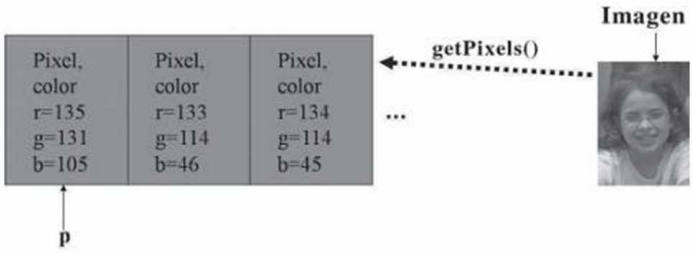
**Idea de ciencias computacionales: los nombres dentro de las funciones son diferentes de los nombres fuera de las funciones**

Los nombres dentro de una función (como `imagen`, `p` y `valor` en el ejemplo de `reducirRojo`) son *completamente* diferentes de los nombres en el área de comandos o en cualquier otra función. Decimos que tienen un *alcance* distinto, lo cual significa el área en donde se define el nombre. Dentro de la función `reducirRojo`, el nombre `p` está en alcance *local*. No puede usarse desde el área de comandos. Los nombres creados en el área de comandos tienen un alcance más *global*. Esos nombres pueden usarse dentro de funciones definidas en el área del programa.

■

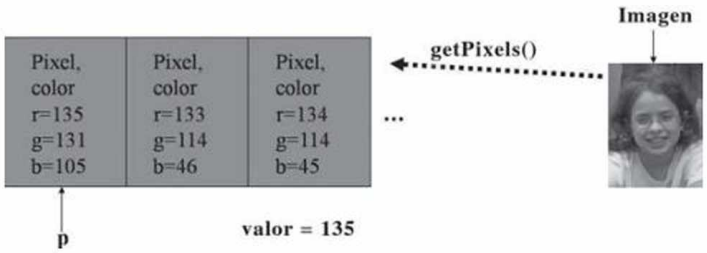
Dentro de la computadora podemos imaginar cómo se ve ahora: hay cierta asociación entre la palabra `imagen` y el objeto imagen que proporcionamos como entrada.

Ahora llegamos a la línea `for p in getPixels(imagen):`. Esto significa que todos los píxeles de la imagen están alineados (dentro de la computadora) como una secuencia (en el arreglo) y que a la variable `p` se le debe asignar (debe asociarse con) el primero. Podemos imaginar que dentro de la computadora ahora se ve así:

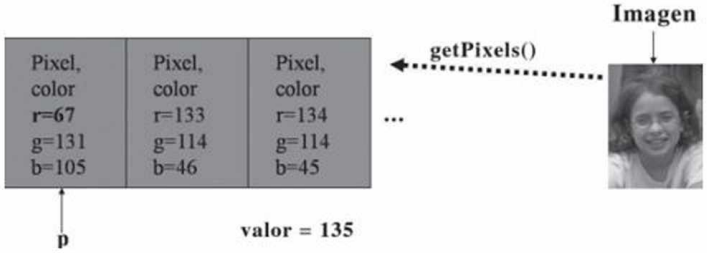



Cada pixel tiene sus propios valores RGB. La variable `p` apunta al primer pixel. Observe que la variable `imagen` todavía está ahí; tal vez ya no la usemos, pero sigue ahí.

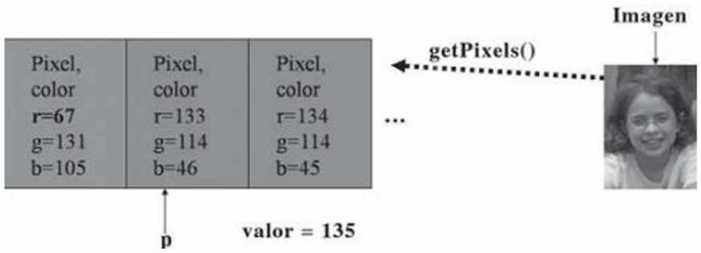
Ahora estamos en `valor=getRed(p)`. Esto simplemente agrega otro nombre a los que la computadora ya está rastreando por nosotros y le proporciona un valor numérico simple.



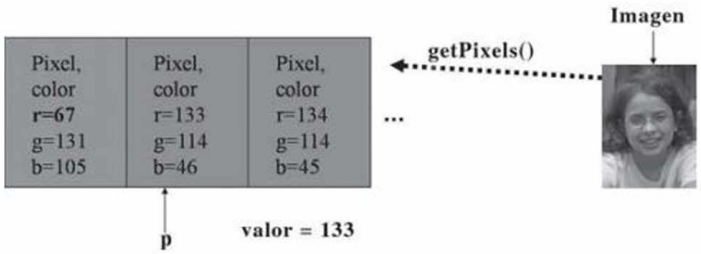

Por último, estamos al final del ciclo. La computadora ejecuta la instrucción `setRed(p,valor*0.5)`, que cambia el canal rojo del píxel `p` al 50% del `valor`. El valor de `p` es impar, por lo que obtendríamos `67.5` al multiplicarlo por `0.5`, pero como vamos a colocar el resultado en un entero, la parte fraccionaria del número tan sólo se descarta. Así, el valor rojo original de 135 cambia a 67.



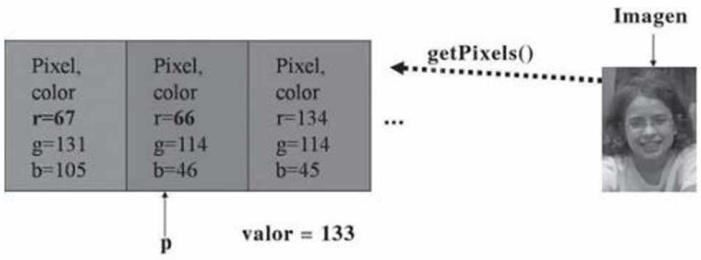

Lo que ocurre a continuación es muy importante: ¡el ciclo vuelve a empezar! Regresamos al ciclo `for` y tomamos el *siguiente* valor en el arreglo. El nombre p se asocia con ese siguiente valor.




Obtenemos un nuevo valor para `valor` en `valor=getRed(p)`, por lo que ahora `valor` es 133, en vez del 135 que correspondía al primer pixel.




Y después cambiamos el canal rojo de *ese* pixel.




Después de un tiempo llegamos a la figura 3.15. Seguimos avanzando por todos los píxeles en la secuencia y cambiando todos los valores de rojo.

### 3.3.3 Prueba del programa: ¿en verdad funcionó?

¿Cómo sabemos que lo que hicimos funcionó en realidad? Seguro, *algo ocurrió* a la imagen, pero ¿en realidad redujimos el color rojo? ¿En un 50%?

**Tip de funcionamiento: ¡no sólo confíe en sus propios programas!**

Es fácil confundirse y pensar que su programa funcionó. Después de todo, usted le indicó a la computadora que hiciera algo, por lo que no debería sorprenderle que la computadora hiciera lo que usted quería. Pero las computadoras son de verdad estúpidas: no pueden averiguar qué es lo que usted quiere. Sólo hacen lo que usted les pide. Es muy fácil que algo salga *casi* perfecto. Asegúrese de verificarlo.

■

Podemos comprobarlo de varias formas. Una de ellas es mediante la herramienta de imágenes de JES. Puede verificar los valores RGB en las mismas coordenadas $x$ y $y$, tanto en la imagen original como en la imagen resultante, usando la herramienta de imagen. Haga clic con el botón del ratón y arrastre el cursor a una ubicación para verificar la imagen original, después escriba las mismas coordenadas $x$ y $y$ en la herramienta de la imagen resultante y presione INTRO. Esto le mostrará los valores de color en la ubicación $x$ y $y$ deseada, tanto en la imagen original como en la resultante (figura 3.16).

También podemos usar las funciones que conocemos en el área de comandos, para verificar los valores del color rojo de los píxeles individuales.

```
>>> archivo = pickAFile()
>>> imagen = makePicture(archivo)
>>> pixel = getPixel(imagen,0,0)
>>> print pixel
Pixel red=168 green=131 blue=105
>> reducirRojo(imagen)
>>> nuevoPixel = getPixel(imagen,0,0)
>>> print nuevoPixel
Pixel red=84 green=131 blue=105
>>> print 168 * 0.5
84.0
```

**FIGURA 3.16**
Uso de la herramienta de imágenes de JES para convencernos de que se redujo el color rojo.

### 3.3.4 Cambiar un color a la vez

Ahora aumentaremos el color rojo en la imagen. Si al multiplicar el componente rojo por 0.5 se redujo, entonces al multiplicarlo por algo mayor a 1.0 debe aumentarlo.

**Programa 10: aumentar el componente rojo en un 20%**

```
def aumentarRojo(imagen):
    for p in getPixels(imagen):
      valor=getRed(p)
      setRed(p,valor*1.2)
```

■

#### Cómo funciona

Usaremos `aumentarRojo` con el mismo tipo de instrucciones en el área de comandos que usamos para `reducirRojo`. Al escribir algo como `aumentarRojo(imagen)`, ocurre el mismo tipo de proceso. Obtenemos todos los píxeles de la imagen de entrada `imagen` (cualquiera que sea) y después asignamos la variable `p` al primer pixel en la lista. Obtenemos su valor de rojo (por decir, 100) y lo nombramos `valor`. Asignamos el valor rojo del pixel actual representado por el nombre `p` a `1.2 * 100`, o 120. Después repetimos el proceso para *cada* pixel `p` en la imagen de entrada.

¿Qué ocurre si aumentamos el rojo en una imagen que tenga mucho rojo y algunos de los valores de rojo resultantes exceden de 255? Hay dos opciones en este caso. El valor puede recortarse a un máximo de 255, o podemos regresarlo usando el operador módulo (residuo). Por ejemplo, si el valor es 200 e intenta duplicarlo a 400, puede permanecer en 255 o regresarse a 144 (400 – 256). Pruebe y vea qué ocurre, después verifique cómo se establecieron sus opciones. JES provee una opción para recortar los valores de color a 255 o regresarlos.

Para cambiar esta opción, haga clic en EDIT (EDITAR) en el menú y después seleccione OPTIONS (OPCIONES). La opción a modificar es MODULO PIXEL COLOR VALUES BY 256 (MÓDULO VALORES DE COLOR DE PIXEL POR 256). Para regresar el valor se usa la operación *módulo*. La opción de evitar que aumente a más de 255 es *no* usar módulo.

Podemos incluso deshacernos de un componente de color por completo. El siguiente programa borra el componente azul de una imagen.

**Programa 11: borrar el componente azul de una imagen**

```
def borrarAzul(imagen):
  for p in getPixels(imagen):
    setBlue(p,0)
```

■

## 3.4 CREACIÓN DE UN ATARDECER

En JES podemos manipular más de una imagen a la vez sin problemas. Una vez, Mark quería generar un atardecer a partir de la escena de una playa. Su primer intento fue aumentar el color rojo pero no funcionó. Algunos de los valores de rojo en una imagen dada son bastante altos. Si pasa más allá de 255 para el valor de un canal, la opción predeterminada es *regresar*. Si establece el rojo (mediante `setRed`) de un pixel a 256, en realidad obtendrá *cero*. Entonces, al aumentar el rojo se crean puntos de color azul-verde (sin rojo).

La segunda idea de Mark fue que tal vez lo que ocurre en un atardecer es que hay *menos* azul y verde, con lo cual se *enfatiza* el rojo sin tener que aumentarlo. He aquí el programa que escribió para eso:

**Programa 12: creación de un atardecer**

```
def crearAtardecer(imagen):
  for p in getPixels(imagen):
    valor=getBlue(p)
    setBlue(p,valor*0.7)
    valor=getGreen(p)
    setGreen(p,valor*0.7)
```

■

### Cómo funciona

Como en los ejemplos anteriores, tomamos una `imagen` de entrada y hacemos que la variable `p` represente a cada pixel en la imagen de entrada. Obtenemos cada componente azul y luego lo establecemos de nuevo, después de multiplicar su valor original por 0.70. Después hacemos lo mismo con el verde. En efecto, estamos cambiando los canales azul y verde, reduciendo cada uno en un 30%. El efecto funciona muy bien, como podemos ver en la figura 3.17.

### 3.4.1 Comprensión de las funciones

Es probable que tenga muchas preguntas sobre las funciones en este punto. ¿Por qué escribimos estas funciones de esa forma? ¿Cómo es que reutilizamos nombres de variables como `imagen`, tanto en la función como en el área de comandos? ¿Hay otras formas de escribir estas funciones? ¿Acaso existen funciones que sean mejores o peores?

Como siempre estamos seleccionando un archivo (o escribiendo el nombre de un archivo) y *luego* creamos una imagen antes de llamar a una de nuestras funciones de manipulación

**FIGURA 3.17**
Escena original de la playa (*izquierda*) y en un atardecer (falso) (*derecha*).

de imágenes, para *después* mostrar o explorar la imagen, es natural preguntar por qué no las integramos al entorno. ¿Por qué no *todas* las funciones tienen a `pickAFile()` y `makePicture` integradas?

Usamos las funciones de manera que sean más *generales* y *reutilizables*. Queremos que cada función haga sólo una tarea, para que podamos usar la función otra vez en un nuevo contexto en donde necesitemos que se realice esa tarea. Un ejemplo podría aclarar mejor esto. Considere el programa para crear un atardecer (programa 12). Funciona al reducir los colores verde y azul, cada uno en un 30%. ¿Qué pasaría si modificáramos esta función de modo que llamara a dos funciones *más pequeñas* que realizaran sólo las dos partes de la manipulación? Terminaríamos con algo como el programa 13.

**Programa 13: creación de un atardecer en tres funciones**

```
def crearAtardecer2(imagen):
  reducirAzul(imagen)
  reducirVerde(imagen)

def reducirAzul(imagen):
  for p in getPixels(imagen):
    valor=getBlue(p)
    setBlue(p,valor*0.7)

def reducirVerde(imagen):
  for p in getPixels(imagen):
    valor=getGreen(p)
    setGreen(p,valor*0.7)
```

■

## Cómo funciona

Lo primero que sale a relucir es que esto sí funciona de verdad. La función `crearAtardecer` hace lo mismo aquí que en la receta anterior. La función `crearAtardecer` toma una imagen de entrada y después llama a `reducirAzul` con la misma imagen de entrada. La función `reducirAzul` hace que `p` represente a cada pixel en la imagen de entrada y reduce el azul de cada pixel en un 30% (al multiplicarlo por 0.7). Después `reducirAzul` termina y el *flujo de control* (es decir, lo que determina qué instrucción se ejecuta a continuación) regresa a `crearAtardecer` para ejecutar la *siguiente* instrucción. Ésta es la llamada a la función `reducirVerde` con la misma imagen de entrada. Como antes, `reducirVerde` toca cada uno de los píxeles y reduce el valor de verde en un 30%.

**Tip de funcionamiento: uso de varias funciones**

Es perfectamente válido tener varias funciones en un área del programa, guardadas en un archivo. Esto puede hacer que las funciones sean más fáciles de leer y reutilizar.

■

Es perfectamente válido que una función (`crearAtardecer` en este caso) use otra función escrita por el programador en el mismo archivo (`reducirAzul` y `reducirVerde`). Puede utilizar `crearAtardecer` de igual forma que antes. Es la misma receta (indica a la computadora que haga lo mismo) pero con distintas funciones. La receta anterior hacía todo en una función y ésta lo hace en tres. De hecho, puede usar también `reducirAzul` y `reducirVerde`: cree una imagen en el área de comandos y proporciónela como entrada a cualquiera de ellas. Funcionarán igual que `reducirRojo`.

La diferencia es que la función `crearAtardecer` es un poco más simple de leer. Indica con claridad lo siguiente: "Crear un atardecer significa reducir el azul y reducir el verde". *Es* importante que sea simple de leer.

**Idea de ciencias computacionales: los programas son para las personas**

A las computadoras no les importa en lo absoluto la apariencia de un programa. Los programas están escritos para comunicarse con las *personas*. Hacer programas fáciles de leer y comprender significa que se pueden modificar y reutilizar con más facilidad, además de que comunican con efectividad los procesos a otros humanos.

■

¿Qué pasaría si hubiéramos escrito `reducirAzul` y `reducirVerde` con `pickAFile`, `show` y `repaint` en su interior? El programa nos pediría dos veces la imagen: una en cada función. Puesto que escribimos estas funciones *sólo* para reducir el azul y reducir el verde ("una y sólo una cosa"), podemos usarlas en funciones nuevas como `crearAtardecer`.

Ahora digamos que colocamos `pickAFile` y `makePicture` dentro de `crearAtardecer`. Las funciones `reducirAzul` y `reducirVerde` son totalmente flexibles y reutilizables de nuevo. Pero ahora `crearAtardecer` es menos flexible y reutilizable. ¿Acaso eso importa mucho? No, si lo único que nos importa es que la función proporcione una apariencia de atardecer a un solo cuadro. Pero ¿qué tal si más adelante desea crear una película con unos cuantos cuadros, y en cada uno de ellos desea una apariencia de atardecer? ¿En realidad desea seleccionar cada uno de esos cientos de cuadros? ¿O desearía mejor crear un ciclo para recorrer los cuadros (lo cual aprenderemos a hacer en unos cuantos capítulos) y enviar cada uno de ellos como entrada para la forma *más general* de `crearAtardecer`? Ésta es la razón por la que las funciones deben ser generales y reutilizables: nunca se sabe cuándo vayamos a necesitar usar una función de nuevo en un contexto más amplio.

**Tip de funcionamiento: no empiece tratando de escribir aplicaciones**

Con frecuencia los programadores principiantes quieren escribir aplicaciones completas que un usuario no técnico pueda usar. Tal vez usted quiera escribir una aplicación `crearAtardecer` que obtenga una imagen para un usuario y genere un atardecer. Crear buenas interfaces de usuario que todo el mundo pueda usar es un trabajo duro. Empiece más despacio. Ya es bastante difícil crear una función reutilizable que reciba una imagen como entrada. Podrá trabajar en las interfaces de usuario más adelante.

■

También podríamos escribir estas funciones con nombres de archivo explícitos, si colocamos la siguiente instrucción al principio de una de las recetas:

```
archivo="C:/ip-book/mediasources/bridge.jpg"
```

De esta manera no tendríamos que escribir el nombre de un archivo cada vez que ejecutáramos las recetas. Pero entonces las funciones sólo trabajarían para ese archivo y, si quisiéramos que trabajaran para algún otro archivo, tendríamos que modificarlas. ¿En realidad desea modificar la función cada vez que la utilice? Es más fácil dejar la función sola y cambiar la imagen que le proporcione.

Desde luego que podríamos cambiar cualquiera de nuestras funciones para que recibiera el nombre de un archivo en vez de una imagen. Por ejemplo, podríamos escribir:

```
def crearAtardecer3(nombrearchivo):
  reducirAzul(nombrearchivo)
  reducirVerde(nombrearchivo)

def reducirAzul(nombrearchivo):
  imagen = makePicture(nombrearchivo)
  for p in getPixels(imagen):
    valor=getBlue(p)
    setBlue(p,valor*0.7)

def reducirVerde(nombrearchivo):
  imagen=makePicture(nombrearchivo)
  for p in getPixels(imagen):
    valor=getGreen(p)
    setGreen(p,valor*0.7)
```

¿Es este código mejor o peor que el que vimos antes? En cierto nivel, no importa; podemos trabajar con imágenes o nombres de archivos, lo que tenga más sentido para nosotros. Sin embargo, la versión que usa el nombre de archivo como entrada tiene varias desventajas. Por una parte, ¡no funciona! La imagen se crea tanto en `reducirVerde` como en `reducirAzul`, pero después no se guarda, por lo que se pierde al final de la función. La versión anterior de `crearAtardecer2` (y sus *subfunciones*, las funciones a las que llama) trabaja debido a los *efectos secundarios*: la función no devuelve nada, pero modifica el objeto de entrada en forma directa.

Podríamos corregir la pérdida de la imagen guardando el archivo en el disco después de terminar con cada función, pero entonces las funciones estarían haciendo más de "una y sólo una cosa". También tenemos la ineficiencia de crear la imagen dos veces, y si añadimos lo de guardarla, guardaríamos la imagen dos veces. De nuevo, las mejores funciones hacen "una y sólo una cosa".

Incluso las funciones más grandes como `crearAtardecer2` hacen "una y sólo una cosa". Aquí, `crearAtardecer2` crea una imagen con apariencia de atardecer. Para ello, *reduce* los colores verde y azul. Llama a otras dos funciones para hacerlo. Al final terminamos con una *jerarquía* de objetivos: la "una y sólo una cosa" que se va a realizar. `crearAtardecer` hace esta única cosa al pedir a otras dos funciones que hagan su única cosa. A esto le llamamos **descomposición jerárquica** (descomponer un problema en partes más pequeñas, después descomponer las partes más pequeñas hasta obtener algo que podamos programar con facilidad) y es muy poderosa para crear programas complejos a partir de piezas que podamos comprender.

Los nombres en las funciones están *completamente* separados de los nombres en el área de comandos. La única manera de pasar datos (imágenes, sonidos, nombres de archivos,

números) del área de comandos a una función es pasarlos como entrada a la función. Dentro de la función, puede usar los nombres que desee. Los nombres que defina primero dentro de la función (como `imagen` en el último ejemplo) y los nombres que utilice para representar los datos de entrada (como `nombrearchivo`) *sólo* existirán mientras la función se esté ejecutando. Cuando la función termine, los nombres de las variables literalmente ya no existirán.

En realidad esto es una ventaja. En secciones anteriores dijimos que la asignación de nombres es muy importante para los científicos computacionales: nombramos todo, desde los datos hasta las funciones. Pero si cada nombre pudiera significar una y sólo una cosa *por siempre*, nos quedaríamos sin nombres. En el lenguaje natural, las palabras significan distintas cosas en contextos diferentes (por ejemplo, "Hazle como quieras" y "Yo como cada vez que tengo hambre"). Una función es un contexto diferente: los nombres pueden significar algo distinto de lo que significan fuera de esa función.

Algunas veces calculará algo dentro de una función que desee regresar al área de comandos o a una función que haya hecho una llamada. Ya vimos funciones que envían un valor de salida, como `pickAFile`, que envía un nombre de archivo de salida. Si utilizó `makePicture` dentro de una función, tal vez desee enviar como salida la imagen que creó dentro de la función. Para ello puede usar `return`, lo cual veremos más adelante.

El nombre que asigne a la entrada de una función puede considerarse como *marcador de posición*. Cada vez que aparezca el marcador de posición, imagine que en vez de ello aparecen los datos de entrada. Así, en una función como:

```
def reducirRojo(imagen):
  for p in getPixels(imagen):
    valor=getRed(p)
    setRed(p,valor*0.5)
```

vamos a llamar a `reducirRojo` con una instrucción como `reducirRojo(miImagen)`. La imagen que se encuentre en `miImagen` se *conocerá como* `imagen` mientras que `reducirRojo` se ejecute. Durante esos pocos segundos, `imagen` en `reducirRojo` y `miImagen` en el área de comandos se *referirán a la misma imagen*. Al cambiar los píxeles en una se cambiarán los píxeles en la otra.

Ya hablamos sobre distintas formas de escribir la misma función: unas mejores y otras peores. Existen otras formas que son muy equivalentes y otras que son mucho mejores. Consideremos unas cuantas maneras más en las que podemos escribir funciones.

Podemos pasar más de una entrada a la vez. Considere esta versión de `reducirRojo`:

```
def reducirRojo(imagen, cantidad):
  for p in getPixels(imagen):
    valor=getRed(p)
    setRed(p,valor*cantidad)
```

Para usar esta función escribimos algo como `reducirRojo(miImagen, 0.25)`. Este uso reduciría el rojo en un 75%. Podríamos decir `reducirRojo(miImagen, 1.25)` para *aumentar* el rojo en un 25%. Tal vez un mejor nombre para esta función sea `cambiarRojo`, ya que es lo que hace ahora: una forma general de cambiar toda la cantidad de rojo en una imagen. Es una función bastante útil y poderosa.

En el programa 11 utilizamos este código:

```
def borrarAzul(imagen):
  for p in getPixels(imagen):
    setBlue(p,0)
```

También podríamos escribir la misma receta de esta forma:

```
def borrarAzul(imagen):
  for p in getPixels(imagen):
    valor = getBlue(p)
    setBlue(p,valor*0)
```

Es importante tener en cuenta que esta función logra *exactamente* lo mismo que la receta anterior. Ambas establecen el canal azul de todos los píxeles en cero. Una ventaja de esta última función es que tiene la misma *forma* que todas las demás funciones modificadoras de color que hemos visto. Esto puede hacerla más comprensible, lo cual es útil. Es un poco menos eficiente: en realidad no es necesario *obtener* el valor azul antes de ponerlo en cero, ni es necesario multiplicar por cero cuando sólo queremos el valor de cero. Esta función en realidad está haciendo más de lo que necesita: no está haciendo "una y sólo una cosa".

## 3.5 ILUMINACIÓN Y OSCURECIMIENTO

Es bastante sencillo iluminar u oscurecer una imagen. Es el mismo patrón que vimos antes, pero en vez de cambiar un componente de color, hay que cambiar el color en general. A continuación le mostraremos programas para iluminar y oscurecer. La figura 3.18 muestra la versión original de una imagen y una más oscura.

**Programa 14: iluminar la imagen**

```
def iluminar(imagen):
  for px in getPixels(imagen):
    color = getColor(px)
    color = makeLighter(color)
    setColor(px,color)
```

■

**FIGURA 3.18**
La imagen original (*izquierda*) y una versión más oscura (*derecha*).

### Cómo funciona

La variable `px` se utiliza para representar cada uno de los píxeles en la imagen de entrada. (¡No es `p`! ¿Acaso importa? No para la computadora; si `p` significa "pixel" para usted, puede usarla, pero siéntase libre de usar `px` o `pxl`, o incluso `pixel`). La variable `color` toma el color del pixel `px`. La función `makeLighter` devuelve el nuevo color más claro. El método `setColor` establece el color del pixel al nuevo color más claro.

**Programa 15: oscurecer la imagen**

```
def oscurecer(imagen):
  for px in getPixels(imagen):
    color = getColor(px)
    color = makeDarker(color)
    setColor(px,color)
```

■

## 3.6 CREACIÓN DE UN NEGATIVO

Es mucho más fácil crear el *negativo* de una imagen de lo que podría pensar en primera instancia. Vamos a analizarlo con detenimiento. Lo que queremos es lo opuesto de cada uno de los valores actuales de rojo, verde y azul. Es más fácil de entender en los extremos. Si tenemos un componente rojo de 0, queremos 255. Si tenemos 255, queremos que el negativo tenga un 0.

Ahora consideremos la parte intermedia. Si el componente rojo es ligeramente rojo (por decir, 50), queremos algo que sea casi totalmente rojo; en donde el "casi" es la misma cantidad de rojo de la imagen original. Queremos el rojo máximo (255), pero 50 menos que eso. Queremos un componente rojo de 255 – 50 = 205. En general, el negativo debería ser 255 – *original*. Necesitamos calcular el negativo de cada componente rojo, verde y azul, para después crear un nuevo color negativo y establecer el pixel a ese color negativo.

He aquí el programa que lo hace; además puede ver que en realidad funciona (figura 3.19).

**FIGURA 3.19**
Negativo de la imagen.

**Programa 16: crear el negativo de la imagen original**

```
def negativo(imagen):
  for px in getPixels(imagen):
    rojo=getRed(px)
    verde=getGreen(px)
    azul=getBlue(px)
    colorNeg=makeColor(255-rojo, 255-verde, 255-azul)
    setColor(px,colorNeg)
```

■

### Cómo funciona

Usamos `px` para representar cada uno de los píxeles en la imagen de entrada. Para cada pixel `px`, usamos las variables `rojo`, `verde` y `azul` para nombrar los componentes rojo, verde y azul del color del pixel. Creamos un *nuevo* color con `makeColor` cuyo componente rojo es `255-rojo`, el verde es `255-verde` y el azul es `255-azul`. Esto significa que el nuevo color es el *opuesto* del color original. Por último, establecemos el color del pixel `px` al nuevo color negativo (`colorNeg`) y avanzamos al siguiente pixel.

## 3.7 CONVERSIÓN A ESCALA DE GRISES

La conversión a escala de grises es una receta divertida. Es corta, no es difícil de entender y aun así tiene un efecto visual agradable. Es en realidad un buen ejemplo de lo que podemos hacer con facilidad y a la vez poderoso, al manipular los valores de color de los píxeles.

Recuerde que el color resultante es gris cada vez que el componente rojo, el verde y el azul tienen el mismo valor. Esto significa que nuestra codificación RGB soporta 256 niveles de gris, desde (0, 0, 0) (negro) a (1, 1, 1) hasta (100, 100, 100) y por último (255, 255, 255) (blanco). El truco está en averiguar cuál debe ser el valor replicado.

Lo que queremos es una idea de la *intensidad* del color, lo cual se conoce como *luminancia*. Resulta ser que hay una forma bastante fácil de calcularla: promediamos los tres componentes de color. Como hay tres componentes, la fórmula que vamos a usar para la intensidad es:

$$\frac{(rojo + verde + azul)}{3}$$

Esto nos conduce al siguiente programa simple y a la figura 3.20.

**Programa 17: conversión a escala de grises**

```
def escalaGrises(imagen):
  for p in getPixels(imagen):
    intensidad = (getRed(p)+getGreen(p)+getBlue(p))/3
    setColor(p,makeColor(intensidad,intensidad,intensidad))
```

■

**FIGURA 3.20**
Imagen a colores convertida en escala de grises.

Ésta es en realidad una noción demasiado simple de escala de grises. A continuación veremos un programa que toma en cuenta la forma en que el ojo humano percibe la *luminancia*. Recuerde que nosotros consideramos que el azul es más oscuro que el rojo, incluso aunque se refleje la misma cantidad de luz. Por lo tanto, *asignamos un peso* menor al azul y mayor al rojo a la hora de calcular el promedio.

**Programa 18: conversión a escala de grises con pesos**

```
def escalaGrisesNuevo(imagen):
  for px in getPixels(imagen):
    nuevoRojo = getRed(px) * 0.299
    nuevoVerde = getGreen(px) * 0.587
    nuevoAzul = getBlue(px) * 0.114
    luminancia = nuevoRojo+nuevoVerde+nuevoAzul
    setColor(px,makeColor(luminancia,luminancia,luminancia))
```

■

## Cómo funciona

Hacemos que `px` represente a cada pixel en la imagen. Después *pesamos* el grado de rojo, de verde y de azul con base en lo que la investigación empírica muestra sobre cómo percibimos la luminancia de cada uno de estos colores. Observe que 0.299 + 0.587 + 0.114 es 1.0. Aun así vamos a terminar con un valor entre 0 y 255, pero vamos a obtener *más* del valor de la luminancia de la parte de verde, menos de la parte de rojo y aún menos de la parte de azul (que, como ya hemos demostrado, se percibe como el más oscuro). Después sumamos los tres valores pesados para obtener nuestra nueva luminancia. Creamos el color y establecemos el color del pixel `px` al nuevo color que hemos creado.

## RESUMEN DE PROGRAMACIÓN

En este capítulo vimos varios tipos de codificaciones de datos (u objetos).

| | |
|---|---|
| Imágenes | Codificaciones de imágenes, por lo general creadas a partir de un archivo JPEG. |
| Píxeles | Un arreglo unidimensional (secuencia) de objetos pixel. El código `pixeles[0]` devuelve el pixel de la esquina superior izquierda en una imagen. |
| Pixel | Un punto en la imagen. Tiene un color y una posición ($x$, $y$) asociados con él. Recuerda su propia imagen, de modo que un cambio en el pixel cambia el punto real en la imagen. |
| Color | Una mezcla de valores de rojo, verde y azul, cada uno entre 0 y 255. |

## PIEZAS DEL PROGRAMA DE IMÁGENES

| | |
|---|---|
| `getPixels` | Recibe una imagen como entrada y devuelve un arreglo unidimensional de objetos pixel en la imagen, con los píxeles de la primera fila primero, después los píxeles de la segunda fila y así en lo sucesivo. |
| `getPixel` | Recibe una imagen, una posición $x$ y una posición $y$ (dos números); luego devuelve el objeto pixel en ese punto en la imagen. |
| `getWidth` | Recibe una imagen como entrada y devuelve su anchura en el número de píxeles a lo largo de la imagen. |
| `getHeight` | Recibe una imagen como entrada y devuelve su longitud en el número de píxeles de arriba hacia abajo en la imagen. |
| `writePictureTo` | Recibe una imagen y un nombre de archivo (cadena) como entrada; después escribe la imagen al archivo en forma de JPEG. (Asegúrese de terminar el nombre de archivo en ".jpg" para que el sistema operativo pueda interpretarla correctamente). |

## PIEZAS DEL PROGRAMA DE PÍXELES

| | |
|---|---|
| `getRed, getGreen, getBlue` | Cada una de estas funciones recibe un objeto pixel y devuelve el valor (entre 0 y 255) del grado de rojo, verde y azul (respectivamente) en ese pixel. |
| `setRed, setGreen, setBlue` | Cada una de estas funciones recibe un objeto pixel y un valor (entre 0 y 255); luego establece el grado de rojo, verde o azul (respectivamente) de ese pixel con el valor dado. |
| `getColor` | Recibe un pixel y devuelve el objeto color en ese pixel. |
| `setColor` | Recibe un objeto pixel y un objeto color; después establece el color para ese pixel. |
| `getX, getY` | Recibe un objeto pixel y devuelve la posición $x$ o $y$ (respectivamente) de la ubicación del pixel en la imagen. |

## PIEZAS DEL PROGRAMA DE COLORES

| | |
|---|---|
| `makeColor` | Recibe tres entradas para los componentes rojo, verde y azul (en orden); devuelve un objeto color. |
| `pickAColor` | No recibe entrada pero muestra un selector de colores. Busque el color que desea y la función devolverá el color que usted haya seleccionado. |
| `makeDarker`, `makeLighter` | Cada una recibe un color y devuelve una versión un poco más oscura o clara (respectivamente) del color. |

Hay un grupo de **constantes** que son de utilidad en este capítulo. Éstas son variables con valores predefinidos. Estos valores son colores: `black`, `white`, `blue`, `red`, `green`, `gray`, `darkGray`, `lightGray`, `yellow`, `orange`, `pink`, `magenta` y `cyan`. Están definidas dentro de JES; en realidad usan código Python como éste:

```
#Constantes
black = Color(0,0,0)
white = Color(255,255,255)
blue = Color(0,0,255)
red = Color(255,0,0)
green = Color(0,255,0)
gray = Color(128,128,128)
darkGray = Color(64,64,64)
lightGray = Color(192,192,192)
yellow = Color(255,255,0)
orange = Color(255,200,0)
pink = Color(255,175,175)
magenta = Color(255,0,255)
cyan = Color(0,255,255)
```

## PROBLEMAS

3.1 Preguntas sobre el concepto de imagen:

- ¿Por qué no vemos puntos rojos, verdes y azules en cada posición en nuestra imagen?
- ¿Qué es la descomposición jerárquica? ¿Para qué sirve?
- ¿Qué es la luminancia?
- ¿Por qué el valor máximo de cualquier componente de color (rojo, verde o azul) es 255?
- La codificación de color que usamos es RGB. ¿Qué significa esto, en términos de la cantidad de memoria requerida para representar el color? ¿Hay un límite para el número de colores que podemos representar? ¿Hay *suficientes* colores que puedan representarse en RGB?

3.2 Sin duda, el programa 9 (página 56) reduce demasiado el color rojo. Escriba una versión que sólo reduzca el rojo en un 10% y luego otro en un 20%. ¿Puede encontrar imágenes en donde cada uno sea más útil? Tenga en cuenta que siempre podrá reducir repetidas veces el rojo en una imagen, pero no es conveniente tener que hacerlo *demasiadas* veces.

3.3 Escriba las versiones de azul y verde de la función de reducción de rojo: programa 9 (página 56).

3.4 Cada una de las siguientes funciones es equivalente a la función para aumentar el rojo: programa 10 (página 61). Pruébelas hasta quedar convencido de que funcionan. ¿Cuál prefiere y por qué?

```
def aumentarRojo2(imagen):
  for p in getPixels(imagen):
    setRed(p,getRed(p)*1.2)

def aumentarRojo3(imagen):
  for p in getPixels(imagen):
    componenteRojo = getRed(p)
    componenteVerde = getGreen(p)
    componenteAzul = getBlue(p)
    nuevoRojo=int(componenteRojo*1.2)
    nuevoColor = makeColor
        (nuevoRojo,componenteVerde,componenteAzul)
    setColor(p,nuevoColor)
```

3.5 Si continúa aumentando el valor rojo y está activada la función de regresar a partir de 255, en un momento dado ciertos píxeles se volverán de color verde y azul *brillante*. Si revisa esos píxeles con la herramienta de imágenes, encontrará que los valores de rojo son muy *bajos*. ¿Qué cree que está sucediendo? ¿Cómo se hicieron tan pequeños? ¿Cómo funciona la característica de regresar?

3.6 Escriba una función para intercambiar los valores de dos colores; por ejemplo, intercambiar el valor de rojo con el valor de azul.

3.7 Escriba una función para establecer los valores rojo, verde y azul a cero. ¿Cuál es el resultado?

3.8 Escriba una función para establecer los valores rojo, verde y azul a 255. ¿Cuál es el resultado?

3.9 ¿Qué hace la siguiente función?

```
def prueba1 (imagen):
    for p in getPixels(imagen):
        setRed(p,getRed(p) * 0.3)
```

3.10 ¿Qué hace la siguiente función?

```
def prueba2 (imagen):
    for p in getPixels(imagen):
        setBlue(p,getBlue(p) * 1.5)
```

3.11 ¿Qué hace la siguiente función?

```
def prueba3 (imagen):
    for p in getPixels(imagen):
        setGreen(p,0)
```

3.12 ¿Qué hace la siguiente función?

```
def prueba4 (imagen):
    for p in getPixels(imagen):
        rojo = getRed(p)
```

```
        verde = getGreen(p)
        azul = getBlue(p)
        color = makeColor
                (rojo + 10, verde + 10, azul + 10)
        setColor(p,color)
```

3.13 ¿Qué hace la siguiente función?

```
def prueba5 (imagen):
    for p in getPixels(imagen):
        rojo = getRed(p)
        verde = getGreen(p)
        azul = getBlue(p)
        color = makeColor
                (rojo - 20, verde - 20, azul - 20)
        setColor(p,color)
```

3.14 ¿Qué hace la siguiente función?

```
def prueba6 (imagen):
    for p in getPixels(imagen):
        rojo = getRed(p)
        verde = getGreen(p)
        azul = getBlue(p)
        color = makeColor (azul, rojo, verde)
        setColor(p,color)
```

3.15 ¿Qué hace la siguiente función?

```
def prueba7 (imagen):
    for p in getPixels(imagen):
        rojo = getRed(p)
        verde = getGreen(p)
        azul = getBlue(p)
        color = makeColor
                (rojo / 2, verde / 2, azul / 2)
        setColor(p,color)
```

3.16 Escriba una función para modificar una imagen a escala de grises y luego convertirla a su negativo.

3.17 Escriba tres funciones, una para borrar el azul (programa 11 [página 62]), otra para borrar el rojo y otra para borrar el verde. Para cada función, ¿cuál sería la más útil en la práctica real? ¿Qué tal las combinaciones de ellas?

3.18. Vuelva a escribir el programa para borrar el azul (programa 11 [página 62]) para *maximizar* el azul (es decir, establecerlo a 255) en vez de borrarlo. ¿Es esto útil? ¿Sería útil la versión para rojo o verde de la función de maximización? ¿Bajo qué condiciones?

## PARA PROFUNDIZAR

Un maravilloso libro sobre cómo funciona la vista y la manera en que los artistas han aprendido a manipularla es *Vision and Art: The Biology of Seeing*, escrito por Margaret Livingstone [28].

CAPÍTULO 4

# Modificación de píxeles en un rango

4.1 COPIA DE PÍXELES
4.2 REFLEJO DE UNA IMAGEN
4.3 COPIA Y TRANSFORMACIÓN DE IMÁGENES

### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Reflejar imágenes en sentido horizontal o vertical.
- Combinar imágenes para formar collages.
- Girar imágenes.
- Escalar imágenes para hacerlas más grandes o más pequeñas.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar ciclos anidados para direccionar los elementos de una matriz.
- Iterar sólo a través de una parte de un arreglo.
- Desarrollar algunas estrategias de depuración; en específico, usar instrucciones `print` para explorar el código en ejecución.

## 4.1 COPIA DE PÍXELES

No podemos llegar muy lejos en el procesamiento de imágenes con `getPixels` sin tener que saber *en dónde* se encuentra un pixel. Por ejemplo, si queremos reducir el rojo sólo en la mitad de una imagen, necesitamos una manera para especificar qué píxeles procesar. Para ello tendremos que empezar a crear nuestros propios ciclos `for` mediante `range`. Una vez que empecemos a hacer esto, tendremos más control sobre las coordenadas $x$, y $y$ que estamos procesando, y podremos empezar a mover píxeles a donde queramos. Ésta es una característica muy poderosa.

Pida a alguien que aplauda 10 veces. ¿Cómo sabe si lo hizo bien? Es probable que haya contado cada aplauso. Entonces la primera vez que aplaudió usted pensó en el 1, la segunda vez en el 2 y así, hasta que la persona dejó de aplaudir. Si el conteo fue de 10 cuando se detuvo, entonces quiere decir que aplaudió 10 veces. ¿En qué valor empezó el conteo? Si usted pensó en el 1 después de que la persona aplaudió una vez, entonces el conteo era cero antes del primer aplauso.

Un ciclo `for` es similar a esto. Hace que una variable índice reciba cada uno de los valores de una secuencia, uno a la vez. Ya hemos usado secuencias de píxeles generados a partir de `getPixels`, pero resulta ser que podemos generar con facilidad secuencias de números

usando la función `range` de Python. Esta función recibe dos entradas: un punto inicial entero y un punto final entero *que no se incluyen en la secuencia.*[1] Algunos ejemplos harán esto más claro.

```
>>> print range(0,3)
[0, 1, 2]
>>> print range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(3,1)
[]
```

Tal vez se pregunte qué significan estos corchetes (por ejemplo, `[0,3]` en el primer ejemplo anterior). Ésa es la notación para un arreglo: es la forma en que Python imprime una serie de números para mostrar que esto es un arreglo.[2] Si usamos `range` para generar el arreglo para el ciclo `for`, nuestra variable recorrerá cada uno de los números secuenciales que generamos.

De manera opcional, `range` puede recibir una tercera entrada: el incremento entre un elemento y otro de la secuencia.

```
>>> print range(0,10,3)
[0, 3, 6, 9]
>>> print range(0,10,2)
[0, 2, 4, 6, 8]
```

Puesto que la mayoría de los ciclos empiezan con cero (por ejemplo, al indexar algunos datos), al proporcionar *una sola* entrada a `range` se *presume* un punto inicial de cero.

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

### 4.1.1 Iterar a través de los píxeles mediante `range`

Si queremos conocer los valores $x$ y $y$ para un pixel, tendremos que usar *dos* ciclos `for`: uno para avanzar en sentido horizontal ($x$) a través de los píxeles y el otro para avanzar en sentido vertical ($y$) para obtener cada pixel (figura 4.1). La función `getPixels` hace esto en su interior, para facilitar la escritura de manipulaciones simples de imágenes. El ciclo interior estará *anidado* dentro del ciclo exterior, literalmente dentro de su bloque. En este punto va a tener que ser muy cuidadoso en cuanto a la aplicación de sangría en su código, para asegurarse de que sus bloques estén alineados en forma correcta.

Sus ciclos se verán así:

```
for x in range(0,getWidth(imagen)):
  for y in range(0,getHeight(imagen)):
    pixel=getPixel(imagen,x,y)
```

Por ejemplo, a continuación le mostramos el programa 14 (página 67), usando índices explícitos de $x$ y $y$ para acceder al pixel.

[1]Tal vez esto sea inesperado pero forma parte de la definición de Python, por lo que no es algo que sea posible cambiar si desea aprender cómo es que Python funciona en verdad.

[2]Técnicamente `range` devuelve una *secuencia*, que viene siendo una colección de datos un poco distinta a un arreglo. Para nuestros fines, podemos considerarla como un arreglo.

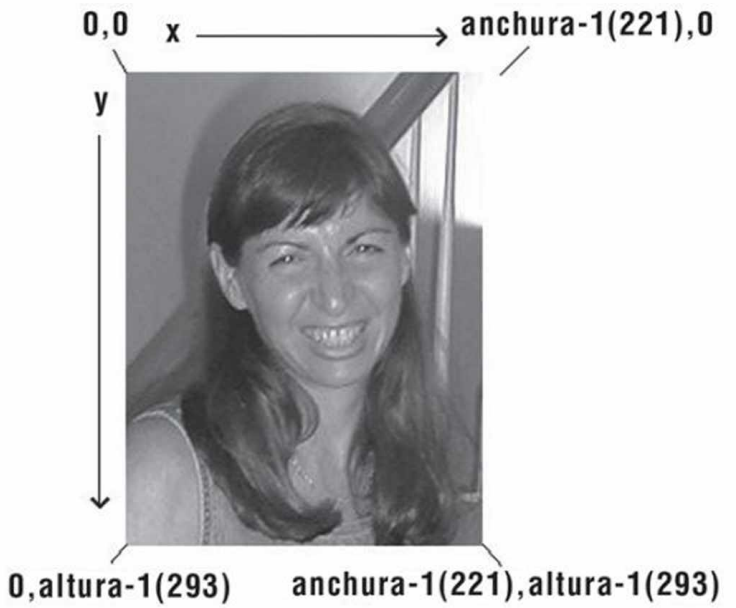



**FIGURA 4.1**
Las coordenadas de la imagen.

**Programa 19: iluminar la imagen usando ciclos anidados**

```
def iluminar(imagen):
  for x in range(0,getWidth(imagen)):
    for y in range(0,getHeight(imagen)):
      px = getPixel(imagen,x,y)
      color = getColor(px)
      color = makeLighter(color)
      setColor(px,color)
```

■

## Cómo funciona

Vamos a analizar (rastrear) la forma en que funcionaría este programa. Imagine que acabamos de ejecutar `iluminar(miImagen)`.

1. `def iluminar(imagen):` la variable `imagen` se convierte en el nuevo nombre de la imagen en `miImagen`.
2. `for x in range(0,getWidth(imagen)):` la variable x recibe el valor 0.
3. `for y in range(0,getHeight(imagen)):` la variable y recibe el valor 0.
4. `px = getPixel(imagen,x,y):` la variable `px` recibe el valor del objeto pixel en la ubicación (0, 0).
5. `setColor(px,color):` establecemos el pixel en (0, 0) para que sea el nuevo color más claro.
6. `for y in range(0,getHeight(imagen)):` ahora la variable `y` se convierte en 1. Acabamos de modificar el color en el pixel (0, 0) y ahora vamos a procesar el pixel en

(0, 1). En otras palabras, estamos avanzando con lentitud por la primera fila de píxeles, la columna en donde x es 0.

7. `px = getPixel(imagen,x,y)`: px se convierte en el pixel en la posición (0,1).
8. Hacemos ese color más claro.
9. `for y in range(0,getHeight(imagen))`: ahora la variable y se convierte en 2. Después procesaremos el pixel en (0, 2), luego (en el siguiente ciclo) (0, 3), (0, 4) y así en lo sucesivo, hasta que y se convierta en la altura de la imagen.
10. `for x in range(0,getWidth(imagen))`: la variable x recibe el valor 1.
11. Ahora y se convierte en 0 otra vez, y empezamos recorriendo la siguiente columna en donde x es 1. Procesamos el pixel en (1, 0), después (iterando sobre la variable y) (1, 1), (1, 2) y así en lo sucesivo, hasta que y sea igual a la altura de la imagen. Después aumentamos x y empezamos recorriendo la columna $x = 2$, procesando cada y para acceder a los píxeles (2, 0), (2, 1), (2, 2) y así en lo sucesivo.
12. Esto continúa hasta que se hacen más claros todos los colores de todos los píxeles.

## 4.2 REFLEJO DE UNA IMAGEN

Empecemos con un efecto interesante que sólo es útil algunas veces, pero divertido. Vamos a reflejar una imagen a lo largo de su eje vertical. En otras palabras, imagine que tiene un espejo y lo coloca sobre una imagen de modo que el lado izquierdo de ésta aparezca en el espejo. Ése es el efecto que vamos a implementar. Lo haremos en un par de formas distintas.

Primero pensemos en lo que vamos a hacer. Y simplifiquemos las cosas un poco. Pensemos en reflejar un arreglo bidimensional de números. Elegiremos un `puntoReflejo` horizontal: la mitad a lo largo de la imagen, `getWidth(imagen)/2`.

Cuando la anchura de la imagen sea par, copiaremos la mitad de la imagen de izquierda a derecha, como se muestra en la figura 4.2. La anchura de este arreglo es 2, por lo que `puntoReflejo` es `2 / 2 = 1`. Necesitamos copiar de `x=0, y=0` a `x=1, y=0` y de `x=0, y=1` a `x=1, y=1`. Observe que el `puntoReflejo` en realidad no es la *mitad* de la imagen, ya que la anchura de ésta es un número par. Piense en lo que ocurriría si la anchura de la imagen fuera 4. ¿Qué píxeles se copiarían aquí?

Cuando la anchura de la imagen sea impar, no copiaremos la columna intermedia de píxeles como se muestra en la figura 4.2. Necesitamos copiar de `x=0, y=0` a `x=2, y=0` y de `x=0, y=1` a `x=2, y=1`.

En cada uno de estos casos, empezamos x en 0 y y en 0, e iteramos con x variando desde 0 hasta el `puntoReflejo` y y variando desde 0 hasta la altura. El valor de x variará desde 0 hasta justo antes del `puntoReflejo`. Para reflejar una imagen, el color en la primera columna debería copiarse a la última columna en la misma fila. El color en la segunda columna debería copiarse a la antepenúltima columna en la misma fila, y así en lo sucesivo. Así, cuando x sea igual a 0, copiaremos el color del pixel en `x=0` y `y=0` a `x = anchura-1` y `y = 0`. Cuando x sea igual a 1, copiaremos el color del pixel en `x=1` y `y=0` a `x=anchura-2` y `y = 0`. Cuando x sea igual a 2, copiaremos el color del pixel en `x=2` y `y=0` a `x=anchura-3` y `y = 0`. Cada vez estamos copiando el color del pixel de la izquierda en los valores x y y actuales, a un pixel de la derecha a la anchura de la imagen menos el valor actual de x menos 1.

Dé un vistazo a la figura 4.2 para convencerse de que en verdad llegamos a cada pixel usando este esquema. He aquí el programa actual.

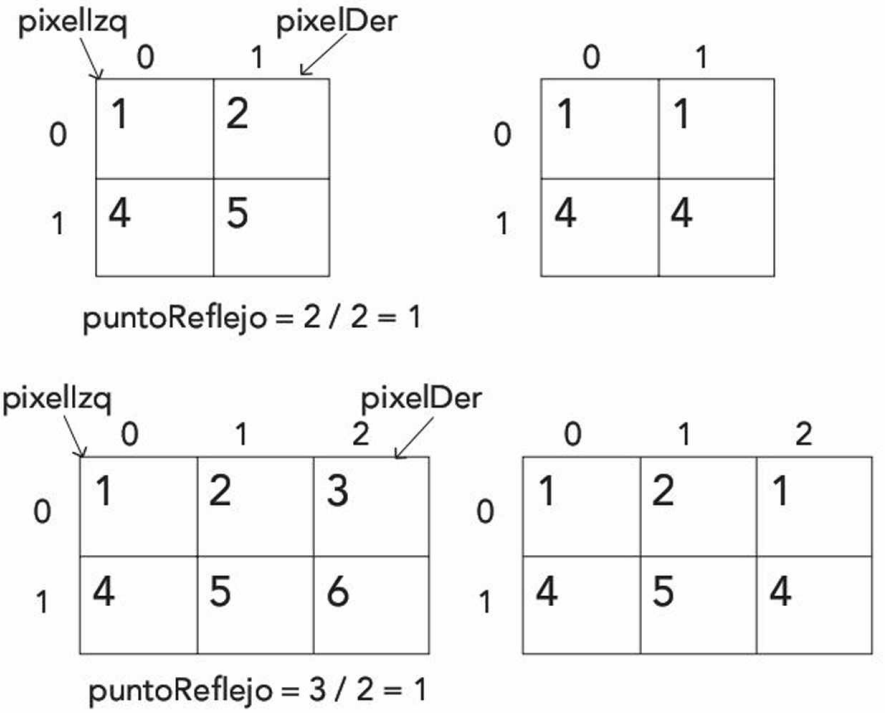



**FIGURA 4.2**
Copiar píxeles de izquierda a derecha alrededor de un punto de reflejo.

**Programa 20: reflejo de píxeles en una imagen a lo largo de una línea vertical**

```
def reflejoVertical(origen):
  puntoReflejo = getWidth(origen) / 2
  anchura = getWidth(origen)
  for y in range(0, getHeight(origen)):
    for x in range(0,puntoReflejo):
      pixelIzq = getPixel(origen,x,y)
      pixelDer = getPixel(origen,anchura - x - 1,y)
      color = getColor(pixelIzq)
      setColor(pixelDer,color)
```

■

## Cómo funciona

`reflejoVertical` recibe una imagen de origen como entrada. Usamos un reflejo vertical a la mitad a lo largo de la imagen, por lo que el `puntoEspejo` es la anchura de la imagen dividida entre 2. Procesamos la altura completa de la imagen, por lo que el ciclo para `y` va desde 0 hasta la altura de la imagen. El valor de `x` va desde 0 hasta `puntoReflejo` (sin incluir el `puntoReflejo`). Cada vez que iteramos por el ciclo, copiamos el color de otra columna de píxeles, de izquierda a derecha.

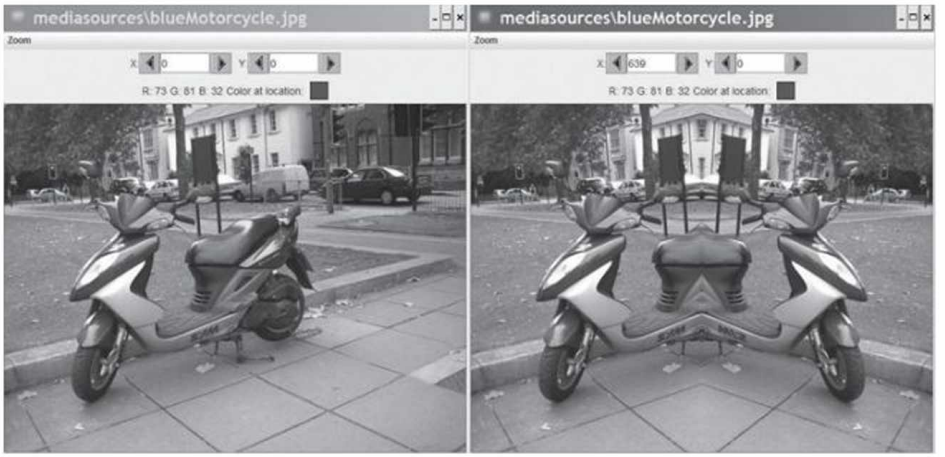



**FIGURA 4.3**
Imagen original (*izquierda*) e imagen reflejada a lo largo del eje vertical (*derecha*).

Usaríamos este programa de la siguiente forma (el resultado aparece en la figura 4.3).

```
>>> archivo="C:/ip-book/mediasources/blueMotorcycle.jpg"
>>> print archivo
C:/ip-book/mediasources/blueMotorcycle.jpg
>>> imagen=makePicture(archivo)
>>> explore(imagen)
>>> reflejoVertical(imagen)
>>> explore(imagen)
```

¿Podemos crear un reflejo horizontal? ¡Desde luego!

**Programa 21: reflejo de píxeles en sentido horizontal, de arriba hacia abajo**

```
def reflejoHorizontal(origen):
  puntoReflejo = getHeight(origen) / 2
  altura = getHeight(origen)
  for x in range(0,getWidth(origen)):
    for y in range(0,puntoReflejo):
      pixelSuperior = getPixel(origen,x,y)
      pixelInferior = getPixel(origen,x,altura - y - 1)
      color = getColor(pixelSuperior)
      setColor(pixelInferior,color)
```

■

Ahora, este último programa copia de la parte superior de la imagen hacia la parte inferior (vea la figura 4.4). Aquí puede ver que obtenemos el color de `pixelSuperior`, el cual proviene de los valores actuales de `x` y `y`, que siempre estarán *por encima* del `puntoReflejo`, ya que los valores más pequeños de y están más cerca de la parte superior de la imagen. Para copiar de la parte inferior hacia arriba, sólo hay que intercambiar `pixelSuperior` y `pixelInferior` (figura 4.4).

**FIGURA 4.4**
Reflejo horizontal, de arriba hacia abajo (*izquierda*) y de abajo hacia arriba (*derecha*).

**Programa 22: reflejo de píxeles en sentido horizontal, de abajo hacia arriba**

```
def reflejoAbajoArriba(origen):
  puntoReflejo = getHeight(origen) / 2
  altura = getHeight(origen)
  for x in range(0,getWidth(origen)):
    for y in range(0,puntoReflejo):
      pixelSuperior = getPixel(origen,x,y)
      pixelInferior = getPixel(origen,x,altura - y - 1)
      color = getColor(pixelInferior)
      setColor(pixelSuperior,color)
```

■

## Generar reflejos útiles

Aunque en su mayor parte los reflejos se usan para efectos interesantes, en algunas ocasiones tienen fines más serios (pero de todas formas divertidos). Mark tomó una fotografía del templo de Hefesto en la antigua ágora en Atenas, Grecia, cuando viajó a una conferencia (figura 4.5). De pura suerte tomó el pedimento del templo totalmente horizontal. El pedimento del templo de Hefesto está dañado. Mark se preguntó si podría "componerlo" al reflejar la parte buena hacia la parte rota.

Para averiguar las coordenadas en donde necesitamos generar el reflejo, use la herramienta de imágenes en JES para explorar la imagen.

```
>>> archivo = "C:/ip-book/mediasources/temple.jpg"
>>> temploP = makePicture(archivo)
>>> explore(temploP)
```

Mark exploró la imagen para descubrir el rango de valores necesarios para el reflejo y el punto alrededor del cual se debía generar (figura 4.6). La función que escribió para realizar la reparación se muestra a continuación y la imagen final se muestra en la figura 4.7; ¡funcionó bastante bien! Desde luego que es posible detectar que se manipuló en forma digital. Por ejemplo, si revisa las sombras podrá ver que el sol debe haber estado a la izquierda y a la derecha al mismo tiempo.

**FIGURA 4.5**
El templo de Hefesto de la antigua ágora en Atenas.

**FIGURA 4.6**
Coordenadas en donde necesitamos realizar el reflejo.

Vamos a usar la *función utilitaria* `getMediaPath(nombreBase)`. Es bastante útil cuando queremos lidiar con varias piezas de medios en el mismo directorio pero no queremos escribir el nombre completo. Sólo tiene que recordar que debe utilizar `setMediaPath()` primero. La función `setMediaPath()` abre un cuadro de diálogo selector de archivos, el cual le

**FIGURA 4.7**
El templo manipulado.

permite seleccionar una carpeta (o directorio: términos diferentes que significan lo mismo en el sistema de archivos) y luego imprime la *ruta* a la nueva carpeta que usted seleccionó. Todo lo que hace `getMediaPath` es anteponer la ruta encontrada en `setMediaPath` al nombre del archivo de entrada.

```
>>> setMediaPath()
'C:\\ip-book\\mediasources\\'
>>> getMediaPath("barbara.jpg")
'C:\\ip-book\\mediasources\\barbara.jpg'
>>> barb=makePicture(getMediaPath("barbara.jpg"))
```

En el ejemplo anterior cuando ejecutamos `setMediaPath()`, establecimos la carpeta de medios en **mediasources**, que se encuentra dentro del directorio **ip-book**, que a su vez se encuentra en la raíz (directorio principal) del disco duro C: (lo cual nos indica que este ejemplo se ejecutó en una computadora Windows). Si ejecutamos `setMediaPath()` en una computadora Mac OS X, podríamos ver algo como esto:

```
>>> setMediaPath()
'/Users/guzdial/Desktop/MediaComp/mediasources/'
```

Lo mismo ocurre en este ejemplo. Aparece un cuadro de diálogo selector de archivos, se selecciona un directorio y luego se imprime. En una Mac OS X, a la raíz del disco duro principal se le conoce simplemente como "/" y la ruta anterior sugiere que la carpeta **Users** se encuentra en el directorio del disco principal; además hay un directorio en su interior llamado **guzdial**, seguido de **Desktop** y luego de **MediaComp**. Por último, los medios se almacenan en la carpeta **mediasources** dentro de **MediaComp**.

**Programa 23: reflejo del templo de Hefestos**

```
def reflejoTemplo():
  origen = makePicture(getMediaPath("temple.jpg"))
  puntoReflejo = 276
  for x in range(13, puntoReflejo):
    for y in range(27,97):
      pizq = getPixel(origen,x,y)
      pder = getPixel(origen,puntoReflejo + puntoReflejo - 1 - x,y)
      setColor(pder,getColor(pizq))
  show(origen)
  return origen
```

■

**Error común: ¡establezca primero la carpeta de medios!**

Si va a usar código que acceda a `getMediaPath(nombreBase)`, tendrá que ejecutar `setMediaPath()` primero.

■

## Cómo funciona

Para encontrar el `puntoReflejo` (276) exploramos la imagen en JES. En realidad no tenemos que copiar de 0 a 276, ya que el borde del templo está en un índice `x` de 13.

Este programa es también uno de los primeros que hemos escrito en donde devolvemos de manera explícita un valor mediante `return`. La palabra clave `return` establece el valor que la función provee como salida. En `reflejoTemplo()`, el valor de retorno es el objeto imagen `origen` en donde se almacena el templo reparado. Si invocáramos esta función con `temploCorregido = reflejoTemplo()`, el nombre `temploCorregido` representaría la imagen devuelta de `reflejoTemplo()`.

**Error común: la palabra clave `return` siempre va al último**

La palabra clave `return` especifica cuál es el valor de retorno de la función, pero también tiene el efecto de *terminar* la función. Un error común es tratar de usar `print` o show *después* de una instrucción `return`, pero eso no funcionará. Una vez que se ejecute `return`, no se ejecutarán más instrucciones en la función.

■

¿Por qué devolvemos el objeto imagen del templo corregido? ¿Por qué *nunca habíamos* devuelto nada antes? Las funciones que escribimos antes de ésta manipulan de manera directa los objetos de entrada: a esto se le conoce como cálculo mediante *efecto secundario*. No podemos hacerlo en este caso debido a que no hay entradas para `reflejoTemplo()`. La regla general sobre cuándo usar `return` es esta: si *crea* el objeto de interés dentro de la función, tendrá que devolverlo mediante `return`, o de lo contrario el objeto simplemente desaparecerá cuando termine la función. Como el objeto imagen `origen` se crea (mediante `makePicture`) dentro de la función `reflejoTemplo()`, el objeto sólo existe dentro de la función.

¿De qué sirve regresar algo mediante `return`? Lo hacemos para un uso futuro. ¿Puede imaginar la *posibilidad* de querer hacer algo con el templo reflejado? ¿Tal vez integrarlo en

un collage, o cambiar su color? Debe devolver el objeto mediante `return` para tener esa opción disponible más adelante.

Vale la pena preguntarnos sobre el ejemplo del templo. Si en realidad lo entiende, podrá responder a preguntas como: "¿Cuál es el *primer* pixel que se va a reflejar en esta función?" y "¿Cuántos píxeles se copian en sí?" Debería ser capaz de averiguar las respuestas analizando el programa: pretenda que usted es la computadora y ejecute el programa en su mente.

Si eso es demasiado difícil, puede insertar instrucciones `print` como en el siguiente ejemplo:

```
def reflejoTemplo():
  origen = makePicture(getMediaPath("temple.jpg"))
  puntoReflejo = 276
  for x in range(13,puntoReflejo):
    for y in range(27,97):
      print "Copiando color de",x,y,"a",puntoReflejo+puntoReflejo-1-x,y
      pizq = getPixel(origen,x,y)
      pder = getPixel(origen,puntoReflejo+puntoReflejo-1-x,y)
      setColor(pder,getColor(pizq))
  show(origen)
  return origen
```

Al ejecutar esta versión, tarda *mucho* tiempo en terminar. Haga clic en el botón STOP después de un rato, puesto que lo único que nos importa son los primeros píxeles. He aquí lo que obtuvimos:

```
>>> p2=reflejoTemplo()
Copiando color de 13 27 a 538 27
Copiando color de 13 28 a 538 28
Copiando color de 13 29 a 538 29
```

Copia de (13, 27) a (538, 27), en donde el 538 se calcula a partir del `puntoReflejo` (276) más el `puntoReflejo` (522) menos 1 (551) y después menos `x` (551 – 13 = 538).

¿Cuántos píxeles procesamos? Podemos hacer que la computadora averigüe eso también. Antes de los ciclos, decimos que nuestro conteo es 0. Cada vez que copiamos un pixel, sumamos 1 a nuestro conteo.

```
def reflejoTemplo():
  origen = makePicture(getMediaPath("temple.jpg"))
  puntoReflejo = 276
  conteo = 0
  for x in range(13,puntoReflejo):
    for y in range(27,97):
      pizq = getPixel(origen,x,y)
      pder = getPixel(origen,puntoReflejo + puntoReflejo - 1 - x,y)
      setColor(pder,getColor(pizq))
      conteo = conteo + 1
  show(origen)
  print "Copiamos",conteo,"píxeles"
  return origen
```

Este programa devuelve `Copiamos 18410 píxeles`. ¿De dónde proviene este número? Copiamos 70 filas de píxeles (y va de 27 a 97). Copiamos 263 columnas de píxeles (x va de 13 a 276). 70 × 263 es 18410.

## 4.3 COPIA Y TRANSFORMACIÓN DE IMÁGENES

Podemos crear imágenes totalmente nuevas cuando copiamos píxeles *a través* de imágenes. Vamos a terminar llevando el registro de una imagen de *origen* de la que tomamos los píxeles, y una imagen *destino* en la que vamos a poner los píxeles. En realidad no copiamos los píxeles; sólo hacemos que los píxeles en la imagen de destino sean del mismo color que los píxeles en la imagen de origen. Para copiar píxeles tenemos que llevar el registro de múltiples variables índice: la posición $(x, y)$ en el origen y la posición $(x, y)$ en el destino.

Lo emocionante sobre copiar píxeles es que el hecho de realizar ciertos pequeños cambios en la forma en que lidiamos con las variables índice nos conduce no sólo a *copiar* la imagen, sino también a *transformarla*. En esta sección vamos a hablar sobre cómo copiar, recortar, girar y escalar imágenes.

Nuestro destino será el archivo JPEG del tamaño de un papel en el directorio **mediasources**; es de $7 \times 9.5$ pulgadas, lo cual cabe en una pieza de papel tamaño carta de $9 \times 11.5$ pulgadas con márgenes de 1 pulgada.

```
>>> archivoPapel=getMediaPath("7inx95in.jpg")
>>> imagenPapel=makePicture(archivoPapel)
>>> print getWidth(imagenPapel)
504
>>> print getHeight(imagenPapel)
684
```

### 4.3.1 Copiado

Para copiar una imagen de un objeto a otro, tan sólo debemos asegurarnos de incrementar las variables `origenX` y `destinoX` (las variables índice de origen y de destino para el eje *X*) al mismo tiempo, y las variables `origenY` y `destinoY` al mismo tiempo. Podríamos usar un ciclo `for`, pero sólo incrementa *una* variable. Tenemos que asegurarnos de incrementar la *otra* variable (la que *no* esté en el ciclo `for`) mediante el uso de una expresión al mismo tiempo (lo más cerca que podamos) que cuando el ciclo `for` realiza los incrementos. Para ello debemos establecer los valores iniciales *justo antes* de que empiece el ciclo, y después sumar uno a la variable índice en la parte *inferior* del ciclo.

He aquí un programa para copiar la imagen de Bárbara al lienzo.

**Programa 24: copiar una imagen a un lienzo**

```
def copiaBarb():
  # Establecer las imágenes de origen y destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  destinoX = 0
  for origenX in range(0,getWidth(barb)):
    destinoY = 0
    for origenY in range(0,getHeight(barb)):
      color = getColor(getPixel(barb,origenX,origenY))
```

```
    setColor(getPixel(lienzo,destinoX,destinoY), color)
    destinoY = destinoY + 1
  destinoX = destinoX + 1
show(barb)
show(lienzo)
return lienzo
```

■

**Idea de ciencias computacionales: ¡los comentarios son buenos!**

En el programa 24 podemos ver el uso de líneas con "#" al principio. Este símbolo indica a Python lo siguiente: "Ignora el resto de esta línea". ¿De qué nos sirve eso? Nos permite poner mensajes en el programa que los humanos (y no la computadora) puedan leer: mensajes que expliquen cómo funcionan las cosas, qué hacen las secciones del programa y por qué hizo lo que hizo. Recuerde que los programas son para humanos, no para computadoras. Los comentarios hacen que los programas se adapten mejor a los humanos.

■

## Cómo funciona

Este programa copia la imagen de Bárbara al lienzo (figura 4.8).

- Las primeras líneas sólo establecen las imágenes de origen (`barb`) y de destino (`lienzo`).
- A continuación viene el ciclo para manejar las variables índice *X*, `origenX` para la imagen de origen y `destinoX` para la imagen de destino. He aquí las partes clave del ciclo:

```
destinoX = 0
for origenX in range(0,getWidth(barb)):
  # AQUÍ VA EL CICLO PARA Y
  destinoX = destinoX + 1
```

**FIGURA 4.8**
Copiar una imagen a un lienzo.

Debido a la forma en que están dispuestas estas instrucciones, desde el punto de vista del ciclo *Y*, `destinoX` y `origenX` siempre se incrementan juntas. `destinoX` se convierte en 0 justo antes de que `origenX` se convierta en 0 en el ciclo `for`. Al final del ciclo `for`, `destinoX` se incrementa en 1, después el ciclo vuelve a empezar y `origenX` también se incrementa en 1 a través de la instrucción `for`.

La instrucción para incrementar `destinoX` puede parecer un poco extraña: `destinoX = destinoX + 1` no está realizando una declaración matemática (que no podría ser verdad), sino que proporciona indicaciones a la computadora. Dice: "Hacer el valor de `destinoX` igual a (lado derecho de =) cualquiera que sea el valor *actual* de `destinoX` más 1".

- Dentro del ciclo para las variables *X* está el ciclo para las variables *Y*. Tiene una estructura muy similar, ya que su objetivo es mantener a `destinoY` y `origenY` en sincronía y exactamente de la misma forma.

```
destinoY = 0
for origenY in range(0,getHeight(barb)):
  color = getColor(getPixel(barb,origenX,origenY))
  setColor(getPixel(lienzo,destinoX,destinoY), color)
  destinoY = destinoY + 1
return lienzo
```

Dentro del ciclo *Y* es de donde realmente se obtiene el color del origen y este mismo color se establece en el pixel correspondiente en el destino.

Resulta que podemos fácilmente poner las variables destino en el ciclo `for` y establecer las variables de origen. El programa que se muestra a continuación hace lo mismo que el programa 24.

**Programa 25: copiar una imagen a un lienzo de una manera diferente**

```
def copiaBarb2():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  origenX = 0
  for destinoX in range(0,getWidth(barb)):
    origenY = 0
    for destinoY in range(0,getHeight(barb)):
      color = getColor(getPixel(barb,origenX,origenY))
      setColor(getPixel(lienzo,destinoX,destinoY), color)
      origenY = origenY + 1
    origenX = origenX + 1
  show(barb)
  show(lienzo)
  return lienzo
```

■

Desde luego que no estamos obligados a copiar desde (0, 0) en el origen hasta (0, 0) en el destino. Es fácil copiar en cualquier otra parte del lienzo. Todo lo que tenemos que hacer

**FIGURA 4.9**
Copiar una imagen a la mitad de un lienzo.

es cambiar el *inicio* de las coordenadas $X$ y $Y$ de destino. El resto queda exactamente igual (figura 4.9).

**Programa 26: copiar en cualquier otra parte del lienzo**

```
def copiarBarbEnMedio():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  destinoX = 100
  for origenX in range(0,getWidth(barb)):
    destinoY = 100
    for origenY in range(0,getHeight(barb)):
      color = getColor(getPixel(barb,origenX,origenY))
      setColor(getPixel(lienzo,destinoX,destinoY), color)
      destinoY = destinoY + 1
    destinoX = destinoX + 1
  show(barb)
  show(lienzo)
  return lienzo
```

■

De igual forma, tampoco tenemos que copiar *toda* la imagen. El *recorte* sólo toma una parte de toda una imagen. En el ámbito digital, sólo es cuestión de modificar las coordenadas iniciales y finales. Para capturar sólo el rostro de Bárbara de la imagen, sólo tenemos que averiguar cuáles son las coordenadas en dónde se encuentra su rostro y después usarlas en las dimensiones de `origenX` y `origenY` (figura 4.10). Para ello hay que explorar la imagen. El rostro está en las coordenadas (42, 25) a (200, 200).

**FIGURA 4.10**
Copiar parte de una imagen a un lienzo.

**Programa 27: recortar una imagen y colocarla en un lienzo**

```
def copiarRostroBarb():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  destinoX = 100
  for origenX in range(45,200):
    destinoY = 100
    for origenY in range(25,200):
      color = getColor(getPixel(barb,origenX,origenY))
      setColor(getPixel(lienzo,destinoX,destinoY), color)
      destinoY = destinoY + 1
    destinoX = destinoX + 1
  show(barb)
  show(lienzo)
  return lienzo
```

■

## Cómo funciona

La única diferencia entre este programa y los anteriores está en los rangos en los índices de `origen`. Sólo queremos los píxeles `x` entre 45 y 200, por lo que éstos componen la entrada

de `range` para `origenX`. Como sólo queremos los píxeles y entre 25 y 200, también componen la entrada para `range` que usaremos para `origenY`. El resto queda justo igual.

Aún podemos intercambiar qué variables están en el ciclo `for` y cuáles se van a incrementar. Sin embargo, el cálculo del rango para el destino es un poco complicado. Si queremos empezar a copiar a (100, 100), entonces la longitud de la imagen es 200 – 45 y la altura es 200 – 25. He aquí el programa.

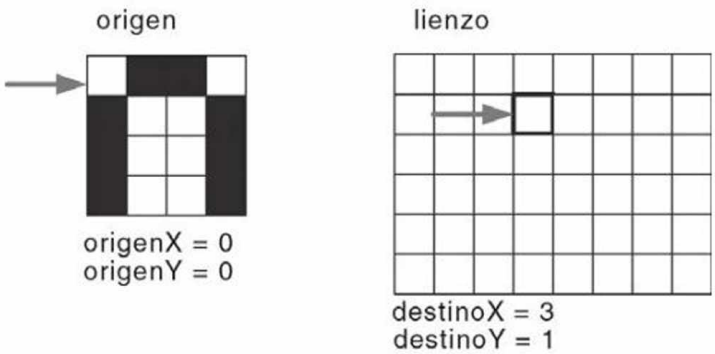
**Programa 28: recortar el rostro y colocarlo en el lienzo de manera diferente**

```
def copiarRostroBarb2():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  origenX = 45
  for destinoX in range(100,100+(200-45)):
    origenY = 25
    for destinoY in range(100,100+(200-25)):
      color = getColor(getPixel(barb,origenX,origenY))
      setColor(getPixel(lienzo,destinoX,destinoY), color)
      origenY = origenY + 1
    origenX = origenX + 1
  show(barb)
  show(lienzo)
  return lienzo
```

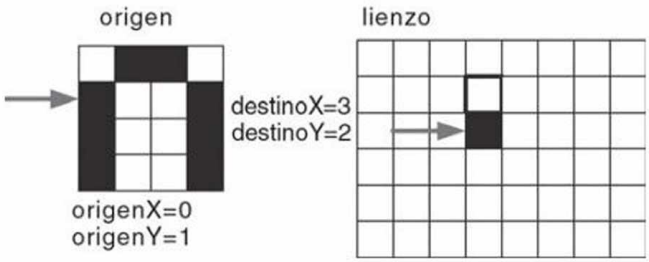
■

## Cómo funciona

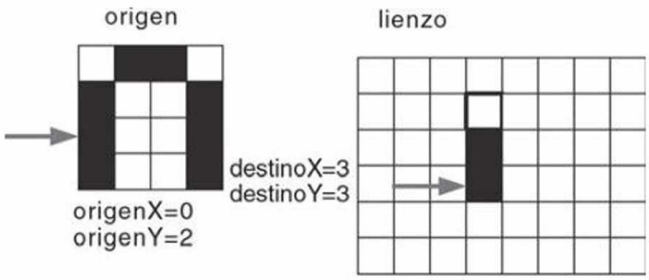
Analicemos un pequeño ejemplo para ver lo que ocurre en el programa de copiado. Empezamos con un origen y un destino, y copiamos del origen hacia el destino, pixel por pixel.



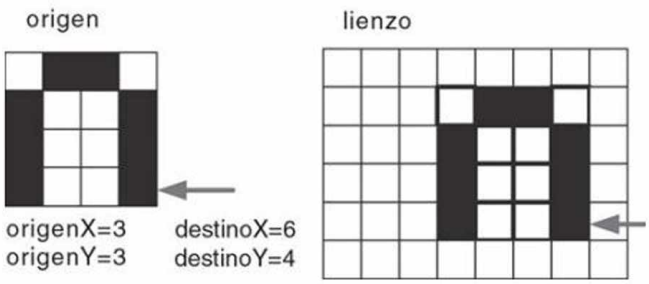
Después incrementamos `origenY` y `destinoY`, y volvemos a copiar.




Seguimos recorriendo la columna en orden descendente, incrementando ambas variables de índice *Y*.




Cuando terminamos con esa columna, incrementamos las variables índice *X* y avanzamos a la siguiente columna, hasta copiar todos los píxeles.




## Copiar y referenciar

**Idea de ciencias computacionales: comparación entre copia y referencia**

Cuando copiamos colores de la imagen de origen a la imagen de destino, la imagen de destino contiene sólo la información de los colores. No sabe nada sobre la imagen de origen. Si cambiamos la imagen de origen, el destino no cambiará. En ciencias computacionales también podemos crear *referencias*. Una referencia es un apuntador a otro objeto. Si la imagen de destino contiene una referencia a la imagen de origen y modificamos esta última, el destino también cambiaría.

■

Ésta es una idea compleja en las ciencias computacionales, por lo que vamos a experimentar un poco para explicarla mejor. Cuando usamos variables simples (por ejemplo, las que sólo contienen números), la asignación *copia* los valores de un lugar a otro.

```
>>> a = 100
>>> b = a
>>> print a,b
100 100
>>> b = 200
>>> print a,b
100 200
>>> a = 300
print a,b
300 200
```

En este ejemplo copiamos el valor 100 a la variable `a` cuando decimos `a = 100`. Cuando asignamos `b = a`, estamos copiando el valor de `a` (que es 100) en `b`. Ahora ambas variables tienen el valor 100. Cuando cambiamos `b` al copiar el valor 200 en ella, no cambiamos `a`. ¿Cómo podríamos? Las variables `a` y `b` sólo contienen valores que se copiaron en ellas. Por lo tanto, `a` sigue siendo 100 y ahora `b` es 200. Cuando copiamos 300 en `a`, no cambiamos `b`.

Las cosas son un poco distintas cuando asignamos a *objetos* complejos imágenes y píxeles. Asignar un pixel a otro objeto es crear una *referencia*. Si se cambia uno, cambia el otro.

```
>>> imagenBarb = makePicture(getMediaPath("barbara.jpg"))
>>> pixel1 = getPixelAt(imagenBarb,0,0)
>>> print pixel1
Pixel red=168 green=131 blue=105
>>> otropixel = pixel1
>>> print otropixel
Pixel red=168 green=131 blue=105
>>> setColor(pixel1,black)
>>> print pixel1
Pixel red=0 green=0 blue=0
>>> print otropixel
Pixel red=0 green=0 blue=0
```

En este ejemplo creamos una imagen y asignamos `pixel1` para representar el primer pixel en la imagen, el que está en (0, 0). Tiene el color rojo = 168, verde = 131, azul = 105. Cuando asignamos `otropixel` a `pixel1`, éstos se convierten en dos nombres distintos para el *mismo* objeto. Cuando imprimimos `otropixel`, tiene el mismo color que `pixel1`. Después establecemos el color del pixel de `pixel1` a negro mediante `black`. Cuando lo imprimimos, los tres componentes de color son 0. Y cuando imprimimos `otropixel`, *también* sus componentes son 0. Sólo hubo un pixel involucrado y al cambiarlo a través de cualquier nombre se produjo el mismo cambio en ambos nombres. Ésa es la diferencia entre una referencia y una copia.

Si consideramos esto desde la perspectiva de lo que hace la computadora, ambos ejemplos están usando copias. En el primer ejemplo, se están copiando valores. En el segundo ejemplo, lo que se copia es una *referencia* al objeto pixel. Considere que *getPixelAt(imagenBarb,0,0)* devuelve la dirección de donde se encuentra el primer pixel en la imagen `imagenBarb`. Primero, `pixel1` obtiene una copia de esa dirección. Cuando asignamos `otropixel = pixel1`, *copiamos* la dirección en `otropixel`. La llamada a la función `setColor(pixel1,black)`

está diciendo: "Cambia el pixel *en esta dirección (pixel1)* a negro". Es la misma dirección que `otropixel`, por lo que ambas hacen referencia al mismo pixel, al que acabamos de asignar el color negro.

### 4.3.2 Creación de un collage

He aquí un par de imágenes de flores (figura 4.11), cada una de 100 píxeles de anchura y de altura. Vamos a formar un *collage* con ellas, combinando varios de nuestros efectos para crear diferentes flores. Las copiaremos todas a la imagen en blanco **`640x480.jpg`**. Todo lo que tenemos que hacer en realidad es copiar los colores de los píxeles en los lugares correctos.

He aquí cómo creamos el collage (figura 4.12):

```
>>> flores=crearCollage()
```

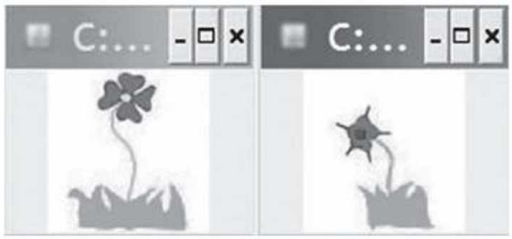



**FIGURA 4.11**
Flores a usar en el collage.

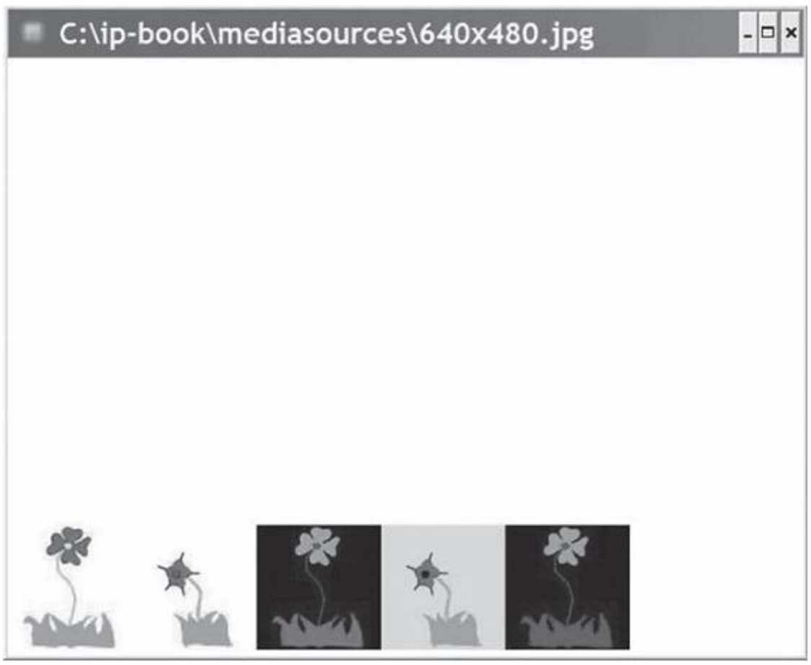



**FIGURA 4.12**
Collage de flores.

**Error común: ¡las funciones referenciadas deben estar en el archivo!**

Este programa usa funciones que escribimos antes. Hay que copiar estas funciones al mismo archivo en el que está la función `crearCollage` para que esto funcione (más adelante veremos una mejor forma de hacer esto con la instrucción `import`).

■

**Programa 29: creación de un collage**

```
def crearCollage():
  flor1=makePicture(getMediaPath("flower1.jpg"))
  print flor1
  flor2=makePicture(getMediaPath("flower2.jpg"))
  print flor2
  lienzo=makePicture(getMediaPath("640x480.jpg"))
  print lienzo
  #Primera imagen, en el borde izquierdo
  destinoX=0
  for origenX in range(0,getWidth(flor1)):
    destinoY=getHeight(lienzo)-getHeight(flor1)-5
    for origenY in range(0,getHeight(flor1)):
      px=getPixel(flor1,origenX,origenY)
      cx=getPixel(lienzo,destinoX,destinoY)
      setColor(cx,getColor(px))
      destinoY=destinoY + 1
    destinoX=destinoX + 1
  #Segunda imagen, a 100 píxeles de distancia
  destinoX=100
  for origenX in range(0,getWidth(flor2)):
    destinoY=getHeight(lienzo)-getHeight(flor2)-5
    for origenY in range(0,getHeight(flor2)):
      px=getPixel(flor2,origenX,origenY)
      cx=getPixel(lienzo,destinoX,destinoY)
      setColor(cx,getColor(px))
      destinoY=destinoY + 1
    destinoX=destinoX + 1
  #Tercera imagen, flor1 negada
  negativo(flor1)
  destinoX=200
  for origenX in range(0,getWidth(flor1)):
    destinoY=getHeight(lienzo)-getHeight(flor1)-5
    for origenY in range(0,getHeight(flor1)):
      px=getPixel(flor1,origenX,origenY)
      cx=getPixel(lienzo,destinoX,destinoY)
      setColor(cx,getColor(px))
      destinoY=destinoY + 1
    destinoX=destinoX + 1
  #Cuarta imagen, flor2 sin azul
  borrarAzul(flor2)
  destinoX=300
  for origenX in range(0,getWidth(flor2)):
    destinoY=getHeight(lienzo)-getHeight(flor2)-5
    for origenY in range(0,getHeight(flor2)):
```

```
    px=getPixel(flor2,origenX,origenY)
    cx=getPixel(lienzo,destinoX,destinoY)
    setColor(cx,getColor(px))
    destinoY=destinoY + 1
  destinoX=destinoX + 1
#Quinta imagen, flor1, negada con rojo reducido
reducirRojo(flor1)
destinoX=400
for origenX in range(0,getWidth(flor1)):
  destinoY=getHeight(lienzo)-getHeight(flor1)-5
  for origenY in range(0,getHeight(flor1)):
    px=getPixel(flor1,origenX,origenY)
    cx=getPixel(lienzo,destinoX,destinoY)
    setColor(cx,getColor(px))
    destinoY=destinoY + 1
  destinoX=destinoX + 1
show(lienzo)
return(lienzo)
```

■

### Cómo funciona

Aunque este programa se ve extenso, en realidad es el mismo ciclo de copiado que hemos visto repetidas veces, sólo que un ciclo después del otro.

- Primero creamos los objetos imagen `flor1`, `flor2` y `lienzo`. Vamos a copiar `flor1` y `flor2` al `lienzo`.
- La primera flor es tan sólo una simple copia de `flor1` en el borde de más a la izquierda del lienzo. Queremos que la parte inferior de la flor esté a cinco píxeles de distancia del borde, por lo que `destinoY` empieza a la altura del lienzo menos la altura de la flor, menos 5. A medida que se incrementa `destinoY` (suma), crecerá para *abajo* hacia la parte inferior. Se incrementará por el número de píxeles en la altura de la flor (vea el ciclo de `origenY`), por lo que el valor máximo que tomará `destinoY` será la altura del lienzo menos 5.
- A continuación copiamos la segunda imagen, en donde `destinoX` empieza 100 píxeles a la derecha, pero en realidad usa los mismos ciclos.
- Ahora negamos `flor1` y después la copiamos, avanzando más hacia la derecha (`destinoX` empieza ahora en 200).
- Luego borramos el azul de `flor2` y la copiamos en el lienzo todavía más hacia la derecha.
- La quinta flor reduce el rojo de `flor1`, *que ya está negada* (por el tercer conjunto de ciclos).
- Luego mostramos el lienzo mediante `show` y lo devolvemos con `return`. Necesitamos devolver el lienzo debido a que lo creamos dentro de la función de collage. Si no lo regresamos, tan sólo desparecerá cuando termine la función.

## 4.3.3 Copiado general

El código para crear el collage es muy extenso y repetitivo. Cada vez que copiamos una imagen en el destino, calculamos el valor de `destinoY` y establecemos el `destinoX`. Lue-

go iteramos a través de los píxeles en la imagen de origen y los copiamos todos al destino. ¿Acaso existe alguna manera de hacer esto más corto? ¿Qué tal si creamos una función de copia general que reciba tanto una imagen a copiar como el destino y especifique en dónde iniciar la copia en el destino?

**Programa 30: una función de copia general**

```
def copiar(origen, destino, destX, destY):
  destinoX = destX
  for origenX in range(0,getWidth(origen)):
    destinoY = destY
    for origenY in range(0,getHeight(origen)):
      px=getPixel(origen,origenX,origenY)
      tx=getPixel(destino,destinoX,destinoY)
      setColor(tx,getColor(px))
      destinoY=destinoY + 1
    destinoX=destinoX + 1
```

■

Ahora podemos reescribir la función de collage para usar esta nueva función de copia general.

**Programa 31: collage mejorado con la función de copia general**

```
def crearCollage2():
  flor1=makePicture(getMediaPath("flower1.jpg"))
  print flor1
  flor2=makePicture(getMediaPath("flower2.jpg"))
  print flor2
  lienzo=makePicture(getMediaPath("640x480.jpg"))
  print lienzo
  #Primera imagen, en el borde izquierdo
  copiar(flor1,lienzo,0,getHeight(lienzo)-getHeight(flor1)-5)
  #Segunda imagen, a 100 píxeles de distancia
  copiar(flor2,lienzo,100,getHeight(lienzo)-getHeight(flor2)-5)
  #Tercera imagen, flor1 negada
  negativo(flor1)
  copiar(flor1,lienzo,200,getHeight(lienzo)-getHeight(flor1)-5)
  #Cuarta imagen, flor2 sin azul
  borrarAzul(flor2)
  copiar(flor2,lienzo,300,getHeight(lienzo)-getHeight(flor2)-5)
  #Quinta imagen, flor1, negada con rojo reducido
  reducirRojo(flor1)
  copiar(flor1,lienzo,400,getHeight(lienzo)-getHeight(flor2)-5)
  return lienzo
```

■

Ahora el código para crear el collage es mucho más fácil de leer, modificar y comprender. Escribir funciones de modo que reciban parámetros las hace más fáciles de reutilizar. El hecho de repetir código varias veces en una función también puede ser un problema si el código repetido contiene un error. Tendrá que corregir el error en varios lugares, en vez de sólo uno.

**Tip de funcionamiento: reutilice las funciones en vez de copiarlas**

Trate de escribir funciones que puedan reutilizarse especificando parámetros. Trate de evitar copiar código en varios lugares, ya que éste se hará más extenso y los errores serán más difíciles de corregir.

■

### 4.3.4 Rotación

La imagen puede transformarse usando las variables índice de manera distinta, o incrementándolas en forma distinta pero conservando el mismo algoritmo de copiado. Ahora vamos a girar la imagen de Bárbara 90 grados a la izquierda: por lo menos eso parecerá. Lo que haremos en realidad será voltear la imagen a lo largo de la diagonal. Para ello, sólo tenemos que intercambiar las variables *X* y *Y* en el destino: las incrementaremos de la misma forma exacta, sólo que *usaremos X* para *Y* y *Y* para *X* (figura 4.13).

**FIGURA 4.13**
Copiar una imagen a un lienzo.

**Programa 32: girar (voltear) una imagen**

```
def copiarBarbDeLado():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  destinoX = 0
```

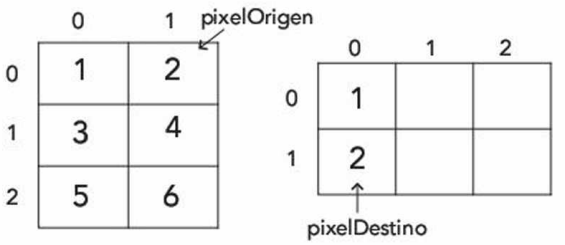
```
for origenX in range(0,getWidth(barb)):
  destinoY = 0
  for origenY in range(0,getHeight(barb)):
    color = getColor(getPixel(barb,origenX,origenY))
    setColor(getPixel(lienzo,destinoY,destinoX), color)
    destinoY = destinoY + 1
  destinoX = destinoX + 1
show(barb)
show(lienzo)
return lienzo
```

■

## Cómo funciona

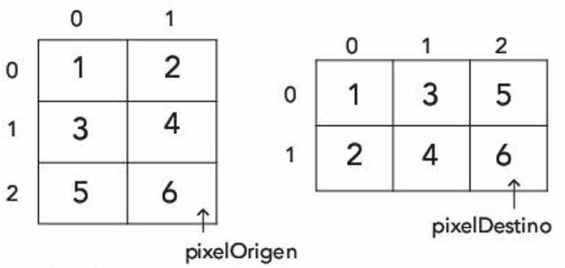
La rotación (voltear a lo largo de la diagonal) empieza con el mismo origen y destino, e incluso los mismos valores de las variables, pero como *usamos* el destino *X* y *Y* de manera distinta, obtenemos un efecto diferente. Para facilitar la comprensión de este problema, vamos a usar una pequeña matriz de números.

Ahora, a medida que incrementamos las variables *Y*, recorremos el arreglo de origen hacia *abajo*, pero *a lo largo* del arreglo de destino.




origenX = 0
origenY = 1
destinoX = origenY = 1
destinoY = origenX = 0

Al terminar hemos realizado la misma copia, pero el resultado es totalmente distinto.




origenX = 1
origenY = 2
destinoX = origenY = 2
destinoY = origenX = 1

¿Cómo giraríamos 90 grados *realmente*? Necesitamos pensar en dónde queremos cada pixel. El siguiente programa *realiza* una rotación de 90 grados de la imagen. La diferencia clave está en la llamada a la función `setColor`. Observe que intercambiamos los índices `x` y `y` de los ejemplos anteriores, pero usamos una ecuación para `y: anchura - destinoX - 1`. Trate de seguirla para convencerse de que es una verdadera rotación.

**Programa 33: girar una imagen**

```
def girarBarbDeLado():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  destinoX = 0
  anchura = getWidth(barb)
  for origenX in range(0,getWidth(barb)):
   destinoY = 0
   for origenY in range(0,getHeight(barb)):
     color = getColor(getPixel(barb,origenX,origenY))
     setColor(getPixel(lienzo,destinoY,anchura - destinoX - 1), color)
     destinoY = destinoY + 1
   destinoX = destinoX + 1
  show(barb)
  show(lienzo)
  return lienzo
```

■

¿Cómo realizaríamos una rotación diferente? ¿Tal vez de 45 grados? ¿O de 33.3 grados? Ahora sí se vuelve difícil, ya que los píxeles sólo están en posiciones enteras y discretas. Hay píxeles en (0, 0) y (1, 2), pero no en (0.33, 2.5). No hay coordenadas de píxeles con componentes decimales, por lo que calculamos la rotación con valores decimales y después averiguamos cómo traducirlos a coordenadas enteras. En este libro no veremos esos cálculos.

### 4.3.5 Cambio de escala

Una transformación muy común para las imágenes es cambiarlas de escala. Ampliar la escala significa hacerlas más grandes y reducir la escala significa hacerlas más pequeñas. Es común reducir la escala de una imagen de 1 o 3 megapíxeles a un tamaño más pequeño, para que sea más fácil colocarla en sitios Web. Las imágenes más pequeñas requieren menos espacio en disco y menos ancho de banda de red, por lo cual se descargan con mayor facilidad y rapidez.

Para reducir la escala de una imagen hay que usar el *muestreo*, que también usaremos con sonidos más adelante. Para hacer una imagen *más pequeña*, debemos tomar *uno de cada dos* píxeles al copiar del origen al destino. Para hacer una imagen *más grande*, vamos a tomar *todos los píxeles dos veces*.

La función más fácil es la de reducir la escala de una imagen. En vez de incrementar las variables *X* y *Y* del origen por 1, simplemente incrementamos por 2. Dividimos la cantidad de espacio entre 2, ya que llenaremos la mitad de espacio: nuestra longitud será (200 – 45)/2 y la altura será (200 – 25)/2, empezando todavía en (100, 100). El resultado es un pequeño rostro en el lienzo (figura 4.14).

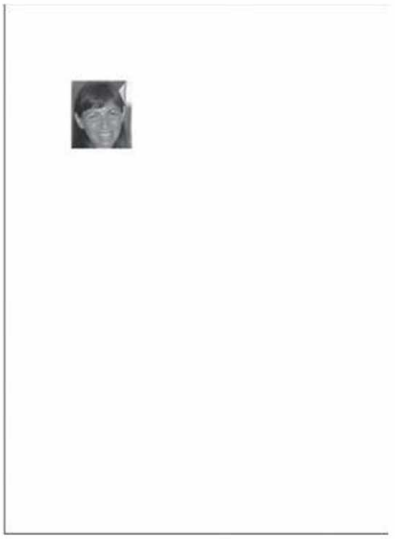


**FIGURA 4.14**
Reducir la escala de una imagen.

**Programa 34: reducir la escala de una imagen (más pequeña)**

```
def copiarRostroBarbMasChico():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
  origenX = 45
  for destinoX in range(100,100+((200-45)/2)):
    origenY = 25
    for destinoY in range(100,100+((200-25)/2)):
      color = getColor(getPixel(barb,origenX,origenY))
      setColor(getPixel(lienzo,destinoX,destinoY), color)
      origenY = origenY + 2
    origenX = origenX + 2
  show(barb)
  show(lienzo)
  return lienzo
```

■

## Cómo funciona

- Para empezar, creamos los objetos imagen: `barb` como nuestro origen, y `lienzo` en donde crearemos la imagen reducida de Barb.
- El rostro de Bárbara está en el rectángulo de (45, 25) a (200, 200). Esto significa que `origenX` empieza en 45 y `origenY` empieza en 25. No especificamos el final del rango para los índices de origen, ya que están controlados por los ciclos `for` para los índices de destino.

- Vamos a iniciar `destinoX` y `destinoY` en 100 cada uno. ¿Cuáles son los puntos finales para los rangos? Queremos obtener *todo* el rostro de Bárbara. La anchura del rostro de Bárbara es 200 – 45 (el índice `x` máximo menos el índice `x` mínimo). Como queremos encoger el rostro de Bárbara a la mitad (en cada dirección), vamos a omitir uno de cada dos píxeles. Esto significa que sólo tendremos la mitad de la anchura en la composición final: (200 – 45)/2 a lo largo. Si empezamos `destinoX` en 100, el punto final para el rango es de 100 más (200 – 45)/2. `destinoY` funciona de la misma forma.
- Puesto que deseamos omitir uno de cada dos píxeles en el origen, incrementamos `origenX` y `origenY` por 2 cada vez que se recorre el ciclo.

Reducir la escala de una imagen es, en sentido literal, descartar uno de cada dos píxeles. Es de por sí un proceso *con pérdidas*. Nos deshacemos de cierta información, descartando parte de la información de los píxeles que estaban en la imagen original. Esta *pérdida* es similar al caso de las imágenes JPEG, que descartan cierta información para facilitar la compresión de la imagen a un archivo más pequeño.

Aumentar la escala de la imagen (hacerla más grande) es un poco más complicado. Aquí debemos tomar cada pixel dos veces. Lo que vamos a hacer es incrementar las variables índices de origen por 0.5. En este caso no podemos hacer referencia al pixel 1.5, pero si hacemos referencia a *int* (1.5) (función entera) obtendremos 1 otra vez, y esto funcionará. La secuencia de 1, 1.5, 2, 2.5 se convertirá en 1, 1, 2, 2... El resultado es una forma más grande de la imagen (figura 4.15).

**FIGURA 4.15**
Aumentar la escala de una imagen.

**Programa 35: aumentar la escala de una imagen (más grande)**

```
def copiarRostroBarbMasGrande():
  # Establecer las imágenes de origen y de destino
  barbarch=getMediaPath("barbara.jpg")
  barb = makePicture(barbarch)
  lienzoarch = getMediaPath("7inX95in.jpg")
  lienzo = makePicture(lienzoarch)
  # Ahora, realizar el proceso de copiado
```

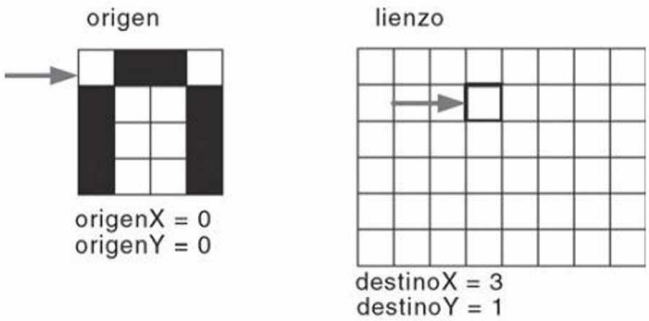
```
origenX = 45
for destinoX in range(100,100+((200-45)*2)):
  origenY = 25
  for destinoY in range(100,100+((200-25)*2)):
    color = getColor(getPixel(barb,int(origenX),int(origenY)))
    setColor(getPixel(lienzo,destinoX,destinoY), color)
    origenY = origenY + 0.5
  origenX = origenX + 0.5
show(barb)
show(lienzo)
return lienzo
```

■

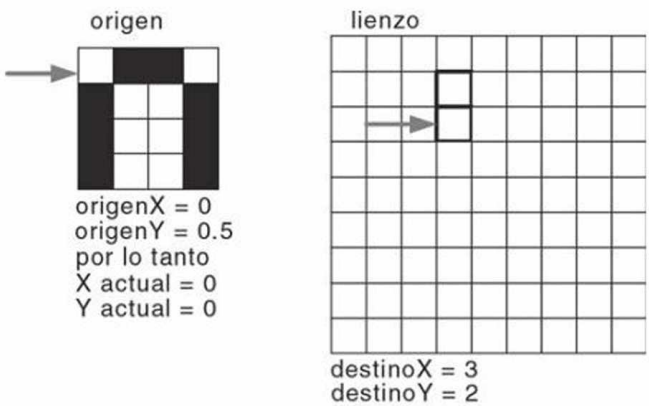
Tal vez desee poder ampliar una imagen a cierto tamaño específico, en vez de usar siempre las imágenes del lienzo. Hay una función llamada `makeEmptyPicture` que crea una imagen de una anchura y altura deseadas (ambas especificadas en píxeles). `makeEmptyPicture(640,480)` crea un objeto imagen de 640 píxeles de anchura por 480 píxeles de altura; justo igual que el lienzo.

### Cómo funciona

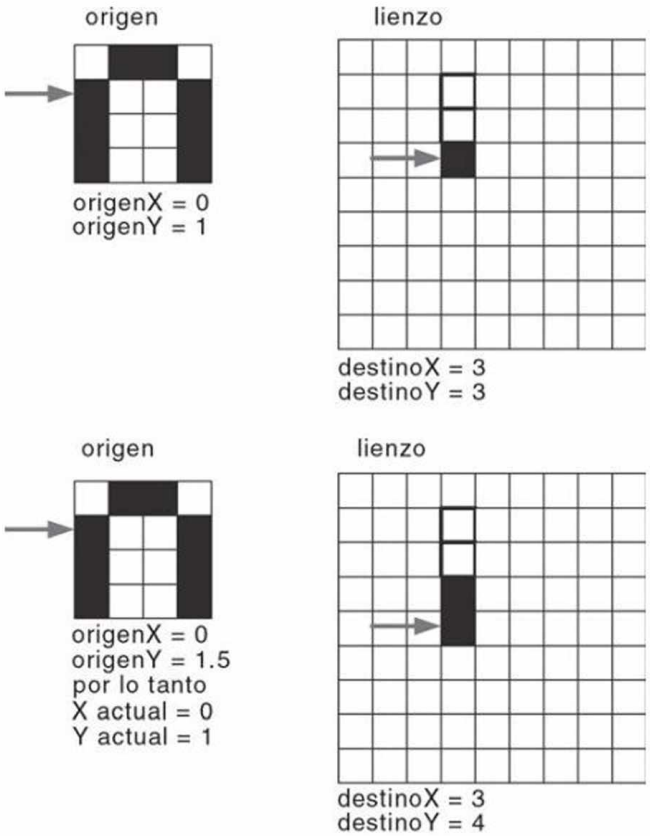
Empezamos desde el mismo lugar que la copia original.




Cuando incrementamos `origenY` por 0.5, terminamos haciendo referencia al mismo pixel en el origen, pero el destino se mueve al siguiente pixel.

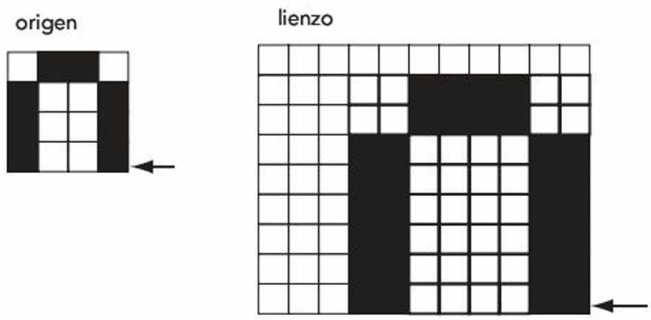



Cuando por segunda ocasión incrementamos `origenY` por 0.5, nos movemos ahora al siguiente pixel que terminará copiándose dos veces verticalmente. Por lo tanto, cada fila de pixel se copia dos veces.




Cuando cambiamos a una nueva columna en el destino, permanecemos en la *misma* columna en el origen. Observe que cada columna también aparece dos veces. Por ende, terminamos duplicando cada pixel tanto en la dimensión horizontal como vertical.

En un momento dado terminamos duplicando la imagen en ambas direcciones, con lo que se cuadruplica efectivamente el área de la figura. Tenga en cuenta que el resultado final se degrada un poco: es más entrecortado que el original. Ésta es una degradación de la calidad, pero no tiene *pérdidas* en el mismo sentido que la reducción de escala: no *perdimos* información que no estuviera en la imagen original. ¿Podríamos hacerla menos entrecortada? ¿Qué pasaría si no sólo copiara una segunda vez, sino que *mezclara* dos valores de color de pixel (por decir, a la izquierda y a la derecha)? Hay muchas formas de estimar el color que debe tener el valor duplicado para evitar lo entrecortado (a lo cual se le conoce como *pixelización*).



## RESUMEN DE PROGRAMACIÓN

| | |
|---|---|
| `range` | Función que crea una secuencia de números. Útil para crear índices para un arreglo o matriz. |
| `setMediaPath()` | Le permite elegir el directorio de medios usando un selector de archivos. |
| `setMediaPath(directorio)` | Le permite especificar el directorio de medios. |
| `getMediaPath(nombreBase)` | Recibe un nombre de archivo base y después regresa la ruta completa a ese archivo (suponiendo que está en el directorio de medios). |
| `makeEmptyPicture(anchura, altura)` | Toma una anchura y una altura y devuelve una imagen vacía (todo en blanco) del tamaño deseado. |

## PROBLEMAS

4.1
```
def nuevaFuncion(a, b, c):
    print a
    lista1 = range(0,4)
    valor = 0
    for x in lista1:
        print b
        valor = valor + 1
    print c
    print valor
```

Si llama a la función anterior escribiendo: `nuevaFuncion("Yo", "tú", "morsa")`, ¿qué imprimirá?

4.2 Hemos visto que, si incrementa el índice de imagen de origen por 2 mientas incrementa el índice de imagen de destino por 1 para cada pixel copiado, la imagen de origen reducirá su escala y se copiará a la imagen de destino. ¿Qué ocurre si incrementa el índice de la imagen de destino por 2 también? ¿Qué ocurre si incrementa ambos índices por 0.5 y usa `int` para obtener sólo la parte entera?

4.3 Escriba una función para reflejar a lo largo de la diagonal desde (0, 0) hasta (*anchura*, *altura*).

4.4 Escriba una función para reflejar a lo largo de la diagonal desde (0, *altura*) hasta (*anchura*, 0).

4.5 Escriba una función que aumente la escala de sólo una parte de la imagen. Trate de hacer más larga la nariz de alguien.

4.6 Escriba una función que reduzca la escala de sólo una parte de la imagen. Haga que la cabeza de alguien se vea más pequeña.

4.7 Escriba una función para voltear una imagen de modo que, si alguien estuviera viendo hacia la derecha, terminara viendo hacia la izquierda.

4.8 Escriba una función de recorte general que tome una imagen de origen, el valor de *X* inicial, el valor de *Y* inicial, el valor de *X* final y el valor de *Y* final. Cree y devuelva la nueva imagen y copie sólo el área especificada en la nueva imagen.

4.9 Escriba una función que copie distintas partes de una imagen a distintos lugares en el destino.

4.10 Escriba una función `aumentarEscala` general que reciba una imagen para luego crear y devolver una nueva imagen del doble de grande, usando `makeEmptyPicture (anchura,altura)`.

4.11 Escriba una función `reducirEscala` general que reciba una imagen para luego crear y devolver una nueva imagen que sea la mitad de grande, usando `makeEmptyPicture (anchura,altura)`.

4.12 Modifique cualquiera de las funciones del último capítulo para usar un ciclo anidado. Compruebe el resultado para asegurarse de que siga haciendo lo mismo.

4.13 Escriba una función para copiar un área triangular de una imagen a otra.

4.14 Escriba una función para reflejar los 20 píxeles de más a la izquierda de la imagen de entrada a los píxeles del 20 al 40.

4.15 Escriba una función que reduzca el rojo en el tercio superior de una imagen y borre el azul en el tercio inferior.

4.16 Escriba una función llamada `crearCollage` para crear un collage de la misma imagen que quepa al menos cuatro veces en la imagen JPEG **7in.x95in.jpg** en blanco (si lo desea, puede agregar imágenes adicionales). Una de esas cuatro copias puede ser la imagen original. Las otras tres deben ser formas modificadas. Puede cambiar la escala, recortar o girar la imagen, crear un negativo de ésta, cambiar o alterar los colores en la imagen y hacerla más oscura o más clara.

Después de componer su imagen, *refléjela*. Puede hacer esto en sentido vertical u horizontal (o de cualquier otra forma), en cualquier dirección; sólo asegúrese de que sus cuatro imágenes base sigan visibles después del reflejo.

Su función individual debe hacer que todo esto ocurra: todos los efectos y la composición deben ocurrir desde la función individual `crearCollage`. Claro que es perfectamente normal *usar* otras funciones, pero hágalo de tal forma que la persona que pruebe su programa sólo necesite llamar a `setMediaPath()`, colocar todas sus imágenes de entrada en un directorio **mediasources** y luego ejecutar `crearCollage()` para generar, mostrar y devolver un collage.

*4.17 Piense en la forma en que funciona el algoritmo de escalas de grises. En esencia, si conoce la *luminancia* de algo visual (por ejemplo, una pequeña imagen o una letra), puede reemplazar un pixel con ese elemento visual de la misma forma en que se crea una imagen de collage. Pruebe a implementar esto. Necesitará 256 elementos visuales de una claridad cada vez mayor, todos del mismo tamaño. Puede crear un collage si reemplaza cada pixel en la imagen original con uno de estos elementos visuales.

## PARA PROFUNDIZAR

La "biblia" de los gráficos de computadora es *Introduction to Computer Graphics* [14]. Lo recomendamos ampliamente.

*Problema más desafiante.

CAPÍTULO 5

# Técnicas de imágenes con selección y combinación


5.1 SUSTITUCIÓN DE COLORES: OJO ROJO, TONOS SEPIA Y POSTERIZACIÓN

5.2 COMBINACIÓN DE PÍXELES: DIFUMINADO

5.3 COMPARACIÓN DE PÍXELES: DETECCIÓN DE BORDES

5.4 MEZCLA DE IMÁGENES

5.5 SUSTRACCIÓN DE FONDO

5.6 CHROMAKEY

5.7 DIBUJAR SOBRE IMÁGENES

5.8 SELECCIONAR SIN VOLVER A PROBAR

5.9 PROGRAMAS PARA ESPECIFICAR EL PROCESO DE DIBUJO


## Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Implementar cambios de color controlados, como eliminación de ojos rojos, tonos sepia y posterización.
- Usar mezclas para combinar imágenes.
- Usar sustracción de fondo para separar las imágenes de primer plano de las de fondo, además de comprender cuándo y cómo funcionará.
- Usar la técnica chromakey para separar las imágenes de primer plano de las imágenes de fondo.
- Agregar texto y figuras a las imágenes existentes.
- Usar el difuminado para suavizar la degradación.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar condicionales para seleccionar ciertos píxeles.
- Poder elegir entre usar formatos de imágenes vectoriales y de mapas de bits.
- Poder elegir cuándo deberíamos escribir un programa para cierta tarea en vez de usar el software de aplicaciones existente.

## 5.1 SUSTITUCIÓN DE COLORES: OJO ROJO, TONOS SEPIA Y POSTERIZACIÓN

Es muy sencillo sustituir colores con otro color. Podemos hacerlo a grandes rasgos (a lo largo de toda la imagen) o sólo dentro de un rango (de valores $x$ y $y$). Tal vez sea de más utilidad cambiar colores que estén *cerca* del color que deseamos. Esta técnica nos permite crear algunos efectos interesantes a lo largo de toda la imagen, o ajustar el efecto para hacer algo específico en la imagen, como convertir el color blanco de los dientes de alguien en morado.

He aquí cómo obtenemos la técnica más interesante: hacemos que la computadora tome una *decisión*. En nuestro programa le decimos qué debe *evaluar* y, si la evaluación resulta verdadera, le decimos qué hacer. Le pedimos a la computadora que *seleccione* ciertos píxeles mediante una instrucción `if`.

Una instrucción `if` utiliza una *evaluación* y un bloque. Si la evaluación es verdadera, se ejecuta el bloque de código. La forma general es la siguiente:

```
if (una evaluación):
  # Estas instrucciones en Python se ejecutarán si la evaluación es verdadera
print "Esto se imprimirá sin importar que la evaluación sea o no verdadera"
```

La expresión "una evaluación" puede ser cualquier tipo de *expresión lógica*. `1 < 2` es una expresión lógica que siempre es verdadera. `a < 2` es una expresión lógica que depende del valor de `a`. Podemos evaluar con `<`, `<=` (menor o igual que), `==` (para igualdad), `>`, `>=` o `<>` (para desigualdad).

Para la técnica de sustitución de color necesitamos la forma de averiguar si un color está *cerca* de otro. Tenemos una función en JES para hacer eso, la cual se llama `distance`. Esta función devuelve un número que representa qué tan cerca están dos colores uno del otro. No averigua qué tan similares son los colores para el ojo humano, lo cual sería la mejor forma de hacerlo. En vez de ello, calcula una distancia euclidiana entre dos colores en el espacio de coordenadas cartesianas. Esto probablemente suene complicado, pero lo más seguro es que usted haya aprendido la fórmula para indicar la distancia entre $(x_1, y_1)$ y $(x_2, y_2)$ en la escuela.

$$\sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$$

Piense en lo que esto significaría para la distancia entre dos colores $(rojo_1, verde_1, azul_1)$ y $(rojo_2, verde_2, azul_2)$.

$$\sqrt{(\text{rojo}_1 - \text{rojo}_2)^2 + (\text{verde}_1 - \text{verde}_2)^2 + (\text{azul}_1 - \text{azul}_2)^2)}$$

He aquí cómo funcionan las distancias en JES:

```
>>> print red
color r=255 g=0 b=0
>>> print magenta
color r=255 g=0 b=255
>>> print pink
color r=255 g=175 b=175
>>> print black
color r=0 g=0 b=0
>>> print white
color r=255 g=255 b=255
```

```
>>> print distance(white, black)
441.6729559300637
>>> print distance(white, pink)
113.13708498984761
>>> print distance(black, pink)
355.3519382246282
>>> print distance(magenta, pink)
192.41881404893857
>>> print distance(red, magenta)
255.0
>>> print distance(red, pink)
247.48737341529164
```

Podemos usar la función `distance` como parte de una evaluación para que `if` seleccione sólo los píxeles cuyos colores estén cerca de los que deseamos cambiar. Si la distancia a partir de un color de pixel dado hasta un color que deseamos cambiar es menor que cierto valor, podemos cambiar el color, ya que consideramos que está lo "bastante cerca".

He aquí un programa que sustituye el color café del cabello de Katie con rojo. Usamos la herramienta de imágenes de JES para averiguar aproximadamente cuáles eran los valores RGB para el cabello color café de Katie, y después escribimos un programa para buscar colores cercanos a éste e incrementamos el componente rojo de esos píxeles. Jugamos mucho con el valor que usamos para la distancia (en este caso, 50.0) y con la cantidad por la que multiplicamos el valor rojo (en este caso, 2). El resultado es que el sofá detrás de Katie aumenta su componente rojo también (figura 5.1).

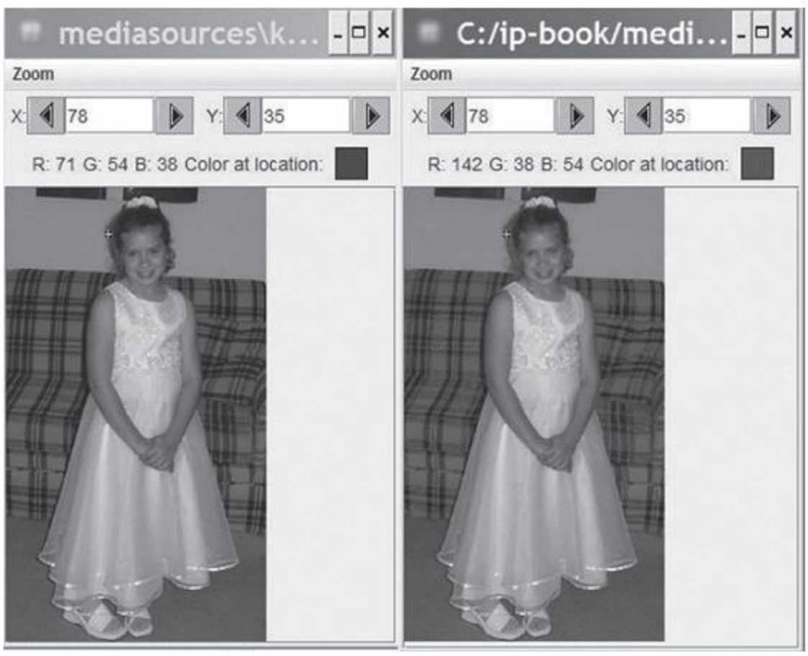



**FIGURA 5.1**
Cambiar café a rojo.

**Programa 36: convertir a Katie en pelirroja**

```
def convertirEnRojo():
  cafe = makeColor(42,25,15)
  archivo="C:/ip-book/mediasources/katieFancy.jpg"
  imagen=makePicture(archivo)
  for px in getPixels(imagen):
    color = getColor(px)
    if distance(color,cafe)<50.0:
      nivelrojo=int(getRed(px)*2)
      nivelazul=getBlue(px)
      nivelverde=getGreen(px)
      setColor(px,makeColor(nivelrojo,nivelazul,nivelverde))
  show(imagen)
  return(imagen)
```

■

## Cómo funciona

En realidad esto es bastante similar a nuestro programa para aumentar el color rojo, sólo que usa un método alternativo para establecer el color.

- Creamos un color `cafe`, que es lo que encontramos en el cabello de Katie usando la herramienta de imágenes en JES.
- Creamos la imagen de Katie.
- Para cada uno de los píxeles `px` en la imagen, obtenemos el color y después lo comparamos con el color `cafe` que identificamos antes. Queremos saber si el color en el pixel `px` está *bastante cerca* de `cafe`. ¿Cómo definimos "bastante cerca"? Decimos que está dentro de un máximo de 50.0. ¿De dónde obtuvimos ese número? Probamos con 10.0, pero cambió muy poco. Luego probamos con 100.0 y había muchas coincidencias (como las rayas en el sofá detrás de la cabeza de Katie). Probamos con distintos números hasta que obtuvimos el efecto deseado.
- Si el color está lo "bastante cerca", obtenemos los componentes rojo, verde y azul del color en `px`. Duplicamos el rojo multiplicándolo por 2.
- Después establecemos el color en `px` a un nuevo color con el rojo ajustado y los mismos componentes azul y verde. Después avanzamos al siguiente pixel.

Con la herramienta para imágenes de JES también podemos averiguar las coordenadas justo alrededor del rostro de Katie para luego ajustar los cafés cerca de su rostro. El efecto no es muy bueno, aunque podemos ver con claridad que funcionó. La línea de nivel de rojo es demasiado pronunciada y rectangular (figura 5.2).

**Programa 37: sustitución de color en un rango**

```
def convertirRojoEnRango():
  cafe = makeColor(42,25,15)
  archivo="C:/ip-book/mediasources/katieFancy.jpg"
  imagen=makePicture(archivo)
  for x in range(63,125):
    for y in range(6,76):
      px=getPixel(imagen,x,y)
      color = getColor(px)
      if distance(color,cafe)<50.0:
```

```
        nivelrojo=int(getRed(px)*2)
        nivelazul=getBlue(px)
        nivelverde=getGreen(px)
        setColor(px,makeColor(nivelrojo,nivelazul,nivelverde))
show(imagen)
return(imagen)
```

■

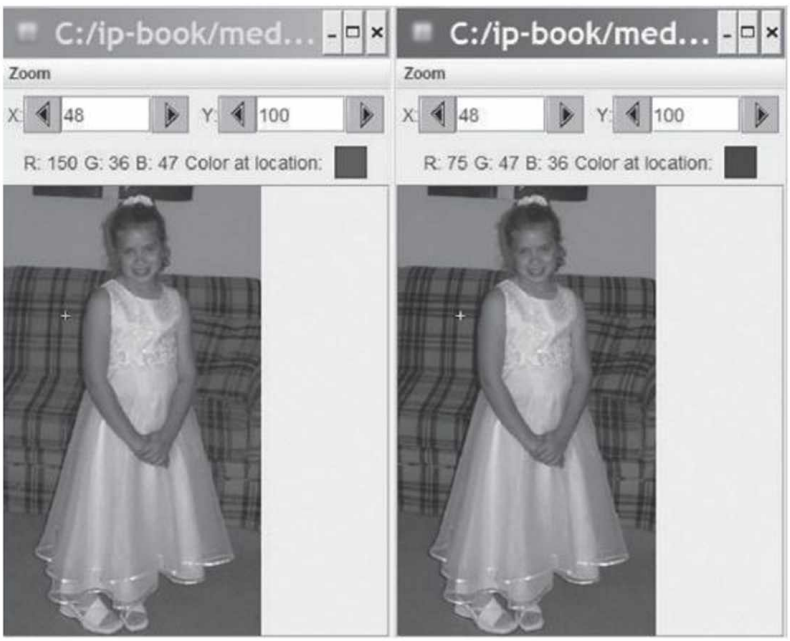



**FIGURA 5.2**
Duplicar el color rojo en un rango de área rectangular.

### 5.1.1 Reducción del ojo rojo

El "ojo rojo" es el efecto en donde el destello de la cámara rebota en la parte trasera de los ojos del sujeto. La reducción del ojo rojo es en realidad un proceso simple. Buscamos los píxeles que estén "bastante cerca" (una distancia del rojo de 165 funciona bien) al rojo y después insertamos un color de sustitución.

Quizás no sea conveniente cambiar toda la imagen. En la figura 5.3 Jenny usa un vestido rojo. No queremos borrar ese vestido. Para corregir los ojos rojos sólo cambiaremos el *rango* en el que se encuentran los ojos de Jenny. Mediante la herramienta para imágenes de JES buscamos las esquinas superior izquierda e inferior derecha de sus ojos. Estos puntos son (109, 91) y (202, 107).

**Programa 38: reducción del ojo rojo**

```
def eliminarOjoRojo(imag,xInicial,yInicial,xFinal,yFinal,colorSustitucion):
  rojo = makeColor(255,0,0)
  for x in range(xInicial, xFinal):
    for y in range(yInicial, yFinal):
      pixelActual = getPixel(imag,x,y)
      if (distance(rojo,getColor(pixelActual)) < 165):
        setColor(pixelActual,colorSustitucion)
```

■

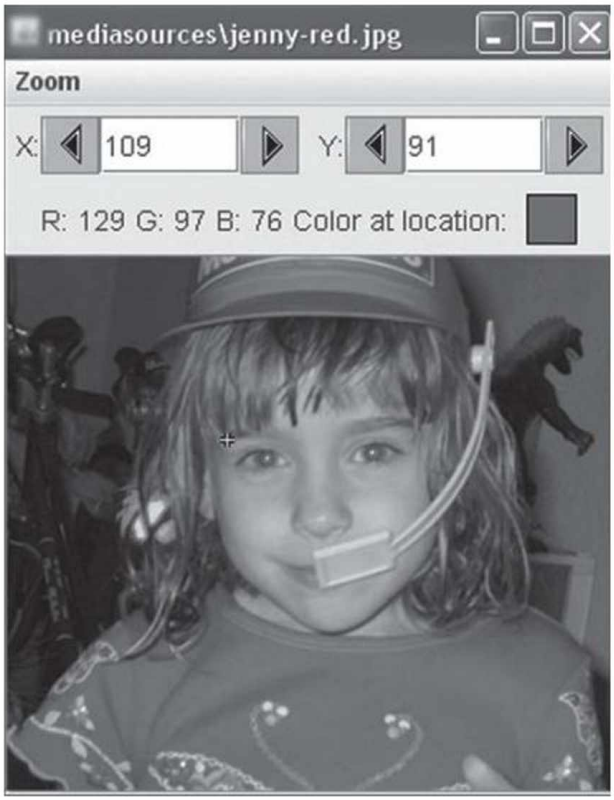



**FIGURA 5.3**
Buscamos el rango en donde los ojos de Jenny son rojos.

Para llamar a esta función y sustituir el rojo con negro, hacemos lo siguiente:

```
>>> jenny = makePicture(getMediaPath("jenny-red.jpg"))
>>> explore(jenny)
>>> eliminarOjoRojo(jenny, 109, 91, 202, 107, makeColor(0, 0, 0))
>>> explore(jenny)
```

Sin duda podrían usarse otros colores de sustitución. El resultado fue bueno; podemos verificar que en realidad el ojo ahora tiene píxeles de color negro solamente (figura 5.4).

## Cómo funciona

Este algoritmo es en sí muy similar al que usamos para cambiar el cabello de Katie a rojo.

- Esta función se escribió para que fuera útil para diversas imágenes, por lo que recibe muchas entradas (*parámetros*) que el usuario sólo tiene que cambiar para los distintos programas. La función recibe una imagen como entrada, junto con las coordenadas de donde debe realizarse el cambio y luego el color con el que se va a sustituir el rojo.
- Definimos el rojo como un rojo puro y fuerte: `makeColor(255,0,0)`.
- Para `x` y `y` dentro del rectángulo que se proporciona como entrada, obtenemos el `pixelActual`.

**FIGURA 5.4**
Verificamos que el color rojo haya cambiado a negro.

- Revisamos si el `pixelActual` está lo bastante cerca del rojo. Para determinar qué es "bastante cerca" buscamos una distancia dentro de un *valor de umbral*. Probamos distintas distancias y nos quedamos con 165 como la distancia que atrapaba la mayor parte del nivel de rojo en el ojo que nos importaba.
- Después intercambiamos el `colorSustitucion` por el color "bastante cercano" en el pixel `pixelActual`.

### 5.1.2 Imágenes con tono sepia y posterización: uso de condicionales para elegir el color

Hasta ahora hemos realizado la sustracción de colores mediante el simple reemplazo de un color con otro. Pero podemos ser más sofisticados en cuanto al intercambio de colores. Podemos buscar un rango de colores mediante el uso de la instrucción `if` y optar por sustituir cierta función del color original, o cambiarlo a un color específico. Los resultados son bastante interesantes.

Por ejemplo, tal vez nos interese generar impresiones en tonos sepia. Las impresiones antiguas algunas veces tienen un tinte amarillento. Podríamos realizar un simple cambio de color en general, pero el resultado final no es placentero a la vista. Si buscamos distintos tipos de colores (relieves, sombras) y los tratamos de manera distinta, podemos obtener un efecto mejorado (figura 5.5).

**FIGURA 5.5**
Escena original (*izquierda*) y después de usar nuestro programa de tonos sepia (*derecha*).

La forma de hacerlo es convertir primero todo a escalas de grises; esto debido a que las impresiones antiguas estaban en escala de grises, además de que es un poco más fácil trabajar de esa manera. Después buscamos rangos altos, medios y bajos de color (en realidad, luminancia), y los modificamos por separado (¿por qué estos valores específicos? Debido a la prueba y error: ajustándolos hasta que nos guste el efecto).

**Programa 39: conversión de una imagen a tonos sepia**

```
def tinteSepia(imagen):
  #Convertir la imagen en escala de grises
  escalaGrisesNuevo(imagen)

  #iterar a través de la imagen para entintar los píxeles
  for p in getPixels(imagen):
    rojo = getRed(p)
    azul = getBlue(p)

    #entintar las sombras
    if (rojo < 63):
      rojo = rojo*1.1
      azul = azul*0.9

    #entintar los medios tonos
    if (rojo > 62 and rojo < 192):
      rojo = rojo*1.15
      azul = azul*0.85

    #entintar los relieves
    if (rojo > 191):
      rojo = rojo*1.08
      if (rojo > 255):
        rojo = 255
      azul = azul*0.93

    #establecer los nuevos valores de colores
    setBlue(p, azul)
    setRed(p, rojo)
```

■

### Cómo funciona

Primero, la función toma una imagen como entrada, después usa nuestra función `escalaGrisesNuevo` para convertirla en escala de grises (le recomendamos que copie la función `escalaGrisesNuevo` en el área del programa, junto con `tinteSepia`; no aparece en el listado anterior). Para cada uno de los píxeles, obtenemos el nivel rojo y azul del pixel. Sabemos que rojo y azul tendrán los *mismos* valores, ya que la imagen está ahora en gris, pero queremos el nivel de rojo y el nivel de azul para *cambiarlo*. Buscamos rangos específicos de colores y los tratamos de manera distinta. Observe que al entintar los relieves (en donde la luz es más brillante), tenemos un `if` dentro de un bloque `if`. La idea aquí es que no queremos que los valores se reinicien; si el rojo se eleva demasiado, queremos limitarlo a 255. Por último, establecemos los valores azul y rojo a los nuevos valores de rojo y azul, y avanzamos al siguiente pixel.

La posterización es un proceso muy similar, que se produce al convertir una imagen a un número más pequeño de colores. Para hacer esto analizamos un rango específico de valores y luego establecemos todos los valores en ese rango a *un* solo valor. El resultado es que reducimos el número de colores en la imagen (figura 5.6). Por ejemplo, en el siguiente programa, si rojo es 1.2, 3, ... , o 64, lo convertimos en 31. De esta forma eliminamos todo un rango de variación del rojo y lo convertimos en un solo valor rojo específico. Probamos esto en la fotografía de un estudiante de computación llamado Anthony.

**FIGURA 5.6**
Reducción de los colores (*derecha*) a partir del original (*izquierda*).

```
>>> archivo = "c:/ip-book/mediasources/anthony.jpg"
>>> estudiante = makePicture(archivo)
>>> explore(estudiante)
>>> posterizar(estudiante)
>>> explore(estudiante)
```

**Programa 40: posterización de una imagen**

```
def posterizar(imagen):

  #iterar a través de los píxeles
  for p in getPixels(imagen):
    #obtener los valores RGB
    rojo = getRed(p)
    verde = getGreen(p)
    azul = getBlue(p)

    # revisar y establecer los valores de rojo
    if (rojo < 64):
      setRed(p, 31)
    if (rojo > 63 and rojo < 128):
      setRed(p, 95)
    if (rojo > 127 and rojo < 192):
      setRed(p, 159)
    if (rojo > 191 and rojo < 256):
      setRed(p, 223)

    #revisar y establecer los valores de verde
    if (verde < 64):
      setGreen(p, 31)
    if (verde > 63 and verde < 128):
      setGreen(p, 95)
    if (verde > 127 and verde < 192):
      setGreen(p, 159)
    if (verde > 191 and verde < 256):
      setGreen(p, 223)

    #revisar y establecer los valores de azul
    if (azul < 64):
      setBlue(p, 31)
    if (azul > 63 and azul < 128):
      setBlue(p, 95)
    if (azul > 127 and azul < 192):
      setBlue(p, 159)
    if (azul > 191 and azul < 256):
      setBlue(p, 223)
```

■

Hay un efecto interesante que se produce al usar las funciones de escala de grises y de posterización juntas. Para ello calculamos una luminancia y luego sólo establecemos el color del pixel en blanco o negro: sólo dos niveles. El resultado es una imagen que parece estampada o algo similar a un dibujo en carbón (figura 5.7).

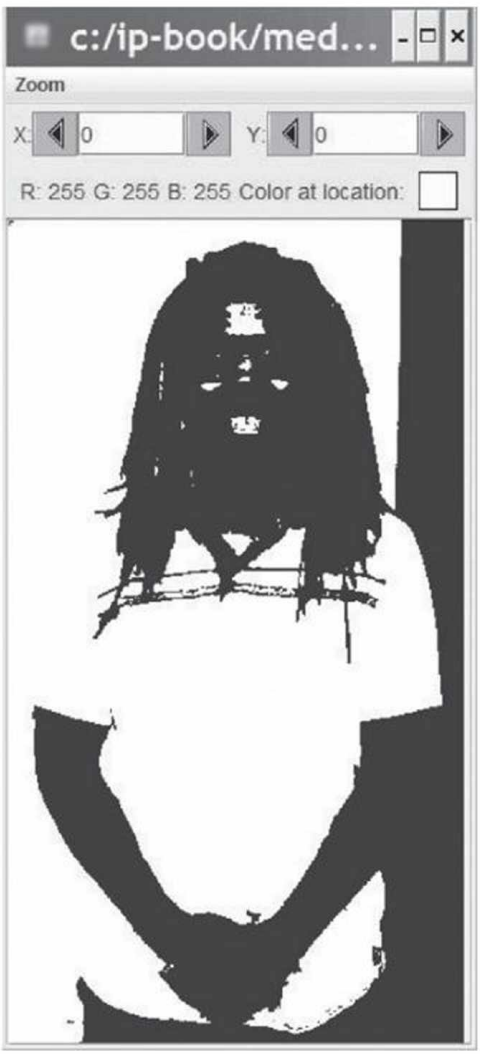



**FIGURA 5.7**
Imagen posterizada en dos niveles de gris.

**Programa 41: posterización en dos niveles de gris**

```
def posterizarGris(imag):
  for p in getPixels(imag):
    r = getRed(p)
    g = getGreen(p)
    b = getBlue(p)
    luminancia = (r+g+b)/3
    if luminancia < 64:
      setColor(p,black)
    if luminancia >= 64:
      setColor(p,white)
```

■

## 5.2 COMBINACIÓN DE PÍXELES: DIFUMINADO

Al hacer las imágenes más grandes (aumentar su escala), por lo general obtenemos bordes ásperos: escalones pronunciados en líneas, lo que denominamos **pixelación**. Podemos reducir la pixelación al *difuminar* la imagen: hacer de manera deliberada que algunos de los bordes ásperos sean "suaves" (es decir, más uniformes y curveados). Ésta es una pérdida de información, pero hace que la imagen sea más placentera a la vista.

Existen *muchas* formas (algoritmos) de difuminado. En este caso usaremos una que es realmente muy simple. Lo que haremos es establecer cada pixel a un color que sea un *promedio* de los colores de los píxeles a su alrededor.

**Programa 42: un difuminado simple**

```
def difuminar(nombrearchivo):
  origen=makePicture(nombrearchivo)
  destino=makePicture(nombrearchivo)
  for x in range(1, getWidth(origen)-2):
    for y in range(1, getHeight(origen)-2):
      superior = getPixel(origen,x,y-1)
      izquierda = getPixel(origen,x-1,y)
      inferior = getPixel(origen,x,y+1)
      derecha = getPixel(origen,x+1,y)
      centro = getPixel(destino,x,y)
      nuevoRojo=(getRed(superior) + getRed(izquierda) + getRed(inferior) +
getRed(derecha)¬ + getRed(centro))/5
      nuevoVerde=(getGreen(superior) + getGreen(izquierda) + getGreen(inferior)¬
+ getGreen(derecha) + getGreen(centro))/5
      nuevoAzul=(getBlue(superior) + getBlue(izquierda) + getBlue(inferior) +
getBlue¬ (derecha) + getBlue(centro))/5
      setColor(centro, makeColor(nuevoRojo, nuevoVerde, nuevoAzul))
  return destino
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un solo comando en Python no puede dividirse entre varias líneas.

■

**Tip de funcionamiento: no divida las líneas a la mitad en Python**

En ejemplos como éstos tal vez vea que las líneas se "ajustan". Las líneas que usted esperaría estuvieran en una sola línea aparecen en dos. Esto es por la necesidad de ajustar el código en la página, pero en realidad no podemos dividir las líneas así en Python. No podemos dividir líneas presionando Intro sino hasta completar la instrucción o expresión.

■

La figura 5.8 muestra cómo se aumenta el tamaño de la flor del collage y luego se difumina. Puede ver la pixelación en la versión más grande: los bordes ásperos con aspecto de bloques. Con el difuminado, parte de la pixelación desaparece. Los difuminados más cuidadosos toman regiones de colores en cuenta (de modo que los bordes entre los colores se mantengan pronunciados) y, por ende, pueden reducir la pixelación sin eliminar nitidez.

### Cómo funciona

Usamos esta función con algo como `nuevaImagen = difuminar (pickAFile())`. Después creamos dos copias del nombre de archivo de imagen que se proporciona como entrada. Modificamos sólo el `destino`, de modo que siempre obtengamos un promedio de los colo-

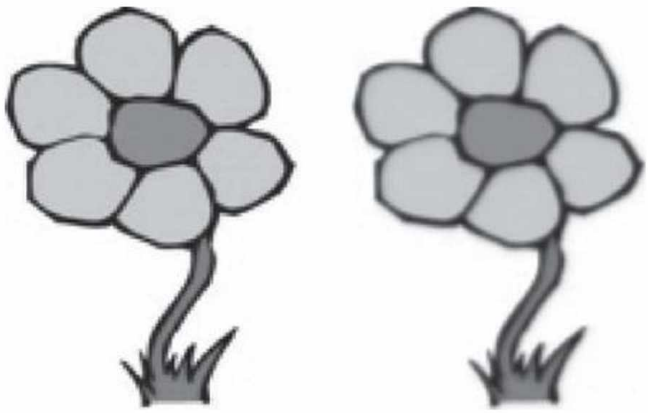


**FIGURA 5.8**
Aumentar el tamaño de la flor (*izquierda*) y después difuminarla para reducir la pixelación (*derecha*).

res *originales* del `origen`. Recorremos los valores índice de $x$ desde 1 hasta la anchura menos 1; aquí aprovechamos de manera explícita el hecho de que `rango` no incluye los valores finales. Hacemos lo mismo con los índices de $y$. La razón es que vamos a sumar 1 y restar 1 de cada $x$ y $y$ para calcular los promedios. No promediamos los píxeles de la orilla, pero esto es muy difícil de detectar. Para cada $(x, y)$ obtenemos el pixel a la izquierda $(x - 1, y)$, a la

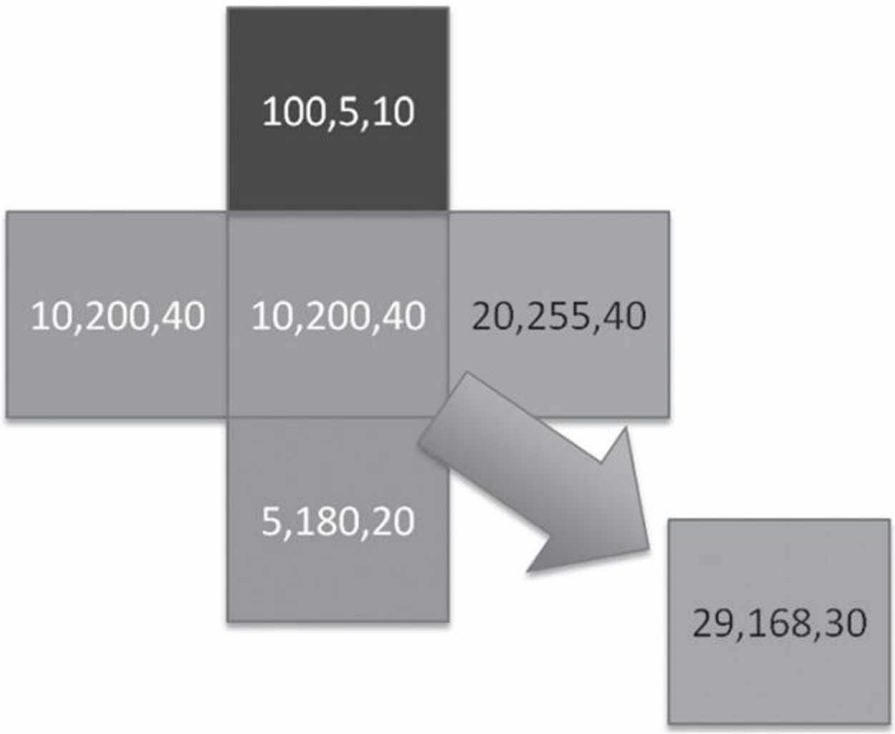



**FIGURA 5.9**
Ejemplo de un cálculo de difuminado.

derecha $(x + 1, y)$, por encima $(x, y - 1)$ y por debajo $(x, y + 1)$, así como el mismo pixel en el centro $(x, y)$ (de `destino`, ya que no podemos estar seguros de que no hayamos cambiado $(x, y)$ todavía). Calculamos el promedio de los rojos de los cinco píxeles, los verdes y los azules, y después establecemos el color en $(x, y)$ al color promedio.

Imaginemos que buscamos el pixel en el centro de la figura 5.9, cuyos valores (*rojo*, *verde*, *azul*) (RGB) son (10, 200, 40). Vamos a promediar sus valores RGB con los píxeles arriba (superior), a la izquierda, a la derecha y debajo (inferior). Para calcular el nuevo valor rojo, obtenemos el promedio de 100 (superior), 10 (izquierda), 10 (centro), 20 (derecha) y 5 (inferior), lo que nos da un valor de 29. Repetimos el promedio para obtener el verde (168) y el azul (30). Tenga en cuenta que *establecemos* el pixel en el `destino`, mientras leemos los píxeles del `origen`. Si usáramos sólo una imagen para leer y establecer, no obtendríamos el difuminado que queremos debido a que estableceríamos píxeles cuyos valores estaríamos leyendo después.

Aunque esto funciona sorprendentemente bien (figura 5.8), no es tan bueno como lo que podríamos obtener. Se pierde cierto detalle al realizar un difuminado simple como éste. ¿Qué pasaría si difumináramos para deshacernos de la pixelación sin perder el detalle? ¿Cómo podríamos hacerlo? ¿Qué tal si revisáramos los valores de luminancia antes de calcular el promedio? Tal vez no deberíamos difuminar a lo largo de límites extensos de luminancia, debido a que esto reduciría el detalle. Es sólo una idea: hay muchos algoritmos buenos para difuminado.

## 5.3 COMPARACIÓN DE PIXELES: DETECCIÓN DE BORDES

El difuminado es el proceso de calcular un promedio a través de varios píxeles. La *detección de bordes* es un proceso similar, en donde comparamos píxeles para determinar qué colocar en un pixel dado, pero en esencia sólo establecemos el pixel al color blanco o negro. La idea es tratar de dibujar líneas de la manera en que un artista bosquejaría un dibujo.

Realmente es una característica sorprendente de nuestros sistemas visuales el hecho de que podamos ver el dibujo lineal de alguien y detectar un rostro u otras características. Mire el mundo a su alrededor. En realidad no hay líneas pronunciadas que definen las características del mundo. No hay líneas claras alrededor de su nariz o de sus ojos, pero cualquier niño puede dibujar un rostro con una marca de verificación como nariz y dos círculos para los ojos, ¡y todos lo reconocemos como un rostro! Por lo general, *vemos* una línea en donde hay una diferencia en luminancia.

Podemos tratar de modelar este proceso (figura 5.10). En el siguiente programa comparamos la luminancia de cada pixel con el pixel que tiene *debajo* y el que está a su *derecha*. Si hay una diferencia considerable en la luminancia debajo y a la derecha, entonces hacemos el pixel negro (la experimentación nos enseñó que un valor de umbral de 10 funciona bastante bien). De lo contrario, lo dejamos blanco.

**Programa 43: creación de un dibujo lineal simple usando la detección de bordes**

```
def deteccionLineas(nombrearchivo):
  orig = makePicture(nombrearchivo)
  hacerBn = makePicture(nombrearchivo)
  for x in range(0,getWidth(orig)-1):
    for y in range(0,getHeight(orig)-1):
      aqui=getPixel(hacerBn,x,y)
      abajo=getPixel(orig,x,y+1)
      derecha=getPixel(orig,x+1,y)
```

```
    aquiL=(getRed(aqui)+getGreen(aqui)+getBlue(aqui))/3
    abajoL=(getRed(abajo)+getGreen(abajo)+getBlue(abajo))/3
    derechaL=(getRed(derecha)+getGreen(derecha)+getBlue(derecha))/3
    if abs(aquiL-abajoL)>10 and abs(aquiL-derechaL)>10:
      setColor(aqui,black)
    if abs(aquiL-abajoL)<=10 or abs(aquiL-derechaL)<=10:
      setColor(aqui,white)
show(hacerBn)
return hacerBn
```

■

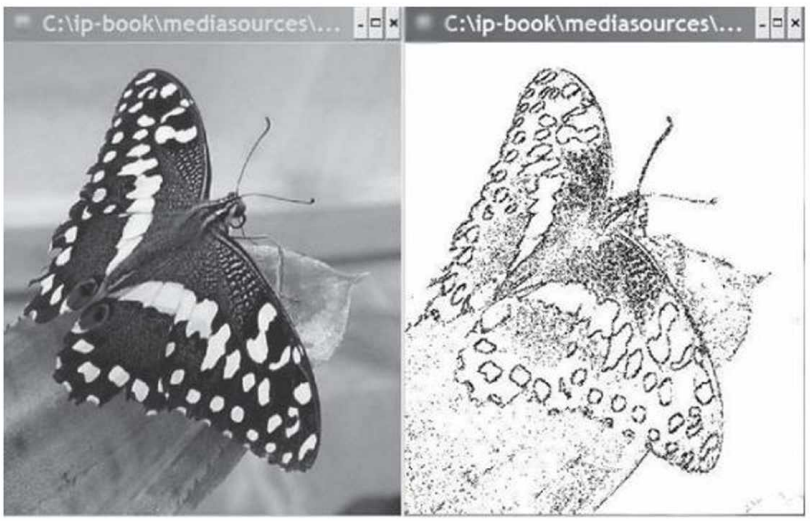



**FIGURA 5.10**
Una mariposa (*izquierda*) convertida en un "dibujo lineal" (*derecha*).

## Cómo funciona

Al igual que la posterización en niveles de gris, el objetivo aquí es establecer cada pixel a negro o blanco, dependiendo de si hay una diferencia en la luminancia o no.

- Llamamos a la función con algo como `versionBn = deteccionLineas(pickAFile())`. Creamos dos copias de la imagen del nombre del archivo de entrada: la fuente `orig` (original) y el destino `hacerBn`.
- Usamos índices para $x$ y $y$ desde 1 hasta la anchura menos 1 y la altura menos 1, pero aquí continuamos con el hecho de que el rango (`range`) no incluirá el último valor. Vamos a comparar el pixel en $(x, y)$ con los píxeles en $(x + 1, y)$ (derecha) y en $(x, y + 1)$ (abajo).
- Obtenemos el pixel `aqui`, el pixel `abajo` y el pixel a la `derecha`.
- Calculamos la luminancia de los tres píxeles.
- Si el *valor absoluto* (`abs`) de la diferencia entre la luminancia en el pixel de comparación (`aquiL`) y la luminancia tanto a la derecha (`derechaL`) como abajo (`abajoL`) es mayor que el valor de umbral de 10, entonces hacemos el pixel negro. Usamos el valor

absoluto ya que sólo nos importa la *diferencia* entre los dos valores de luminancia. En realidad no nos importa cuál es más grande.

- Si la diferencia entre la luminancia *no* es mayor que el valor de umbral (menor o igual a éste), entonces establecemos el píxel a blanco.

El último programa es bastante difícil de leer, ya que hay ciertas ecuaciones complicadas del lado derecho. Al *sumar* dos funciones podemos hacer el programa un poco más claro, y un poco más fácil de seguir para los humanos. Aquí creamos una función para calcular la luminancia a partir de un pixel `obtenLum` y una función para indicarnos si las diferencias entre los píxeles aquí, abajo y derecha sugieren una línea, `esLinea`. He aquí el código que funciona de la misma forma, sólo que usando algunas *funciones auxiliares* adicionales.

**Programa 44: creación de un dibujo lineal simple usando detección de bordes**

```
def obtenLum(pixel):
  return (getRed(pixel)+getGreen(pixel)+getBlue(pixel))/3

def esLinea(aqui,abajo,derecha):
  return abs(aqui-abajo)>10 and abs(aqui-derecha)>10

def deteccionLineas(nombrearchivo):
  orig = makePicture(nombrearchivo)
  hacerBn = makePicture(nombrearchivo)
  for x in range(0,getWidth(orig)-1):
    for y in range(0,getHeight(orig)-1):
      aqui=getPixel(hacerBn,x,y)
      abajo=getPixel(orig,x,y+1)
      derecha=getPixel(orig,x+1,y)
      aquiL=obtenLum(aqui)
      abajoL=obtenLum(abajo)
      derechaL=obtenLum(derecha)
      if esLinea(aquiL,abajoL,derechaL):
        setColor(aqui,black)
      if not esLinea(aquiL,abajoL,derechaL):
        setColor(aqui,white)
  show(hacerBn)
  return hacerBn
```

■

Existen algoritmos que realizan procesos de detección de bordes y dibujo lineal de una manera mucho más conveniente. Por ejemplo, aquí sólo establecemos cada pixel a negro o blanco. En realidad no estamos considerando la noción de una "línea". Podríamos usar técnicas como el difuminado para suavizar la imagen y hacer que los puntos se parezcan más a las líneas. También podríamos hacer los píxeles sólo negros si vamos a hacer los píxeles cercanos negros; esto es, crear una línea en vez de sólo hacer puntos.

## 5.4 MEZCLA DE IMÁGENES

En este capítulo hablaremos sobre las técnicas para crear imágenes a partir de otras piezas. Compondremos imágenes de nuevas formas (por ejemplo, extraer a alguien de un fondo y colocarlo en un nuevo entorno) y crearemos imágenes a partir de cero, sin tener que establecer cada pixel de manera explícita.

**FIGURA 5.11**
Mezcla de las imágenes de madre e hija.

Una de las maneras en que podemos combinar imágenes para crear otras nuevas es mezclando los colores de los píxeles para reflejar ambas imágenes. Cuando creamos collages mediante copias, cualquier traslape significa por lo general que una imagen aparece *sobre* otra. La última imagen que se pinta es la que aparece encima de la otra. Pero no tiene que ser así. Podemos *mezclar* imágenes si multiplicamos sus colores y los sumamos. Esto nos da el efecto de *transparencia.*

Sabemos que el 100% de algo es todo completo; el 50% de un elemento y el 50% de otro también es todo. En el siguiente programa mezclamos una imagen de la mamá y la hija con un traslape de 70 (la anchura de Bárbara menos 150) columnas de píxeles (figura 5.11).

**Programa 45: mezcla de dos imágenes**

```
def mezclarImagenes():
  barb = makePicture(getMediaPath("barbara.jpg"))
  katie = makePicture(getMediaPath("Katie-smaller.jpg"))
  lienzo = makePicture(getMediaPath("640x480.jpg"))
  # Copiar las primeras 150 columnas de Barb
  origenX=0
  for destinoX in range(0,150):
    origenY=0
    for destinoY in range(0,getHeight(barb)):
      color = getColor(getPixel(barb,origenX,origenY))
      setColor(getPixel(lienzo,destinoX,destinoY),color)
      origenY = origenY + 1
    origenX = origenX + 1
```

```
#Ahora, obtener el resto de Barb
# al 50% Barb y 50% Katie
traslape = getWidth(barb)-150
origenX=0
for destinoX in range(150,getWidth(barb)):
  origenY=0
  for destinoY in range(0,getHeight(katie)):
    bPixel = getPixel(barb,origenX+150,origenY)
    kPixel = getPixel(katie,origenX,origenY)
    nuevoRojo = 0.50*getRed(bPixel)+0.50*getRed(kPixel)
    nuevoVerde = 0.50*getGreen(bPixel)+0.50*getGreen(kPixel)
    nuevoAzul = 0.50*getBlue(bPixel)+0.50*getBlue(kPixel)
    color = makeColor(nuevoRojo,nuevoVerde,nuevoAzul)
    setColor(getPixel(lienzo,destinoX,destinoY),color)
    origenY = origenY + 1
  origenX = origenX + 1
# Últimas columnas de Katie
origenX=traslape
for destinoX in range(150+traslape,150+getWidth(katie)):
  origenY=0
  for destinoY in range(0,getHeight(katie)):
    color = getColor(getPixel(katie,origenX,origenY))
    setColor(getPixel(lienzo,destinoX,destinoY),color)
    origenY = origenY + 1
  origenX = origenX + 1
show(lienzo)
return lienzo
```

■

### Cómo funciona

Esta función consta de tres partes: la parte en donde Barb está sin Katie, la parte en donde hay algo de cada quien y la parte en donde sólo está Katie.

- Primero creamos los objetos imagen para `barb`, `katie` y el `lienzo` de destino.
- Para 150 columnas de píxeles, simplemente copiamos píxeles de `barb` al lienzo.
- La siguiente sección es la mezcla en sí. Nuestro índice `destinoX` empieza en 150, debido a que ya tenemos 150 columnas de `barb` en el lienzo. Usamos `origenX` y `origenY` para indexar *tanto* a `barb` como a `katie`, pero tenemos que sumar 150 a `origenX` al indexar a `barb`, puesto que ya hemos copiado 150 píxeles de `barb`. Nuestro índice *y* sólo llegará hasta la altura de la imagen de `katie` debido a que es más corta que la imagen de `barb`.
- El cuerpo del ciclo es en donde ocurre la mezcla. Obtenemos un pixel de `barb` y lo llamamos `bPixel`. Obtenemos uno de `katie` y lo llamamos `kPixel`. Después calculamos el rojo, verde y azul para el pixel de destino (el que está en `destinoX` y `destinoY`), para lo cual tomamos el 50% del rojo, verde y azul de cada una de las imágenes de origen.
- Por último, terminamos copiando el resto de los píxeles de `katie`.

## 5.5 SUSTRACCIÓN DE FONDO

Supongamos que tiene una imagen de alguien y una imagen del mismo fondo sin esa persona (figura 5.12). ¿Podría *sustraer* el fondo de la persona (es decir, averiguar en dónde son los colores exactamente iguales) y luego sustituir otro fondo? Por ejemplo, ¿de la Luna (figura 5.13)?

**Programa 46: sustraer el fondo y sustituirlo con uno nuevo**

```
def cambiarFondo(imag1, atras, nuevoFondo):
  for x in range(0,getWidth(imag1)):
    for y in range(0,getHeight(imag1)):
      p1Pixel = getPixel(imag1,x,y)
      atrasPixel = getPixel(atras,x,y)
      if (distance(getColor(p1Pixel),
        getColor(atrasPixel)) < 15.0):
        setColor(p1Pixel,getColor(getPixel(nuevoFondo,x,y)))
  return imag1
```

■

**FIGURA 5.12**
Una imagen de una niña (Katie) y el fondo sin ella.

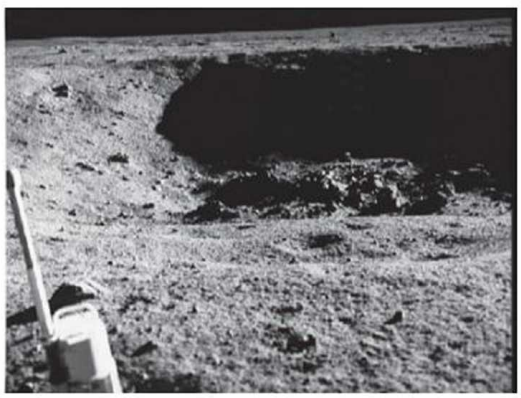


**FIGURA 5.13**
Un nuevo fondo: la Luna.

### Cómo funciona

La función `cambiarFondo` recibe una imagen (que contenga tanto el primer plano como el fondo), una imagen del fondo y un nuevo fondo. Para todos los píxeles en la imagen de entrada:

- Obtiene los píxeles que coinciden (mismas coordenadas) tanto de la imagen como del fondo.
- Compara las distancias entre los colores. Aquí usamos un valor de umbral de 15.0, pero puede probar con otros.
- Si la distancia es pequeña (menor que el valor de umbral), entonces supone que el pixel es parte del fondo. Toma el color del pixel en las mismas coordenadas en el *nuevo* fondo y establece el pixel en la imagen de entrada al nuevo color.

Puede hacer esto, pero el efecto no es tan bueno como quisiera (figura 5.14). El color de la blusa de nuestra hija estaba demasiado cerca del color de la pared, por lo que la Luna se mezcló en su blusa. Aunque la luz era tenue, hay una sombra que en definitiva está afectando aquí. Como la sombra no estaba en la imagen de fondo, el algoritmo la trata como parte del primer plano. Este resultado sugiere que la diferencia en el color entre el fondo y el primer plano es importante para que la sustracción de fondo funcione, ¡al igual que tener una buena iluminación!

Mark intentó lo mismo con una imagen de dos estudiantes enfrente de una pared tipo mosaico. Mark usó un trípode (algo de verdad imprescindible para alinear los píxeles), pero por desgracia dejó el enfoque automático activado, por lo que las dos imágenes originales (figura 5.15) en realidad no eran tan comparables. El cambio de fondo (con la escena de la jungla) tuvo muy poco efecto. Mark cambió el valor de umbral a 50 y por fin consiguió *algo* de intercambio (figura 5.16).

**FIGURA 5.14**
Katie en la luna.

**FIGURA 5.15**
Dos personas enfrente de una pared (*izquierda*) y una imagen de la pared (*derecha*).

**FIGURA 5.16**
Intercambio de una jungla por la pared, usando sustracción de fondo con un valor de umbral de 50.

No siempre basta con cambiar el umbral para mejorar las cosas. En definitiva se logra clasificar una mayor parte del primer plano como fondo. Pero para problemas como cuando el nuevo fondo traspasa la ropa debido a los colores que coinciden, el valor de umbral no ayuda mucho.

## 5.6 CHROMAKEY

Los meteorólogos de la televisión mueven las manos para mostrar un frente de tormenta acercándose a través de un mapa. La realidad es que están siendo filmados parados ante un fondo de un color fijo (por lo general azul o verde) y después ese color de fondo se sustituye digitalmente con píxeles del mapa deseado. A esto se le conoce como **chromakey**. Es más fácil sustituir un color conocido y no es tan sensible a los problemas de iluminación. Mark tomó la sábana azul de nuestro hijo, la colocó sobre el centro de entretenimiento familiar y

**FIGURA 5.17**
Mark enfrente de una sábana azul.

luego tomó una foto de él mismo parado enfrente de la sábana, usando un temporizador en una cámara (figura 5.17).

**Programa 47: Chromakey: sustitución de todo el azul con el nuevo fondo**

```
def chromakey(origen,fondo):
  # el origen debería tener algo enfrente de lo azul, fondo es el nuevo fondo
for x in range(0,getWidth(origen)):
    for y in range(0,getHeight(origen)):
      p = getPixel(origen,x,y)
      # Mi definición de azul: si el nivel de rojo
        + nivel de verde < nivel de azul
      if (getRed(p) + getGreen(p) < getBlue(p)):
      # Entonces, obtener el color en el mismo punto del nuevo
fondo
        setColor(p,getColor(getPixel(fondo,x,y)))
  return origen
```

■

## Cómo funciona

Aquí tomamos sólo una imagen de origen (que contiene el primer plano y el fondo) y un nuevo `fondo`. *¡Estas imágenes deben ser del mismo tamaño!* Mark usó la herramienta de imágenes de JES para idear una regla de lo que se consideraría como "azul" en este programa. No quería buscar igualdad o incluso una distancia al color (0, 0, 255), ya que sabía que una parte muy pequeña del azul sería *exactamente* un azul de intensidad completa. Descubrió que los píxeles que consideraba como azul tendían a incluir pequeños valores de rojo y verde y, de hecho, los valores de azul eran mayores que la suma del rojo y del verde. Entonces eso fue

lo que buscó en este programa. Siempre que el azul era mayor que el rojo y verde, él intercambiaba el nuevo color del pixel de fondo.

El efecto es en realidad bastante impactante (figura 5.18). Observe los "dobleces" en la superficie lunar. Lo verdaderamente impresionante es que este programa funciona para cualquier fondo que tenga el mismo tamaño que la imagen (figura 5.19).

Hay otra forma de escribir este código, que es más corta pero hace lo mismo. Usa las funciones `getX` y `getY` para averiguar las coordenadas. Esto conduce a una instrucción `setColor` un poco más complicada.

**FIGURA 5.18**
Mark en la Luna.

**FIGURA 5.19**
Mark en la jungla.

**Programa 48: Chromakey más corto**

```
def chromakey2(origen,fondo):
  for p in getPixels(origen):
    if (getRed(p)+getGreen(p) < getBlue(p)):
      setColor(p,getColor(getPixel(fondo,getX(p),getY(p))))
  return origen
```

■

En realidad no conviene utilizar la técnica chromakey con un color común como el rojo; para empezar, hay mucho de ese color en nuestro rostro. Mark intentó con las dos imágenes de la figura 5.20: una con el flash encendido y otra con el flash apagado. Cambió la prueba a `if getRed(p) > (getGreen(p) + getBlue(p)):`. La prueba sin flash fue terrible: el rostro del estudiante se mezcló con la jungla. La prueba con el flash fue mejor, pero aún puede verse el flash después del intercambio (figura 5.21). Con esto nos queda claro por qué los cineastas y los meteorólogos usan fondos azules o verdes para la técnica chromakey: hay menos traslape con los colores comunes, como los del rostro.

Los dispositivos y el software que implementan chromakey en el ámbito profesional usan un proceso algo distinto a éste. Nuestro algoritmo busca el color a sustituir y después realiza la sustitución. En la técnica chromakey profesional, se produce una *máscara*. Ésta es del mismo tamaño que la imagen original, en donde los píxeles que van a cambiarse son blancos en la máscara

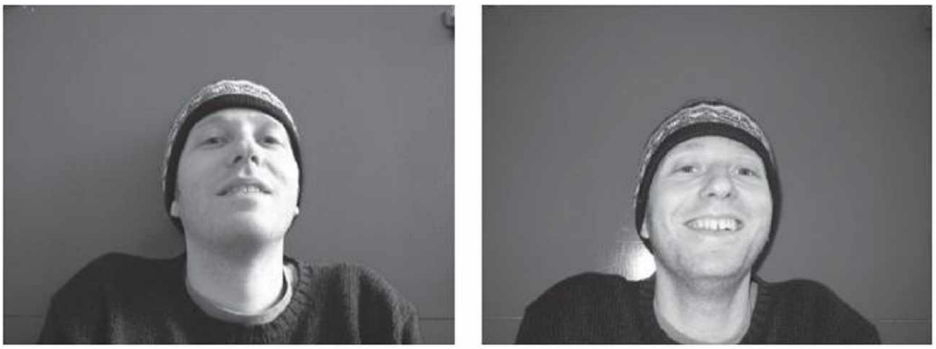


**FIGURA 5.20**
Un estudiante enfrente de un fondo rojo sin el flash (*izquierda*) y con el flash encendido (*derecha*).

**FIGURA 5.21**
Uso del programa chromakey con fondo rojo, flash apagado (*izquierda*) y flash encendido (*derecha*).

y los que no deben cambiarse son negros. La máscara se usa después para decidir cuáles píxeles van a cambiarse. Una ventaja en cuanto a usar una máscara es que separamos los procesos de (a) detectar cuáles píxeles van a cambiarse y (b) hacer los cambios en los píxeles. Al separar los procesos podemos mejorar cada uno de ellos y, por ende, mejorar todo el efecto en general.

## 5.7 DIBUJAR SOBRE IMÁGENES

Algunas veces es conveniente crear nuestras *propias* imágenes desde cero. Sabemos que es sólo cuestión de establecer los valores de los píxeles en lo que queramos, pero es difícil establecer valores de píxeles individuales para dibujar una línea, un círculo o incluso algunas letras. Una forma simple de dibujar imágenes es establecer los píxeles de manera apropiada. He aquí un ejemplo que crea líneas horizontales y verticales sobre Carolina, una estudiante de computación en Georgia Tech (figura 5.22). El programa funciona pidiendo que seleccione un archivo y después crea una imagen a partir de ese archivo. Luego llama a una función `lineasVerticales` que dibuja líneas verticales sobre la imagen, con 5 píxeles de separación. Después llama a una función `lineasHorizontales` que dibuja líneas horizontales en la imagen, con 5 píxeles de separación. Al final muestra y devuelve la imagen resultante.

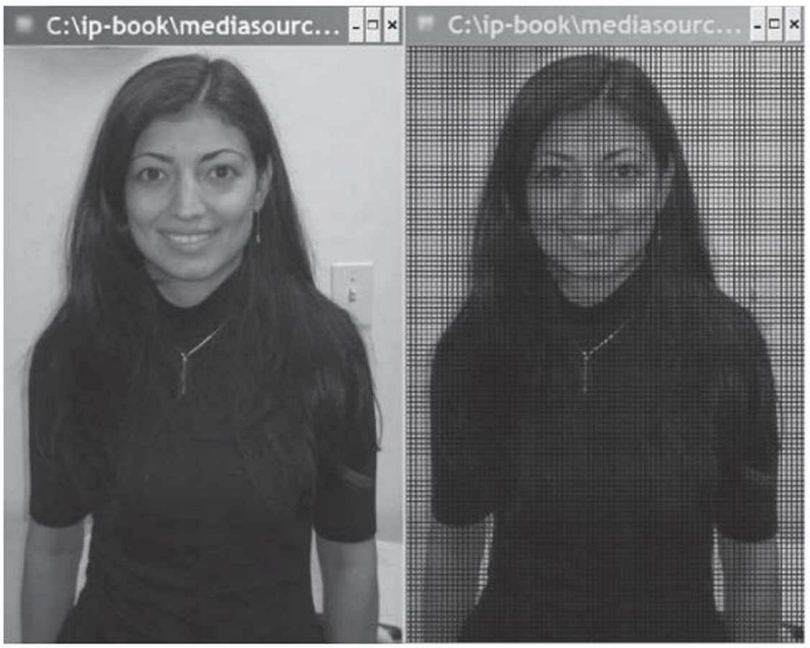



**FIGURA 5.22**
Carolina normal (*izquierda*) y con líneas agregadas (*derecha*).

**Programa 49: dibujo de líneas mediante el establecimiento de píxeles**

```
def ejemploLineas():
  img = makePicture(pickAFile())
  lineasVerticales(img)
  lineasHorizontales(img)
  show(img)
  return img
```

```
def lineasHorizontales(orig):
  for x in range(0,getHeight(orig),5):
    for y in range(0,getWidth(orig)):
      setColor(getPixel(orig,y,x),black)

def lineasVerticales(orig):
  for x in range(0,getWidth(orig),5):
    for y in range(0,getHeight(orig)):
      setColor(getPixel(orig,x,y),black)
```

■

Observe que este programa utiliza el nombre del color negro (`black`). Tal vez recuerde que JES predefine un grupo de colores para usted: negro (`black`), blanco (`white`), azul (`blue`), rojo (`red`), verde (`green`), gris (`gray`), gris claro (`lightGray`), gris oscuro (`darkGray`), amarillo (`yellow`), naranja (`orange`), rosa (`pink`), magenta (`magenta`) y cian (`cyan`). Puede usar cualquiera de estos colores de la misma manera que usaría otra función o comando.

Podemos imaginar dibujar cualquier cosa que queramos de esta manera, con sólo establecer los píxeles individuales a cualquier color deseado. Podríamos dibujar rectángulos o círculos con sólo averiguar qué píxeles necesitan ser de cierto color. Incluso podríamos dibujar letras: al establecer los píxeles apropiados a los colores apropiados, podríamos crear cualquier letra que quisiéramos. Aunque podríamos hacerlo, se requeriría demasiado trabajo para realizar todos los cálculos matemáticos para todas las distintas figuras y letras. Puesto que esto es lo que muchas personas necesitan, ya se integraron herramientas de dibujo básicas a las bibliotecas para usted.

### 5.7.1 Dibujar mediante comandos de dibujo

La mayoría de los lenguajes de programación modernos con bibliotecas de gráficos proveen funciones que nos permiten dibujar de manera directa una variedad de tipos distintos de figuras en las imágenes, además de dibujar texto directamente en una imagen. He aquí algunas de esas funciones:

- `addText(imagen,x,y,cadena)` coloca la cadena empezando en la posición $(x, y)$ en la imagen.
- `addLine(imagen,x1,y1,x2,y2)` dibuja una línea desde la posición $(x1, y1)$ hasta $(x2, y2)$.
- `addRect(imagen,x1,y1,anchura,altura)` dibuja un rectángulo con líneas negras, con la esquina superior izquierda en $(x1, y1)$, una `anchura` y una `altura`.
- `addRectFilled(imagen,x1,y1,anchura,altura,color)` dibuja un rectángulo relleno del color que usted especifique, con la esquina superior izquierda en $(x1, y1)$, una `anchura` y una `altura`.

Podemos usar estos comandos para agregar cosas a imágenes existentes. ¿Qué tal si apareciera una misteriosa caja roja en la playa? Podemos hacer que aparezca esa escena con estos tipos de comandos (figura 5.23).

**Programa 50: agregar una caja a una playa**

```
def agregarUnaCaja():
  playa = makePicture(getMediaPath("beach.jpg"))
  addRectFilled(playa,150,300,50,50,red)
  show(playa)
  return playa
```

■

**FIGURA 5.23**
Una caja arrastrada a la orilla de la playa.

A continuación veremos otro ejemplo del uso de estos comandos de dibujo (figura 5.24).

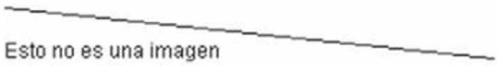


**FIGURA 5.24**
Una imagen dibujada muy pequeña.

**Programa 51: un ejemplo de uso de comandos de dibujo**

```
def imagenChica():
  lienzo=makePicture(getMediaPath("640x480.jpg"))
  addText(lienzo,10,50,"Esto no es una imagen")
  addLine(lienzo,10,20,300,50)
  addRectFilled(lienzo,0,200,300,500,yellow)
  addRect(lienzo,10,210,290,490)
  return lienzo
```

■

### 5.7.2 Representaciones vectoriales y de mapas de bits

Considere lo siguiente: ¿Cuál de éstos es más chico, la imagen (figura 5.24) o el programa? La imagen en el disco ocupa cerca de 15 kilobytes (un *kilobyte* equivale a 1024 bytes). El programa 51 tiene menos de 100 bytes. Pero para muchos fines son *equivalentes*. ¿Qué pasaría si sólo guardara el programa y no los píxeles? Esto es de lo que trata una **representación vectorial** para los gráficos.

Las representaciones gráficas basadas en vectores son programas ejecutables que generan los colores cuando se desean. Las representaciones basadas en vectores se utilizan en PostScript, Flash y AutoCAD. Cuando hacemos un cambio en una imagen en Flash o AutoCAD, en realidad hacemos un cambio en la representación subyacente; en esencia está modificando el programa, al igual que como en el programa 51. Después el programa se ejecuta de nuevo para hacer que aparezca la imagen. Pero gracias a la Ley de Moore, la ejecución y la nueva visualización ocurren tan rápido que parece como si hubiera modificado la imagen.

Los lenguajes de definición de fuentes como PostScript y TrueType definen programas en miniatura (o ecuaciones) por cada una de las letras o símbolos. Si desea la letra o símbolo en un tamaño específico, se ejecuta el programa para averiguar qué píxeles deberían establecerse en ciertos valores (algunos especifican más de un color para crear el efecto de curvas más suaves). Debido a que los programas se escriben para manejar el tamaño de fuente deseado como entrada, pueden generarse las letras y símbolos en cualquier tamaño.

Por otra parte, las representaciones gráficas de mapas de bits almacenan cada pixel individual o una representación comprimida de los píxeles. Los formatos como BMP, GIF y JPEG son en esencia representaciones de mapas de bits. GIF y JPEG son representaciones comprimidas: no representan a todos y cada uno de los píxeles con 24 bits. En vez de ello representan la misma información, sólo que con menos bits.

¿Qué significa la **compresión**? Nos indica que se han usado varias técnicas para hacer el archivo más pequeño. Algunas técnicas de compresión se conocen como **compresión con pérdida**: se pierde cierto detalle, pero con suerte será el menos importante (tal vez inclusive sea invisible al ojo o el oído humano). Otras técnicas conocidas como **compresión sin pérdida** no pierden detalle y aun así comprimen el archivo. Una de las técnicas sin pérdida es la **codificación de longitud de tirada** (RLE).

Imagine que tiene una larga línea de píxeles amarillos en una imagen, rodeados por algunos píxeles azules. Algo así como esto:

B B Y Y Y Y Y Y Y Y Y B B

¿Qué pasaría si codificara esto, no como una larga línea de píxeles, sino como algo así?:

B B 9 Y B B

En otras palabras, usted codifica "azul, azul, después nueve amarillos, luego azul y azul". Como cada uno de los píxeles amarillos ocupa 24 bits (3 bytes para rojo, verde y azul) y

para registrar la palabra "nueve" se ocupa un solo byte, hay un enorme ahorro. Decimos que codificamos la *longitud* de la *tirada* de amarillos; por ende, codificamos la longitud de la tirada. Éste es sólo uno de los métodos de compresión que se utiliza para hacer más pequeñas las imágenes.

Existen varios beneficios para las representaciones basadas en vectores en comparación con las representaciones de mapas de bits. Si puede representar la imagen que desea enviar (por decir, a través de Internet) mediante una representación basada en vectores, es mucho más pequeña que enviar todos los píxeles; en cierto sentido, la notación vectorial ya está comprimida. En esencia, usted envía las *instrucciones* que indican cómo hacer la imagen, en vez de enviar la imagen en sí. Sin embargo, para imágenes muy complejas las instrucciones pueden ser tan extensas como la imagen (imagine enviar todas las indicaciones sobre cómo pintar la *Mona Lisa*), por lo que no hay un beneficio. Pero cuando las imágenes son lo bastante simples, con representaciones como las que se usan en Flash se pueden obtener tiempos de envío y descarga más rápidos que si se enviaran las mismas imágenes JPEG.

El verdadero beneficio de las notaciones basadas en vectores se obtiene cuando queremos modificar la imagen. Digamos que está trabajando en un dibujo arquitectónico y extiende una línea en su herramienta de dibujo. Si su herramienta sólo trabaja con imágenes de mapas de bits (lo que se conoce algunas veces como **herramienta para pintar**), entonces todo lo que tiene son más píxeles en la pantalla que son adyacentes a los otros píxeles en la pantalla que representan a la línea. No hay nada en la computadora que indique que todos esos píxeles representan una línea de cierto tipo; sólo son píxeles. Pero si su herramienta de dibujo trabaja con representaciones basadas en vectores (lo que se conoce algunas veces como **herramienta de dibujo**), entonces extender una línea significa que está modificando una representación subyacente de una línea.

¿Por qué es eso importante? La representación subyacente es en realidad una *especificación* del dibujo y puede usarse siempre que se requiera una especificación. Imagine tomar el dibujo de una pieza, para después operar las máquinas de corte y estampado con base en ese dibujo. Esto ocurre con regularidad en muchos talleres y es posible gracias a que el dibujo no consta sólo de píxeles: es una especificación de las líneas y sus relaciones, que a su vez pueden escalarse y usarse para determinar el comportamiento de las máquinas.

Tal vez se pregunte, "pero ¿cómo podríamos *cambiar* el programa? ¿Podemos escribir un programa que en esencia vuelva a escribir el programa o partes del mismo?". Seguro que podemos, y lo haremos en el capítulo 10.

## 5.8 SELECCIONAR SIN VOLVER A PROBAR

Considere el programa 41 que vimos anteriormente en el capítulo:

```
def posterizarGris(imag):
  for p in getPixels(imag):
    r = getRed(p)
    g = getGreen(p)
    b = getBlue(p)
    luminancia = (r+g+b)/3
    if luminancia < 64:
      setColor(p,black)
    if luminancia >= 64:
      setColor(p,white)
```

Si la `luminancia` no es menor a 64, *debe* ser mayor o igual a 64. El hecho de evaluar la `luminancia` de nuevo parece un poco ineficiente. Y, sin embargo, hicimos *justo* eso. En

esencia queremos usar una bifurcación aquí: establecer el pixel al color blanco, o a negro, y no hay ninguna otra opción.

Como la bifurcación de decisiones ocurre con tanta frecuencia, Python (y muchos otros lenguajes) le proporciona una manera de hacer justo este tipo de elección, con un `else`. Esta instrucción viene después de un `if` y antes de un bloque. El *bloque else* sólo ocurre si la evaluación del `if` es *falsa*.

Podemos volver a escribir el programa de esta forma:

**Programa 52: posterización en dos niveles de gris, con `else`**

```
def posterizarGris(imag):
  for p in getPixels(imag):
    r = getRed(p)
    g = getGreen(p)
    b = getBlue(p)
    luminancia = (r+g+b)/3
    if luminancia < 64:
      setColor(p,black)
    else:
      setColor(p,white)
```

■

Ahora bien, algunas veces es conveniente tomar una bifurcación con tres opciones. Si algo es verdadero, queremos realizar cierta acción y luego elegir una de otras dos rutas. Para ello decimos: *Si* (`if`) ocurre esto, entonces hacer una cosa, *de lo contrario si* (`else if`) esta otra cosa es verdadera, tomar la ruta uno, *de lo contrario* (`else`) tomar la ruta dos. Esta combinación `else if` es tan común que podemos abreviarla como `elif`.

¿Le pareció confuso el párrafo anterior? Ése es el problema de `else` y `elif`. Compare los dos programas anteriores de esta sección. Ambos *hacen* exactamente lo mismo. El segundo tiene *implícita* la evaluación de `luminancia >= 64` en el `else`, pero no es explícita. ¿Acaso eso es algo malo? Resulta ser que es malo para las computadoras, pero excelente para los humanos.

Un `else` evita que la computadora realice una segunda evaluación, lo cual es un poco más eficiente. Pero al hacer la instrucción *explícita* logramos mejorar la legibilidad. Por lo menos en un estudio[1], al tener esa segunda evaluación explícita es posible mejorar la habilidad de los nuevos programadores para comprender sus programas ¡hasta *diez veces*!

Recuerde: los programas son para las personas, no para las computadoras. Primero escriba sus programas para que sean comprensibles. Después preocúpese por la eficiencia.

## 5.9 PROGRAMAS PARA ESPECIFICAR EL PROCESO DE DIBUJO

Es posible usar funciones de dibujo como éstas para crear imágenes que se especifiquen con exactitud: cosas que podrían ser demasiado difíciles de hacer a mano. Por ejemplo, considere la figura 5.25.

Esta imagen es una representación de una famosa ilusión óptica y no es tan efectiva como la verdadera, pero es fácil comprender cómo funciona esta versión. Nuestros ojos nos indican que la mitad izquierda de la imagen es más clara que la mitad derecha, aun cuando las cuartas partes de los extremos tienen exactamente la misma sombra de gris. Sólo las dos

[1]Sime, M., Green, T., y Guest, D. (1976). Scope marking in computer conditionals: A psychological evaluation. *International Journal of Man-Machine Studies*, 9, 107-118.

**FIGURA 5.25**
Un efecto de escala de grises programado.

cuartas partes de en medio son las que cambiaron realmente. El efecto se origina por el límite tan pronunciado entre las cuartas partes de en medio, en donde la cuarta parte a la izquierda del centro pasa (de izquierda a derecha) de gris a blanco, y la cuarta parte a la derecha del centro pasa de negro a gris (de izquierda a derecha).

La imagen en la figura 5.25 es una imagen definida y creada con cuidado. Sería muy difícil hacerla con lápiz y papel. Sería posible con algo como Photoshop, pero no sería fácil. No obstante, si usamos las funciones de gráficos de este capítulo, podremos especificar con facilidad y exactitud cómo debe ser la imagen.

**Programa 53: dibujar el efecto del color gris**

```
def efectoGris():
  archivo = getMediaPath("640x480.jpg")
  imag = makePicture(archivo)
  # Primero, 100 columnas de gris 100
  gris = makeColor(100,100,100)
  for x in range(0,100):
    for y in range(0,100):
      setColor(getPixel(imag,x,y),gris)
  # Segundo, 100 columnas de gris cada vez más oscuro
  nivelGris = 100
  for x in range(100,200):
    gris = makeColor(nivelGris, nivelGris, nivelGris)
    for y in range(0,100):
      setColor(getPixel(imag,x,y),gris)
    nivelGris = nivelGris + 1
  # Tercero, 100 columnas de nivel de gris cada vez mayor, desde 0
  nivelGris = 0
  for x in range(200,300):
    gris = makeColor(nivelGris, nivelGris, nivelGris)
    for y in range(0,100):
      setColor(getPixel(imag,x,y),gris)
    nivelGris = nivelGris + 1
  # Por último, 100 columnas de gris 100
  gris = makeColor(100,100,100)
  for x in range(300,400):
    for y in range(0,100):
      setColor(getPixel(imag,x,y),gris)
  return imag
```

■

Las funciones gráficas son muy buenas con dibujos que se repiten en donde las posiciones de las líneas y figuras además de la selección de colores se pueden realizar mediante relaciones matemáticas. Observe algo interesante en el siguiente programa. Usamos un *intervalo* negativo en la función `range`. La expresión `range(25,0,-1)` cuenta desde 25 y desciende hasta 1, por –1.

**Programa 54: dibuja la imagen en la figura 5.26**

```
def imagenDivertida():
  lienzo = makePicture(getMediaPath("640x480.jpg"))
  for indice in range(25,0,-1):
    color = makeColor(indice*10,indice*5,indice)
    addRectFilled(lienzo,0,0,indice*10,indice*10,color)
  show(lienzo)
  return lienzo
```

■

**Programa 55: dibuja la imagen en la figura 5.27**

```
def imagenDivertida2():
  lienzo = makePicture(getMediaPath("640x480.jpg"))
  for indice in range(25,0,-1):
    addRect(lienzo,indice,indice,indice*3,indice*4)
    addRect(lienzo,100+indice*4,100+indice*3,indice*8,indice*10)
  show(lienzo)
  return(lienzo)
```

■

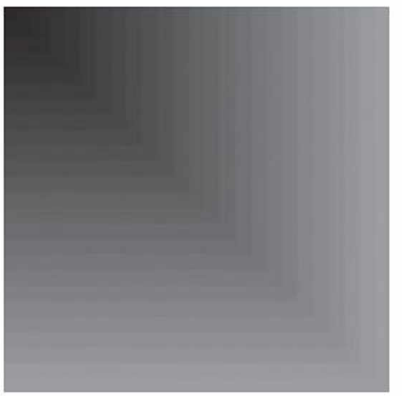


**FIGURA 5.26**
Imagen de rectángulos de colores anidados.

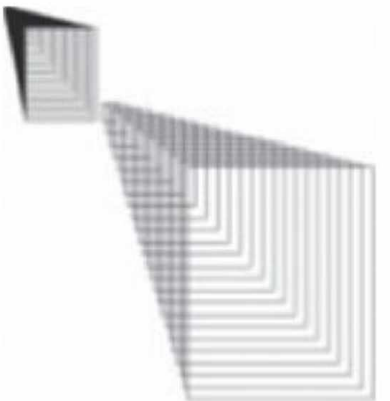


**FIGURA 5.27**
Imagen de rectángulos blancos anidados.

### 5.9.1 ¿Por qué escribimos programas?

¿Por qué escribir programas, en especial los que dibujan imágenes? ¿Podríamos dibujar imágenes como éstas en Photoshop o Visio? Claro que sí podemos, pero tendríamos que saber *cómo*, y este conocimiento no se adquiere con facilidad. ¿Podríamos *enseñarle* cómo hacer esto en Photoshop? Tal vez, pero se requeriría mucho esfuerzo: Photoshop no es simple.

Pero si le *proporcionamos* estos programas, podrá crear la imagen cada vez que lo desee. Además, al proporcionarle el programa le estoy dando la definición *exacta* que usted puede modificar por su cuenta.

**Idea de ciencias computacionales: escribimos programas para encapsular y comunicar un proceso**

La razón por la que escribimos programas es para especificar con exactitud un proceso y comunicarlo a otras personas.

■

Imagine que tiene que comunicar cierto proceso. No tiene que ser de dibujo; suponga que es un proceso financiero (como el que podría hacer en una hoja de cálculo o en un programa como Quicken) o algo que puede hacer con texto (como redactar texto para un libro o folleto). Si puede hacer algo con la mano, debería hacerlo. Si necesita *enseñar* a alguien cómo hacerlo, considere la opción de escribir un programa para hacerlo. Si necesita explicar cómo hacerlo a *muchas* personas, en definitiva le conviene usar un programa. Si desea que muchas personas puedan realizar el proceso por su cuenta, sin que alguien tenga que enseñarles algo primero, en definitiva debe escribir un programa y dárselo a esas personas.

## RESUMEN DE PROGRAMACIÓN

He aquí las funciones que presentamos en este capítulo:

| | |
|---|---|
| `addText(imag,x,y,cadena)` | Coloca la cadena a partir de la posición ($x$, $y$) en la imagen. |
| `addLine(imagen,x1,y1,x2,y2)` | Dibuja una línea desde la posición ($x1$, $y1$) hasta ($x2$, $y2$). |
| `addRect(imag,x1,y1,w,h)` | Dibuja un rectángulo con líneas negras, con la esquina superior izquierda en ($x1$, $y1$), una anchura de $w$ y una altura de $h$. |
| `addRectFilled(imag,x1,y1,w, h,color)` | Dibuja un rectángulo relleno con el color que usted elija, con la esquina superior izquierda en ($x1$, $y1$), una anchura de $w$ y una altura de $h$. |

## PROBLEMAS

5.1 Escriba una función llamada `cambiarColor` que reciba como entrada una imagen $y$ una cantidad para aumentar o disminuir la intensidad de un color, *además* del número 1 (para rojo), 2 (para verde) o 3 (para azul). La cantidad será un número entre –.99 y .99.

- `cambiarColor(imag,-.10,1)` deberá reducir la cantidad de rojo en la imagen en un 10%.

- `cambiarColor(imag,.30,2)` deberá incrementar la cantidad de verde en la imagen en un 30%.
- `cambiarColor(imag,0,3)` no deberá hacer ninguna modificación en la cantidad de azul (o rojo, o verde) en la imagen.

5.2 ¿Cuál de los siguientes programas recibe una imagen y elimina todo el azul de cada pixel que ya tenga un valor de azul mayor a 100?

1. Sólo A
2. Sólo D
3. B y C
4. C y D
5. Ninguno
6. Todos

¿Qué hacen los demás?

A.
```
def azulCien(imagen):
   for x in range(0,100):
       for y in range(0,100):
           pixel = getPixel(imagen,x,y)
           setBlue(pixel,100)
```

B.
```
def quitarAzul(imagen):
  for p in getPixels(imagen):
      if getBlue(p) > 0:
      setBlue(p,100)
```

C.
```
def sinAzul(imagen):
  azul = makeColor(0,0,100)
  for p in getPixels(imagen):
      color = getColor(p)
      if distance(color,azul) > 100:
          setBlue(p,0)
```

D.
```
def adiosAzul(imagen):
   for p in getPixels(imagen):
   if getBlue(p) > 100:
          setBlue(p,0)
```

5.3 Vuelva a escribir el programa 40, usando esta función auxiliar para hacerlo más corto.

```
def elegirValorPosterizacion(actual):
  if (actual < 64):
     return 31
  if (actual > 63 and actual < 128):
     return 95
  if (actual > 127 and actual < 192):
     return 159
  if (actual > 191 and actual < 256):
     return 223
```

5.4 ¿Qué se imprimirá con esta función si ejecutara cada una de las siguientes líneas?:
(a) `pruebaMe (1, 2, 3)`
(b) `pruebaMe (3, 2, 1)`
(c) `pruebaMe (5, 75, 20)`

```
def pruebaMe(p,q,r):
  if q > 50:
    print r
  valor = 10
  for i in range(0,p):
    print "Hola"
    valor = valor - 1
  print valor
  print r
```

5.5 Empiece con una imagen de alguien que conozca y haga ciertos cambios de colores específicos en ella:

- Haga los dientes morados.
- Haga los ojos rojos.
- Haga el cabello naranja.

Desde luego que, si los dientes de su amigo(a) ya son morados, los ojos rojos o el cabello naranja, elija mejor otro color final.

5.6 Escriba un programa llamado `revisarLuminancia` que reciba como entrada valores de rojo, verde y azul, y calcule la luminancia usando el promedio ponderado (como se indica a continuación). Pero después imprima una advertencia para el usuario con base en la luminancia calculada:

- Si la luminancia es menor de 10, "Esto va a estar demasiado oscuro".
- Si la luminancia está entre 50 y 200, "Parece un buen rango".
- Si es mayor a 250, "¡Esto va a estar casi todo blanco!".

5.7 Intente usar la técnica chromakey en un rango: extraiga algo de su fondo, en donde ese algo se encuentre sólo en una parte de la imagen. Por ejemplo, coloque una aureola alrededor de la cabeza de alguien, pero no modifique el resto del cuerpo.

5.8 Escriba una función para mezclar dos imágenes, empezando con la tercera parte superior de la primera imagen y luego mezcle las dos en la tercera parte de en medio, para después mostrar la última tercera parte de la segunda imagen. Esto funciona mejor si las dos imágenes son del mismo tamaño.

5.9 Dibuje una línea negra en una imagen en donde la diferencia entre los colores de un pixel y el que está a la derecha sea mayor que cierta cantidad que se pase como parámetro.

5.10 Escriba una función para intercalar dos imágenes. Reciba los primeros 20 píxeles de la primera imagen, después 20 píxeles de la segunda imagen y luego los siguientes 20 píxeles de la primera imagen, luego los siguientes 20 píxeles de la segunda imagen y así, en lo sucesivo, hasta que se hayan usado todos los píxeles.

5.11 Escriba una función que tome el 25% de una imagen y lo mezcle con el 75% de otra.

5.12 Vuelva a escribir el programa 41 para usar un `if` con un `else`.

5.13 Cree un cartel de cine, dibujando texto sobre una imagen.

5.14 Cree una tira cómica, colocando de tres a cuatro imágenes una seguida de la otra en sentido horizontal, y luego agregue texto.

5.15 Use las funciones de dibujo para dibujar un tiro al blanco.

5.16 Use las herramientas de dibujo que presentamos en este capítulo para dibujar una casa; use como referencia la simple casa de niños con una puerta con dos ventanas, paredes y un techo.

5.17 Dibuje líneas horizontales y verticales en una imagen con 10 píxeles entre las líneas.

5.18 Dibuje líneas diagonales sobre una imagen, desde la parte superior izquierda hasta la parte inferior derecha.

5.19 Dibuje líneas diagonales sobre una imagen, desde la parte superior derecha hasta la parte inferior izquierda.

5.20 ¿Qué es una imagen basada en vectores? ¿Qué diferencia tiene de una imagen de mapa de bits? ¿Cuándo es mejor usar una imagen basada en vectores?

5.21 Dibuje una casa en la playa en la imagen en donde colocamos antes una misteriosa caja.

5.22 Escriba una función para dibujar un rostro simple con ojos y una boca.

5.23 ¿Por qué los cineastas usan una pantalla verde o azul para efectos especiales en vez de una roja?

5.24 Consiga una cartulina verde y tome una fotografía a un amigo o amiga enfrente de ella. Ahora use la técnica chromakey para ambientar la fotografía en la jungla o en París.

5.25 Ahora use su función de la casa para dibujar una ciudad con docenas de casas de distintos tamaños. Tal vez quiera modificar su función de la casa para dibujar en una coordenada de entrada y después cambiar la coordenada en la que se va a dibujar cada casa.

5.26 Dibuje un arcoíris. Use lo que sabe acerca de los colores, los píxeles y el dibujo de operaciones para dibujar un arcoíris. ¿Es esto más fácil de hacer con nuestras funciones de dibujo, o manipulando los píxeles individuales? ¿Por qué?

## PARA PROFUNDIZAR

John Maeda está en el Grupo de estética y computación en el MIT Media Lab. Revise su entorno de procesamiento (`http://www.processing.org`) para el desarrollo de arte interactivo, procesamiento de video en vivo y visualización de datos. El procesamiento le permite realizar algunos de los mismos efectos que vimos en este capítulo.

PARTE 2

# SONIDO


Capítulo 6 Modificación de sonidos mediante ciclos

Capítulo 7 Modificación de muestras en un rango

Capítulo 8 Creación de sonidos mediante la combinación de piezas

Capítulo 9 Creación de programas más grandes

CAPÍTULO 6

# Modificación de sonidos mediante ciclos


6.1 CÓMO SE CODIFICA EL SONIDO

6.2 MANIPULACIÓN DE SONIDOS

6.3 CAMBIAR EL VOLUMEN DE LOS SONIDOS

6.4 NORMALIZACIÓN DE SONIDOS


### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Comprender cómo digitalizamos los sonidos, además de las limitaciones del oído humano que nos permiten digitalizar los sonidos.
- Usar el teorema de Nyquist para determinar la velocidad de muestreo necesaria para digitalizar un sonido deseado.
- Manipular el volumen.
- Crear (y evitar) recortes.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Comprender y usar los arreglos como una estructura de datos.
- Usar la fórmula de que *n* bits producen $2^n$ patrones posibles para averiguar el número de bits necesarios para guardar valores.
- Usar el objeto sonido.
- Depurar programas de sonido.
- Usar la iteración (en los ciclos `for`) para manipular sonidos.
- Usar el alcance para comprender cuándo está disponible una variable para usarla.

## 6.1 CÓMO SE CODIFICA EL SONIDO

Hay dos partes para comprender la forma en que el sonido se codifica y manipula.

- Primero, ¿cuál es la física del sonido? ¿cómo es que escuchamos una variedad de sonidos?
- Después, ¿cómo podemos entonces asignar los sonidos a números en una computadora?

### 6.1.1 La física del sonido

En el sentido físico, los sonidos son ondas de presión de aire. Cuando algo hace un sonido, genera ondulaciones en el aire justo como las piedras o gotas de lluvia que caen en un estanque

**FIGURA 6.1**
Las gotas de lluvia provocan ondas en la superficie del agua, así como el sonido provoca ondas en el aire.

producen ondulaciones en la superficie del agua (vea la figura 6.1). Cada gota provoca que una onda de presión pase sobre la superficie del agua, lo cual genera elevaciones visibles en el agua además de depresiones menos visibles, pero igual de grandes. Las elevaciones son aumentos en la presión y las depresiones son reducciones en la presión. Algunas de las ondas que vemos en realidad surgen debido a las *combinaciones* de ondas: algunas olas son las sumas e interacciones de otras olas.

En el aire, a estos aumentos de presión les llamamos *compresiones*, y a las reducciones en presión les decimos *rarefacciones*. Son estas compresiones y rarefacciones las que nos permiten escuchar sonidos. La forma de las ondas, su *frecuencia* y su *amplitud* tienen impacto en lo que percibimos en el sonido.

El sonido más simple del mundo es una **onda senoidal** (figura 6.2). En este tipo de onda, las compresiones y rarefacciones llegan con igual tamaño y regularidad. En una onda senoidal,

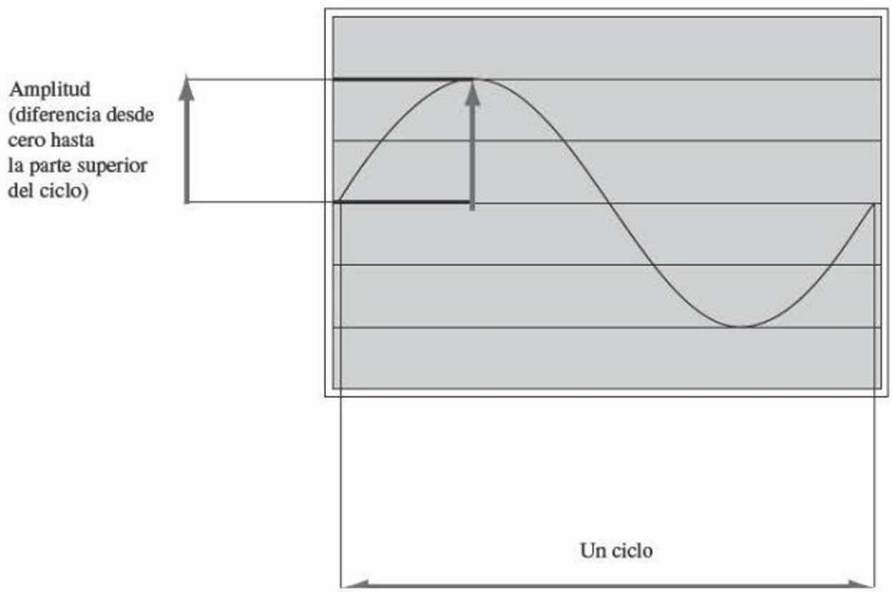



**FIGURA 6.2**
Un ciclo del sonido más simple: una onda senoidal.

a una compresión más una rarefacción se le conoce como **ciclo**. En cierto punto en el ciclo, tiene que haber un punto en donde haya cero presión, justo entre la compresión y la rarefacción. A la distancia desde el punto cero hasta el punto en donde hay mayor presión (o menor presión) se le conoce como **amplitud**.

En general, la amplitud es el factor más importante en nuestra percepción del *volumen*: si la amplitud aumenta, por lo general percibimos el sonido como más fuerte. Al percibir un aumento en el volumen, decimos que estamos percibiendo un aumento en la *intensidad* del sonido.

La percepción humana del sonido no es una asignación directa de la realidad física. Al estudio de la percepción humana del sonido se le conoce como *psicoacústica*. Uno de los hechos extraños sobre la psicoacústica es que la mayoría de nuestras percepciones del sonido se relacionan en forma *logarítmica* al fenómeno real.

Para medir el cambio en intensidad usamos **decibeles** (dB). Tal vez ésta sea la unidad que se asocia con más frecuencia con el volumen. Un decibel es una medida logarítmica, por lo que coincide con la forma en la que percibimos el volumen. Siempre es una relación, una comparación de dos valores. $10 * \log_{10}(I_1/I_2)$ es el cambio en intensidad en decibeles entre dos intensidades, $I_1$ y $I_2$. Si se miden dos amplitudes bajo las mismas condiciones, podemos expresar la misma definición como amplitudes: $20 * \log_{10}(A_1/A_2)$. Si $A_2 = 2 * A_1$ (es decir, la amplitud se duplica), la diferencia es de aproximadamente 6 dB.

Cuando se usa el decibel como una medición absoluta, es en referencia al umbral de audibilidad al *nivel de presión de sonido* (SPL): 0 dB SPL. El habla normal tiene una intensidad aproximada de 60 dB SPL. Los gritos tienen cerca de 80 dB SPL.

La **frecuencia** indica qué tan seguido ocurre un ciclo. Si el ciclo es corto, entonces puede haber muchos de ellos por segundo. Si el ciclo es largo, entonces hay menos de ellos. A medida que aumenta la frecuencia, percibimos que aumenta el **tono**. Medimos la frecuencia en *ciclos por segundo* (cps) o *Hertz* (Hz).

Todos los sonidos son periódicos: siempre hay cierto patrón de rarefacción y compresión que conduce a los ciclos. En una onda senoidal, la noción de ciclo es sencilla. En ondas naturales, no es tan claro ver en dónde se repite un patrón. Incluso en las olas en un estanque, las ondas no son tan regulares como podríamos pensar. El tiempo entre picos en las ondas no es siempre el mismo: varía. Esto significa que en un ciclo puede haber varios picos y valles hasta que se repita.

Los humanos escuchan entre los 2 y los 20 000Hz (o 20 kilohertz, que se abrevia como 20 kHz). De nuevo, al igual que con las amplitudes, es un rango enorme. Para que tenga una idea de dónde se ajusta la música en ese espectro, la nota La por encima del Do central es de 440 Hz en la afinación tradicional de *temperamento igual* (figura 6.3).

**FIGURA 6.3**
La nota La por encima del Do central es de 440 Hz.

Como la intensidad, nuestra percepción del tono es casi exactamente proporcional al logaritmo de la frecuencia. No percibimos diferencias absolutas en el tono, sino la *relación* de las frecuencias. Si escuchara un sonido de 100 Hz seguido de un sonido de 200 Hz, percibiría el mismo cambio en el tono (o *intervalo de tono*) como un cambio de 1000 Hz a 2000 Hz. Sin duda, una diferencia de 100 Hz es mucho más pequeña que un cambio de 1000 Hz, pero la percibimos como lo mismo.

En la afinación estándar, la relación en la frecuencia entre las mismas notas en octavas adyacentes es de 2 : 1. La frecuencia se duplica en cada octava. En párrafos anteriores mencionamos que la nota La por encima del Do central es de 440 Hz. Por consiguiente, la siguiente nota de La en la escala es de 880 Hz.

La forma en que pensamos sobre la música depende de nuestros estándares culturales, pero hay algunos estándares universales. Entre ellos se encuentra el uso de los intervalos de tono (por ejemplo, la relación entre las notas Do y Re permanece igual en cada octava), la relación constante entre octavas, y la existencia de cuatro a siete tonos principales (sin considerar los sostenidos y bemoles aquí) en una octava.

¿Qué hace a la experiencia de un sonido distinta de otra? ¿Por qué una flauta que reproduce una nota suena *tan* diferente de una trompeta o un clarinete que reproduce la misma nota? Todavía no entendemos todo sobre la psicoacústica y las propiedades físicas que influyen en nuestra percepción del sonido, pero he aquí algunos de los factores que nos conducen a percibir sonidos diferentes (en especial los instrumentos musicales) como distintos:

- Los sonidos reales casi nunca son ondas de sonido de una sola frecuencia. La mayoría de los sonidos naturales tienen *varias* frecuencias en ellos, a menudo con diferentes amplitudes. Estas frecuencias adicionales se conocen algunas veces como *armónicos*. Por ejemplo, cuando un piano reproduce la nota Do, parte de la riqueza del tono es que las notas Mi y Sol *también* están en el sonido, pero con amplitudes menores. Los distintos instrumentos tienen armónicos diferentes en sus notas. El tono central, el que tratamos de reproducir, se denomina *fundamental*.
- Los sonidos de los instrumentos no son continuos con respecto a la amplitud y la frecuencia. Algunos llegan lentamente a la frecuencia y amplitud de destino (como los instrumentos de viento), mientras que otros lo hacen con mucha rapidez y luego el volumen se desvanece mientras la frecuencia permanece muy constante (como un piano).
- No todas las ondas de sonido se representan bien mediante ondas senoidales. Los sonidos reales tienen bultos graciosos y bordes pronunciados. Nuestros oídos pueden detectarlos, cuando menos en las primeras ondas. Podemos hacer un trabajo razonable de sintetizar con ondas senoidales, pero algunas veces los sintetizadores usan otros tipos de formas de onda para obtener distintos tipos de sonidos (figura 6.4).

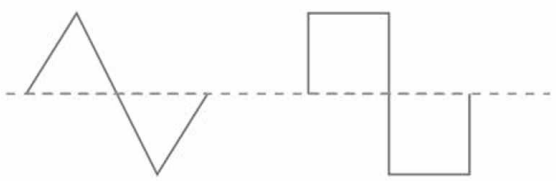


**FIGURA 6.4**
Algunos sintetizadores usan ondas triangulares (o de *diente de sierra*) o cuadradas.

### 6.1.2 Exploración de la apariencia de los sonidos

En `http://www.mediacomputation.org` encontrará la aplicación MediaTools con documentación sobre cómo iniciarla. Esta aplicación contiene herramientas para sonido, gráficos y video. Las herramientas de sonido le permiten observar sonidos a medida que entran en el micrófono de su computadora, para que tenga una idea de cómo se ven los sonidos fuertes y suaves, y cómo se ven los sonidos con tonos más altos y más bajos.

También encontrará un menú MEDIA TOOLS en JES. Las herramientas de medios de JES también le permiten inspeccionar sonidos e imágenes, pero no puede analizar los sonidos en *tiempo real* (al momento en que llegan al micrófono de su computadora). En la aplicación MediaTools puede ver los sonidos en tiempo real.

El editor de sonidos de la aplicación MediaTools se ve como en la figura 6.5. Puede grabar sonidos, abrir archivos WAV en su disco y ver los sonidos en varias formas (¡suponiendo que tenga un micrófono en su computadora!).

Para ver los sonidos, haga clic en el botón RECORD VIEWER (VISOR DE GRABACIÓN) y luego en el botón RECORD (GRABAR) (presione el botón STOP [DETENER] para detener la grabación). Hay tres tipos de vistas que podemos crear con respecto al sonido.

La primera es la **vista de señal** (figura 6.6). En esta vista, usted analiza el sonido puro; cada aumento en la presión del aire produce una elevación en el gráfico y cada reducción en la presión del sonido produce una caída en el gráfico. Observe la rapidez con la que cambia la onda. Pruebe con algunos sonidos más suaves y más fuertes, de modo que pueda ver cómo cambia su apariencia. Siempre podrá retroceder hasta la vista de señal desde otra vista, haciendo clic en el botón SIGNAL (SEÑAL).

**FIGURA 6.5**
Herramienta principal del editor de sonido.

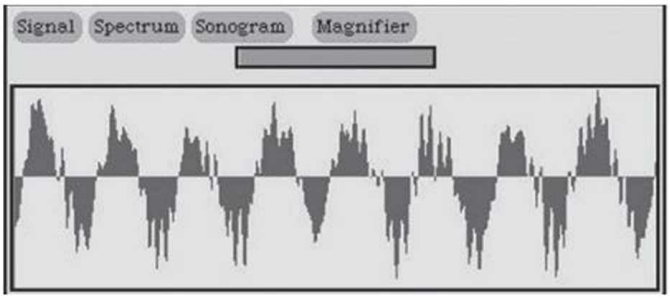



**FIGURA 6.6**
Vista de la señal del sonido a medida que entra.

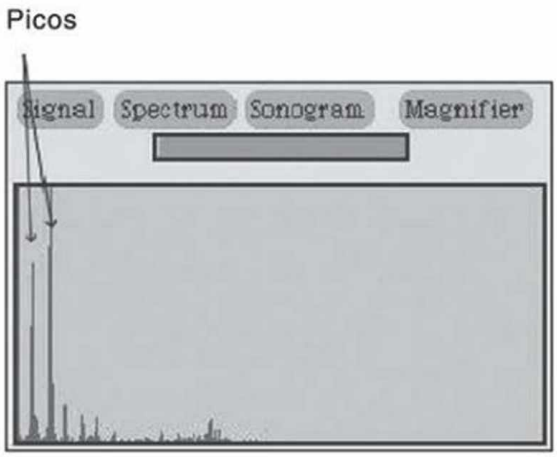



**FIGURA 6.7**
Vista de espectro de un sonido con múltiples picos.

La segunda vista es la **vista de espectro** (figura 6.7). Esta vista es una perspectiva totalmente distinta sobre el sonido. En la sección anterior leyó que, por lo general, los sonidos naturales están compuestos de varias frecuencias diferentes a la vez. La vista de espectro muestra estas frecuencias individuales. Esta vista también se conoce como *dominio de frecuencia*.

Las frecuencias aumentan en la vista de espectro de izquierda a derecha. La altura de una columna indica la cantidad de energía (a grandes rasgos, el volumen) de esa frecuencia en el sonido. Los sonidos naturales se ven como la figura 6.7 con más de un *pico* (elevación en el gráfico). (Las elevaciones más pequeñas alrededor de un pico se consideran por lo común como *ruido*).

El término técnico para describir cómo se genera una vista de espectro es **transformada de Fourier**. Una transformada de Fourier lleva el sonido del *dominio del tiempo* (elevaciones y caídas en el sonido con el tiempo) al dominio de la frecuencia (para identificar qué frecuencias están en un sonido, además de la energía de esas frecuencias, con el tiempo). Las frecuencias aumentan en esta vista de izquierda a derecha (las de más a la izquierda son más bajas, las de más a la derecha son más altas) y una mayor energía en esa frecuencia genera un pico más alto.

La tercera vista es la **vista de sonograma** (figura 6.8). Es muy parecida a la vista de espectro, en cuanto a que describe el dominio de la frecuencia pero presenta estas frecuencias con el tiempo. Cada columna en la vista de sonograma, que algunas veces se le conoce como *división* o *ventana* (*de tiempo*), representa todas las frecuencias en un momento dado en el tiempo. Las frecuencias aumentan en la división desde la más baja (inferior) hasta la más alta (superior). Así, la figura 6.8 representa un sonido que empezó a un nivel bajo, luego se hizo más alto y después volvió a bajar. La *oscuridad* del punto en la columna indica la cantidad de energía de esa frecuencia en el sonido de entrada en el momento dado. La vista de sonograma es excelente para estudiar cómo cambian los sonidos con el tiempo (por ejemplo, cómo cambia el sonido de una tecla de piano al ser golpeada a medida que la nota se desvanece, cómo difieren los distintos instrumentos en cuanto a sus sonidos, o cómo difieren los distintos sonidos vocales).

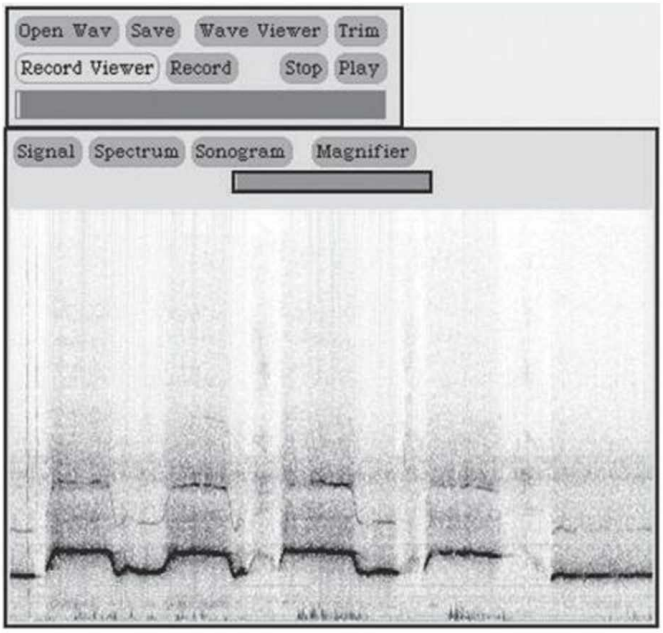



**FIGURA 6.8**
Vista de sonograma de la señal del sonido.

**Tip de funcionamiento: ¡explore los sonidos!**

En verdad debería probar estas diferentes vistas en sonidos reales. Así podrá comprender mucho mejor el sonido y qué es lo que las manipulaciones que realizamos en este capítulo hacen con los sonidos.

■

### 6.1.3 Codificación del sonido

Acabamos de ver cómo funcionan los sonidos físicamente y cómo los percibimos. Para manipular sonidos y reproducirlos en una computadora, tenemos que digitalizarlos. Esto significa que debemos tomar este flujo de ondas y convertirlo en números. Debemos ser capaces de capturar un sonido, tal vez manipularlo y luego reproducirlo (por medio de las bocinas de la computadora) para escuchar lo que capturamos con la mayor exactitud posible.

La primera parte del proceso de digitalizar el sonido se maneja mediante el hardware de la computadora, su maquinaria física. Si una computadora tiene un micrófono y equipo de sonido apropiado (como una tarjeta de sonido SoundBlaster en las computadoras Windows), entonces es posible medir en cualquier momento la cantidad de presión de aire contra el micrófono como un número individual. Los números positivos corresponden a elevaciones en la presión y los números negativos corresponden a las rarefacciones. A esto le llamamos *conversión analógica a digital (ADC)*: pasamos de una señal analógica (una onda de sonido que cambia en forma continua) a un valor digital. Esto significa que podemos obtener una medida instantánea de la presión del sonido, pero sólo es uno de los pasos que debemos llevar a cabo. El sonido es una onda de presión que cambia en forma continua. ¿Cómo almacenamos eso en nuestra computadora?

Por cierto, los sistemas de reproducción en las computadoras funcionan esencialmente igual al revés. El hardware de sonido realiza la *conversión digital a analógica (DAC)* y la señal analógica se envía a continuación a las bocinas. El proceso DAC también requiere números para representar la presión.

Si sabe de cálculo, tendrá una idea de cómo podríamos hacer esto. Sabemos que podemos acercarnos a la medición del área debajo de una curva con cada vez más rectángulos cuya altura coincida con la curva (figura 6.9). Con esta idea, queda bastante claro que si capturamos suficientes de estas lecturas de presión del micrófono, capturamos la onda. A cada lectura de presión le llamamos *muestra*: en sentido literal, estamos "muestreando" el sonido en ese momento. Pero ¿cuántas muestras necesitamos? En el cálculo integral, calculamos el área bajo la curva (en concepto) teniendo un número infinito de rectángulos. Aunque las memorias de computadora están aumentando su tamaño cada vez más todo el tiempo, no podemos capturar un número infinito de muestras por sonido.

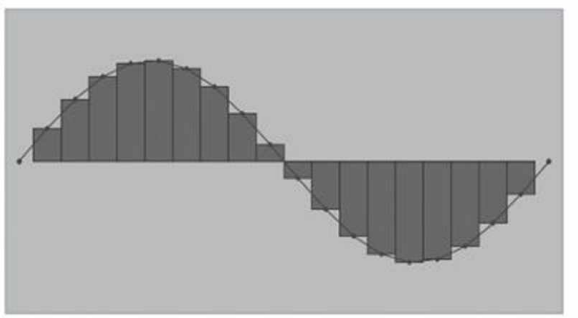


**FIGURA 6.9**
Área bajo una curva estimada con rectángulos.

Los matemáticos y los físicos se cuestionaban estas cosas mucho antes de que hubiera computadoras. De hecho, la respuesta a cuántas muestras necesitamos se calculó hace mucho tiempo. La respuesta depende de la *frecuencia* más alta que desee capturar. Digamos que no le importan los sonidos mayores de 8000 Hz. El **teorema de Nyquist** nos dice que tendríamos que capturar 16 000 muestras por segundo para capturar y definir por completo una onda cuya frecuencia sea menor de 8000 ciclos por segundo.

**Idea de ciencias computacionales: teorema de Nyquist**

Para capturar un sonido de a lo más *n* ciclos por segundo, necesita capturar 2*n* muestras por segundo.

■

Podemos definir el razonamiento básico del teorema de Nyquist con un poco de análisis. Parece bastante claro que si realizamos un muestreo a una frecuencia menor que el tono más alto en el sonido perderíamos muchos ciclos. Imagine que realizó el muestreo del sonido a la misma frecuencia que el sonido en sí. Podría capturar de manera inadvertida cada ciclo en su pico, o en su depresión. Al analizar todos los valores, sólo serían una línea recta. Al usar

el doble de la frecuencia, tenemos una buena probabilidad de atrapar cada ciclo en su pico y en su depresión.

El teorema de Nyquist no es sólo un resultado teórico. Influye en las aplicaciones en nuestra vida diaria. Resulta que las voces humanas por lo general no van más allá de los 4000 Hz. Ésta es la razón por la que nuestro sistema telefónico está diseñado para capturar 8000 muestras por segundo. También es la razón por la que al reproducir música por el teléfono no se escucha muy bien. El límite del oído humano (de casi todos) está alrededor de los 22000 Hz. Si capturamos 44000 muestras por segundo, podríamos capturar cualquier sonido que pudiéramos escuchar. Para crear los CD se captura el sonido a 44100 muestras por segundo: sólo un poco más de 44 kHz por razones técnicas y por un factor compensatorio.

A la velocidad con la que recolectamos las muestras le llamamos *velocidad de muestreo*. La mayoría de los sonidos que escuchamos en la vida diaria están a frecuencias mucho menores de los límites más altos de nuestra capacidad auditiva. Podemos capturar y manipular sonidos en esta clase a una velocidad de muestreo de 22 kHz (22000 muestras por segundo) y sonarán bastante razonables. Si usamos una velocidad de muestreo demasiado baja para capturar un sonido agudo, de todas formas se escuchará algo al reproducir el sonido, pero el tono sonará extraño.

Por lo general, cada una de estas muestras se codifica en 2 bytes, o 16 bits. Aunque hay *tamaños de muestra* más grandes, 16 bits es un valor aceptable para la mayoría de las aplicaciones. El sonido con calidad de CD usa muestras de 16 bits.

### 6.1.4 Números binarios y complemento a dos

En 16 bits, los números que pueden codificarse varían de –32768 a 32767. Éstos no son números mágicos; tienen perfecto sentido cuando comprendemos la codificación. Estos números se codificaron en 16 bits mediante una técnica conocida como **notación de complemento a dos**, pero podemos entenderlo sin necesidad de conocer los detalles de esa técnica. Tenemos 16 bits para representar números positivos y negativos. Vamos a hacer a un lado uno de estos bits (recuerde, es sólo 0 o 1) para representar si estamos hablando sobre un número positivo (0) o uno negativo (1). A éste le llamamos *bit de signo*. Eso nos deja 15 bits para representar el valor actual. ¿Cuántos patrones distintos de 15 bits hay? Podríamos empezar a contar:

```
000000000000000
000000000000001
000000000000010
000000000000011
. . .
111111111111110
111111111111111
```

Esto se ve perturbador. Veamos si podemos averiguar un patrón. Si tenemos dos bits, hay cuatro patrones: 00, 01, 10, 11. Si tenemos tres bits, hay ocho patrones: 000, 001, 010, 011, 100, 101, 110, 111. Resulta que $2^2$ es cuatro y $2^3$ es ocho. Juguemos con cuatro bits. ¿Cuántos patrones hay? $2^4 = 16$. Ahora bien, podemos declarar esto como principio general.

**Idea de ciencias computacionales: $2^n$ patrones en n bits**

Si tiene *n* bits, hay $2^n$ posibles patrones en esos *n* bits.

■

$2^{15} = 32768$. ¿Por qué hay un valor más en el rango negativo que en el positivo? Cero no es negativo ni positivo, pero si queremos representarlo como bits, necesitamos definir cierto patrón como cero. Usamos uno de los valores de rango positivo (en donde el bit de signo es cero) para representar el cero, de modo que ocupa uno de los 32768 patrones.

La forma en que las computadoras representan comúnmente los números enteros positivos y negativos se conoce como *complemento a dos*. En el complemento a dos, los números positivos se muestran de la manera usual en binario. El número 9 es 00001001 en binario. Para calcular el complemento a dos de un número negativo, empezamos con el número positivo en binario y luego lo invertimos de modo que todos los 1 se conviertan en 0 y todos los 0 en 1. Por último, sumamos 1 al resultado. Así, –9 empieza como 00001001, que después de la inversión es 11110110 y al sumarle 1 queda como 11110111. Una ventaja de representar números en complemento a dos es que, si sumamos un número negativo (–3) al número positivo del mismo valor (3), el resultado es cero ya que 1 más 1 es 0 y se acarrea el 1 (figura 6.10).

```
 1111111
 00001001
+11110111
 00000000
```

**FIGURA 6.10**
Suma de 9 y –9 en complemento a dos.

### 6.1.5 Almacenamiento de sonidos digitalizados

El tamaño de muestra es una limitación sobre la amplitud del sonido que puede capturarse. Si tiene un sonido que genere una presión mayor a 32767 (o una rarefacción mayor a –32 768), sólo capturará hasta los límites de 16 bits. Si analizara la onda en la vista de señal, se vería como si alguien hubiera tomado unas tijeras y *recortado* los picos de las ondas. A este efecto le llamamos *recorte* por esa misma razón. Si reproduce (o genera) un sonido recortado, suena mal: parece como si sus bocinas estuvieran descompuestas.

Hay otras formas de digitalizar el sonido, pero hasta ahora ésta es la más común. El término técnico para esta forma de codificar el sonido es *modulación por codificación de pulsos (PCM)*. Tal vez se encuentre con este término si lee más sobre audio o si experimenta con software de audio.

Lo que esto significa es que un sonido en una computadora es una lista larga de números, cada uno de los cuales es una muestra en el tiempo. Hay un orden en estas muestras: si reproduce las muestras desordenadas, no obtendrá el mismo sonido. La manera más eficiente de almacenar una lista ordenada de elementos de datos en una computadora es mediante un **arreglo**. Éste es, en sentido literal, una secuencia de bytes uno después de otro en la memoria. A cada valor en un arreglo le llamamos **elemento**.

Podemos almacenar con facilidad las muestras que componen a un sonido en un arreglo. Considere que en cada dos bytes se almacena una sola muestra. El arreglo será grande: para sonidos con calidad de CD, habrá 44 100 elementos por cada segundo de grabación. Una grabación con duración de un minuto producirá un arreglo con 26 460 000 elementos.

Cada elemento de un arreglo tiene un número asociado, al cual se le conoce como su **índice**. Los números de índice aumentan en forma secuencial. El primer elemento del arreglo está en el índice 0, el segundo en el índice 1, y así en lo sucesivo. El último elemento en el arreglo está en un índice igual al número de elementos en el arreglo menos 1. Podemos considerar un arreglo como una larga línea de cajas, cada una de las cuales contiene un valor y cada caja tiene un número de índice (figura 6.11).

Mediante el uso de MediaTools, puede explorar un sonido (figura 6.12) y darse cuenta si el sonido es silencioso (amplitudes pequeñas) o fuerte (amplitudes grandes). Esto es importante si queremos manipular el sonido. Por ejemplo, los huecos entre las palabras grabadas tienden a ser silenciosos; por lo menos más que las palabras en sí. Podemos detectar en dónde terminan las palabras si analizamos los huecos, como en la figura 6.12.

Pronto veremos cómo leer un archivo que contiene la grabación de un sonido y colocarla en un *objeto sonido*, ver las muestras en el sonido y cambiar los valores de los elementos del arreglo del sonido. Al cambiar los valores en el arreglo, cambiamos el sonido. La manipulación de un sonido es tan sólo cuestión de manipular los elementos en un arreglo.

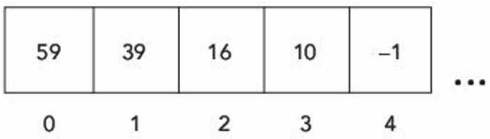



**FIGURA 6.11**
Una representación de los primeros cinco elementos en el arreglo de un sonido real.

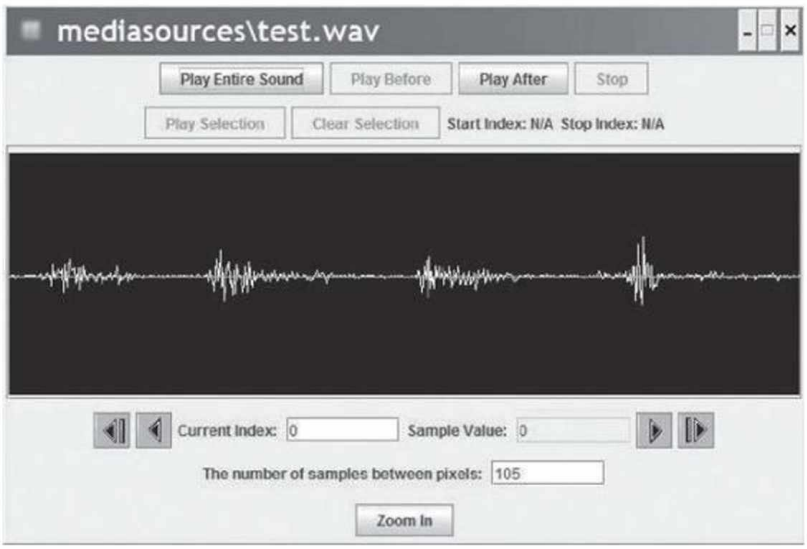



**FIGURA 6.12**
La grabación de un sonido graficada en MediaTools.

## 6.2 MANIPULACIÓN DE SONIDOS

Ahora que sabemos cómo se codifican los sonidos, podemos manipularlos mediante nuestros programas en Python. He aquí lo que tenemos que hacer.

1. Necesitamos obtener el nombre de un archivo WAV y crear un sonido a partir de él.
2. Lo más común es obtener las muestras del sonido. Los objetos muestra son fáciles de manipular; además saben que cuando los cambiamos, deben cambiar de manera automática el sonido original. Primero veremos cómo manipular las muestras para empezar, después cómo manipular las muestras de sonido desde el interior del sonido en sí.
3. Ya sea que obtenga los objetos muestra de un sonido o sólo esté lidiando con las muestras en el objeto sonido, el siguiente paso será hacer algo con esas muestras.
4. Tal vez quiera explorar tanto el sonido original como el modificado para revisar que los resultados coincidan con lo que usted esperaba que ocurriera.
5. Quizás desee escribir el sonido en un nuevo archivo para usarlo en otra parte.

### 6.2.1 Abrir sonidos y manipular muestras

Podemos obtener el nombre de ruta completo de un archivo si lo elegimos con `pickAFile` y después creamos un objeto con `makeSound`. He aquí un ejemplo de cómo hacer esto en JES.

```
>>> nombrearchivo=pickAFile()
>>> print nombrearchivo
C:/ip-book/mediasources/preamble.wav
>>> unSonido=makeSound(nombrearchivo)
>>> print unSonido
Sound file: C:\ip-book\mediasources\preamble.wav
    number of samples: 421110
```

Lo que hace `makeSound` es recoger todos los bytes del nombre de archivo proporcionado como entrada, vaciarlos en la memoria y colocar un gran cartel sobre ellos que diga: "¡Esto es un sonido!". Al ejecutar `unSonido = makeSound(nombrearchivo)`, estamos diciendo: "¡A ese objeto sonido lo llamaremos `unSonido`!". Cuando usamos el sonido como entrada para las funciones, decimos: "Usa ese objeto sonido (sí, al que le llamé `unSonido`) como entrada para esta función".

Podemos obtener las muestras de un sonido mediante la función `getSamples`. Esta función recibe un sonido como entrada y devuelve un arreglo de todas las muestras como objetos muestra. Al ejecutar esta función, tal vez tarde cierto tiempo antes de terminar: será más tiempo para sonidos largos y menos tiempo para sonidos cortos.

La función `getSamples` crea un arreglo de *objetos* muestra a partir del arreglo de muestra básico. Un *objeto* es algo más que sólo un simple valor como vimos antes: una de las diferencias es que un objeto muestra también sabe de qué sonido proviene y cuál es su índice. Más adelante veremos más sobre los objetos, pero ahora sólo basta con saber que `getSamples` nos proporciona un grupo de objetos muestra que podemos manipular; de hecho las manipulaciones se facilitan mucho. Podemos obtener el valor de un objeto muestra usando `getSampleValue` (con un objeto muestra como entrada) y podemos establecer el valor de la muestra con `setSampleValue` (con un objeto muestra y un nuevo valor como entrada).

Pero antes de empezar con las manipulaciones, veamos algunas otras formas de obtener y establecer valores de muestras. Podemos usar la función `getSampleValueAt` para pedir al

sonido que nos proporcione los valores de muestras específicas en índices específicos. Los valores de entrada para `getSampleValueAt` son un sonido y un número de índice.

```
>>> print getSampleValueAt(unSonido,0)
36
>>> print getSampleValueAt(unSonido,1)
29
```

Los valores de índices válidos son cualquier entero (0.1289 no es un buen valor de índice) entre 0 y 1 menos que la longitud del sonido en las muestras. Para obtener la longitud usamos `getLength()`. Observe el error que obtenemos a continuación si tratamos de obtener una muestra usando la longitud del arreglo.

```
>>> print getLength(unSonido)
421110
>>> print getSampleValueAt(unSonido,421110)
You are trying to access the sample at index: 421110,
but the last valid index is at 421109
The error was:
Inappropriate argument value (of correct type).
An error occurred attempting to pass an argument
to a function.
```

**Tip de depuración: obtener más información sobre los errores**

Si obtiene un error y desea más información sobre él, vaya al elemento Options (Opciones) en el menú Edit (Edición) de JES y seleccione Expert (Experto) en vez de Normal (figura 6.13). El modo experto puede darle algunas veces más detalles; tal vez más de los que desee, y probablemente más de una vez, pero puede ser útil a veces.

■

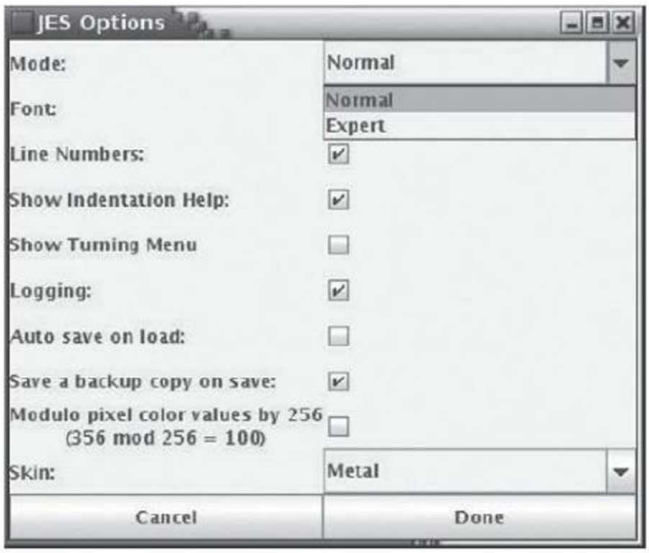



**FIGURA 6.13**
Activación del modo Experto de errores.

De igual forma, podemos cambiar los valores de muestra usando `setSampleValueAt`. Esta función recibe un sonido, un índice y también un nuevo valor para la muestra en ese índice. Podemos revisar de nuevo con `getSampleValueAt`.

```
>>> print getSampleValueAt(unSonido,0)
36
>>> setSampleValueAt(unSonido,0,12)
>>> print getSampleValueAt(unSonido,0)
12
```

**Error común: escribir mal un nombre**

Acaba de ver un grupo de nombres de funciones y algunos de ellos son bastante largos. ¿Qué ocurre si escribe uno de ellos mal? JES se quejará de que no sabe lo que está diciendo, como en el siguiente ejemplo:

```
>>> writeSndTo(unSonido,"misonido.wav")
Name not found globally.
A local or global name could not be found. You need to
define the function or variable before you try to use
it in any way.
```

No es gran cosa. Use la flecha arriba en el teclado para que aparezca lo último que escribió y use la flecha izquierda para regresar al lugar que desea corregir. Luego corrija el error. Asegúrese de regresar al último carácter en la línea usando la tecla de flecha derecha antes de presionar INTRO.

■

¿Qué cree que pasaría si reprodujéramos este sonido? ¿En realidad sonaría diferente que antes, ahora que convertimos la primera muestra del número 36 al número 12? En definitiva, no. Para explicar por qué no, vamos a averiguar cuál es la velocidad de muestreo para este sonido, usando la función `getSamplingRate`, que recibe un sonido como entrada.

```
>>> print getSamplingRate(unSonido)
22050.0
```

El sonido que manipulamos en este ejemplo (una grabación de Mark leyendo una parte del preámbulo de la Constitución de EUA) tiene una velocidad de muestreo de 22 050 muestras por segundo. Al cambiar una muestra se cambia 1/22 050 parte del primer segundo del sonido. Si puede escuchar eso, entonces tiene una capacidad auditiva sorprendente, ¡pero tendremos dudas sobre su honradez!

Sin duda, para realizar una manipulación considerable del sonido, tenemos que manipular cientos, si no es que miles, de muestras. Y es evidente que no vamos a hacerlo escribiendo miles de líneas como éstas:

```
setSampleValueAt(unSonido,0,12)
setSampleValueAt(unSonido,1,24)
setSampleValueAt(unSonido,2,100)
setSampleValueAt(unSonido,3,99)
setSampleValueAt(unSonido,4,-1)
```

Necesitamos aprovechar el hecho de que la computadora ejecute nuestra receta, indicándole que haga algo cientos o miles de veces. Ése será el tema de la siguiente sección.

Pero por ahora terminaremos esta sección hablando sobre cómo escribir sus resultados de vuelta a un archivo. Una vez que manipulamos el sonido y queremos grabarlo para usarlo en otra parte, usamos `writeSoundTo`. Esta función recibe un sonido y un nuevo nombre de archivo como entrada. Asegúrese de que su archivo termine con la extensión ".wav" si va a guardar un sonido, para que su sistema operativo sepa qué hacer con él.

```
>>> print nombrearchivo
C:/ip-book/mediasources/preamble.wav
>>> writeSoundTo(unSonido,"C:/ip-book/mediasources/nuevo-preamble.wav")
```

Tal vez descubra que, al reproducir sonidos con mucha frecuencia, si usamos `play` un par de veces en rápida sucesión, se mezclarán los sonidos. El segundo `play` empieza antes de que termine el primero. ¿Cómo podemos asegurarnos de que la computadora reproduzca sólo un sonido y luego espere a que termine? Podemos usar la función `blockingPlay`, que funciona igual que `play`, sólo que espera a que el sonido termine para que ningún otro sonido pueda interferir con su reproducción.

### 6.2.2 Uso de MediaTools de JES

Las herramientas MediaTools de JES están disponibles mediante el menú MEDIATOOLS en JES. Cuando seleccione la herramienta de imagen o de sonido, aparecerá un menú de nombres de *variables* de imágenes o sonidos, según la herramienta que seleccione (figura 6.14). Haga clic en Aceptar para entrar a la herramienta de sonido de JES. También puede invocar a la herramienta de sonido si explora el sonido, de igual forma que con las imágenes.

```
>> explore(unSonido)
```

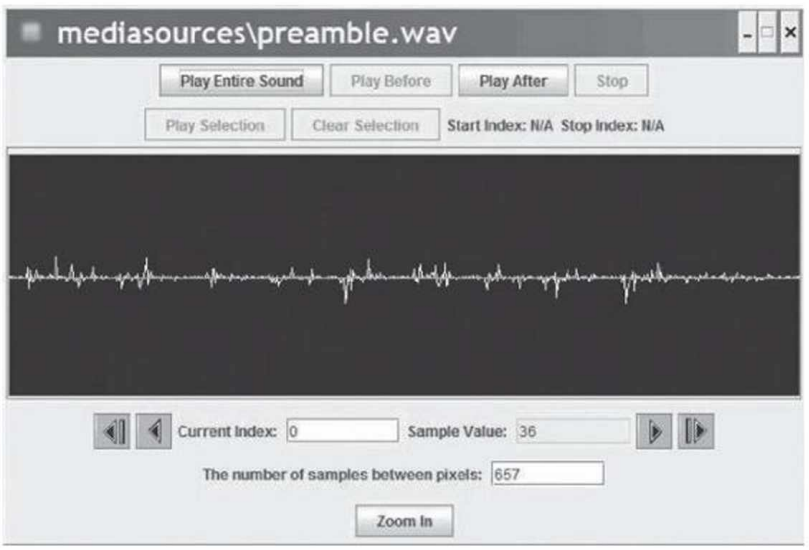



**FIGURA 6.14**
Exploración de un sonido en JES.

La herramienta de sonido le permite explorar un sonido.

- Puede reproducir el sonido y después hacer clic en cualquier parte del mismo para establecer un punto de cursor, y luego reproducirlo antes o después de ese cursor.
- Puede seleccionar una región (al hacer clic y arrastrar) y luego reproducir sólo esa región (figura 6.15).
- Al establecer un cursor, aparecerán el índice de muestra y el valor de muestra en ese punto.
- También puede hacer un acercamiento para ver todos los valores del sonido (figura 6.16); tendrá que desplazarse por la ventana para ver todos los valores.

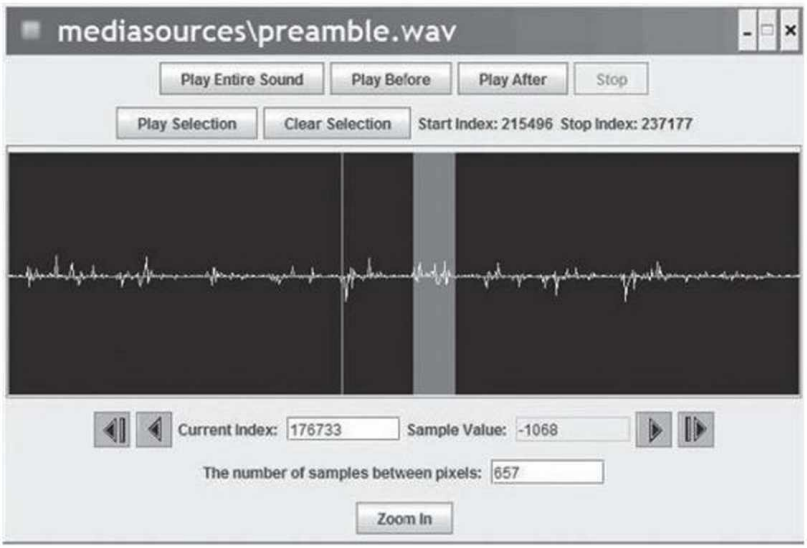



**FIGURA 6.15**
Exploración de un sonido en JES.

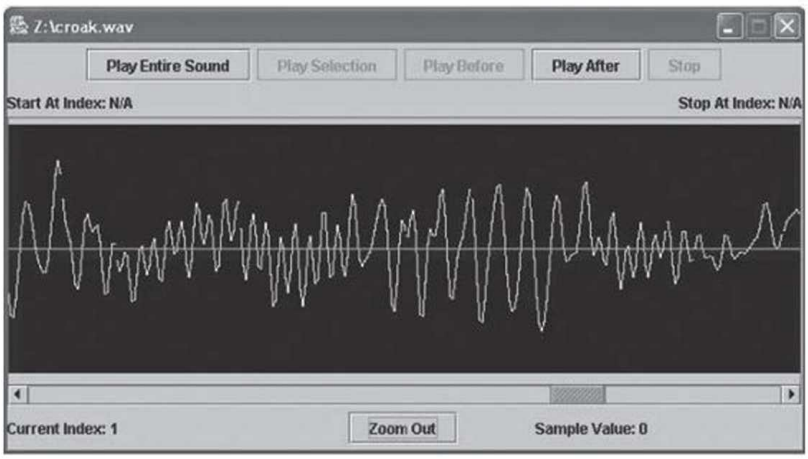



**FIGURA 6.16**
Acercamiento para ver todos los valores de un sonido.

**Error común: Windows y archivos WAV**

El mundo de los archivos WAV no es tan compatible y uniforme como podríamos querer. Los archivos WAV que se crean con otras aplicaciones (por ejemplo, la Grabadora de sonidos de Windows) *tal vez* no se reproduzcan en JES, así como tal vez los archivos WAV de JES no se reproduzcan en todas las demás aplicaciones (como WinAmp 2). La aplicación Apple QuickTime Player Pro (`http://www.apple.com/quicktime`) es buena para leer *cualquier* archivo WAV y exportar uno nuevo que casi cualquier otra aplicación pueda leer.

■

### 6.2.3 Ciclos

El problema al que nos enfrentamos es común en la computación: ¿cómo logramos que la computadora haga algo una y otra vez? Necesitamos que la computadora **realice un ciclo o una iteración**. Python tiene comandos especiales para los ciclos (o iteraciones). La mayor parte de las veces usaremos el comando `for`. Un ciclo `for` ejecuta comandos (que usted especifica) para una secuencia (que usted provee), en donde cada vez que se ejecutan los comandos, una variable específica (que usted nombrará) tendrá el valor de un elemento diferente de la secuencia.

## 6.3 CAMBIAR EL VOLUMEN DE LOS SONIDOS

En párrafos anteriores dijimos que la amplitud de un sonido es el principal factor en el volumen. Esto significa que si aumentamos la amplitud, aumentamos el volumen. O si reducimos la amplitud, reducimos el volumen.

No queremos que se confunda: cambiar la amplitud no significa ir y dar vuelta a la perilla del volumen en su aparato de sonido. Si bajamos el volumen de su bocina (o el volumen de la computadora), el sonido nunca podrá oírse muy fuerte. El punto es lograr que el sonido en sí sea más fuerte. ¿Alguna vez vio una película en TV en donde, sin cambiar el volumen en la televisión, el sonido se vuelve tan bajo que apenas si puede escucharlo? (Nos viene a la mente el diálogo de Marlon Brando en la película *El Padrino*). O ¿alguna vez ha observado que los comerciales siempre tienen un sonido más fuerte que los programas de TV normales? Eso es lo que estamos haciendo aquí. Podemos hacer que los sonidos *griten* o *susurren* al ajustar la amplitud.

### 6.3.1 Aumentar el volumen

Vamos a incrementar el volumen de un sonido, para lo cual duplicaremos el valor de cada muestra en el sonido. Podemos usar `getSamples` para obtener una secuencia (arreglo) de las muestras en un sonido. Podemos usar un ciclo for para iterar por todas las muestras en la secuencia. Para cada muestra, obtendremos el valor actual y luego lo estableceremos al valor actual multiplicado por 2.

He aquí una función que duplica la amplitud de un sonido de entrada.

**Programa 56: aumentar el volumen de un sonido, duplicando la amplitud**

```
def aumentarVolumen(sonido):
  for muestra in getSamples(sonido):
    valor = getSampleValue(muestra)
    setSampleValue(muestra,valor * 2)
```

■

Ahora vaya y escriba lo anterior en el área del programa de JES. Haga clic en LOAD PROGRAM (cargar programa) para que Python procese la función y nos permita usar el nombre

`aumentarVolumen`. Siga el ejemplo a continuación para que tenga una mejor idea de cómo funciona todo esto.

Para usar esta receta, tiene que crear primero un sonido y luego pasar ese sonido a la función `aumentarVolumen` como entrada. En el siguiente ejemplo, para obtener el nombre de archivo establecemos la variable `f` de manera explícita a una cadena que representa un nombre de archivo, en vez de usar `pickAFile`. No olvide que no puede escribir este código y esperar que funcione como está: sus nombres de ruta serán distintos de los de nosotros.

```
>>> f="C:/ip-book/mediasources/test.wav"
>>> s=makeSound(f)
>>> explore(s)
>>> aumentarVolumen(s)
>>> explore(s)
```

Primero creamos un sonido, al cual llamamos `s`. Después exploramos el sonido, con lo cual aparecerá la herramienta de sonido de JES con una copia del sonido. Luego evaluamos `aumentarVolumen(s)`; el sonido llamado `s` *también* se llama `sonido`, pero sólo dentro de esa función. Éste es un punto *muy* importante. ¡Ambos nombres hacen referencia al mismo sonido! Los cambios que ocurren en `aumentarVolumen` en realidad cambian el *mismo* sonido. Puede considerar cada nombre como un *alias* para el otro: hacen referencia a la misma *cosa.*

Hay un punto que debemos mencionar aquí de paso, pero que se volverá más importante después: cuando termina la función `aumentarVolumen`, el nombre sonido *no tiene valor.* Sólo existe durante el tiempo que dure la ejecución de esa función. Decimos que sólo existe dentro del *alcance* de la función `aumentarVolumen`. El alcance de una variable es el área en la que se conoce. Por ejemplo, las variables definidas en el área de comandos tienen alcance de área de comandos, ya que sólo se conocen en esta área. Las variables que se definen en una función tienen alcance de función, ya que sólo se conocen en esa función.

Ahora podemos reproducir el archivo para escuchar que es más fuerte y escribirlo en un nuevo archivo.

```
>>> play(s)
>>> writeSoundTo(s,"c:/ip-book/mediasources/test-masfuerte.wav")
```

**Error común: mantenga los sonidos cortos**

Los sonidos más extensos ocupan más memoria y se procesan con más lentitud.

■

### 6.3.2 ¿De verdad funcionó eso que hicimos?

Ahora bien, ¿en realidad es más fuerte, o sólo parece serlo? Podemos verificar esto de varias formas. Siempre es posible hacer el sonido aún más fuerte si evaluamos `aumentarVolumen` en nuestro sonido unas cuantas veces más; en un momento dado quedará totalmente convencido de que el sonido es más fuerte. Pero existen formas de evaluar efectos incluso más sutiles.

Si comparamos gráficos de los dos sonidos usando la herramienta de sonido de MediaTools de JES, descubriremos que el gráfico del sonido tiene una mayor amplitud después de aumentarlo mediante el uso de nuestra función. Compruébelo en la figura 6.17.

Tal vez no esté seguro de que está viendo en verdad una onda más grande en la segunda imagen. Puede usar la herramienta de sonido de JES para comprobar los valores de las

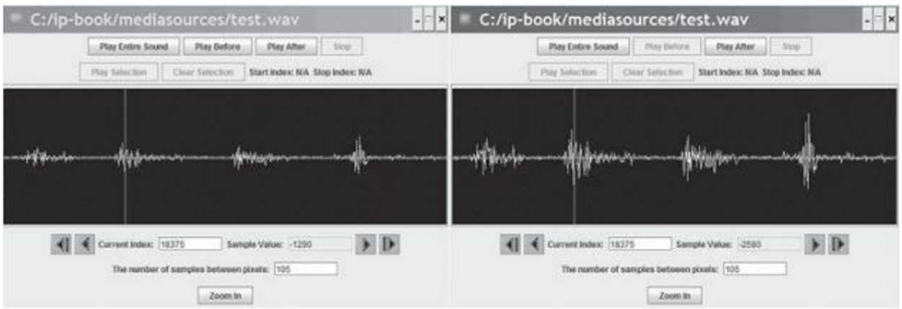



**FIGURA 6.17**
Comparación de los gráficos del sonido original (*izquierda*) y el sonido más fuerte (*derecha*).

muestras individuales. Haga clic en una parte de la onda que tenga un valor distinto de 0 en una herramienta de sonido, después introduzca el índice en la otra herramienta de sonido y presione INTRO. Compare los valores de las muestras. Observe en la figura 6.17 que el valor original en el índice 18375 era de –1290 y después de aumentar el volumen es –2580, por lo que la función sí duplicó el valor.

Por último, siempre podremos asegurarnos de los resultados desde JES. Si ha estado siguiendo el ejemplo,[1] entonces la variable `s` es ahora el sonido más fuerte. La variable `f` debería ser todavía el nombre de archivo del sonido original. Ahora cree un nuevo objeto de sonido que sea el *original*; en el siguiente ejemplo le llamamos `sOriginal` (por *sonido original*). Revise cualquier valor de muestra que desee. El sonido más fuerte siempre tendrá el doble de los valores de muestra del sonido original.

```
>>> print s
Sound file: C:/ip-book/mediasources/test.wav
    number of samples: 67585
>>> print f
C:/ip-book/mediasources/test.wav
>>> sOriginal=makeSound(f)
>>> print getSampleValueAt(s,0)
0
>>> print getSampleValueAt(sOriginal,0)
0
>>> print getSampleValueAt(s,18375)
-2580
>>> print getSampleValueAt(sOriginal,18375)
-1290
>>> print getSampleValueAt(s,1000)
4
>>> print getSampleValueAt(sOriginal,1000)
2
```

Aquí puede ver que los valores negativos se hacen *más* negativos. Esto es lo que queremos decir con "aumentar la amplitud". La amplitud de la onda se expande en *ambas* direcciones. Tenemos que hacer la onda más grande tanto en la dimensión positiva como en la negativa.

[1] ¿Qué? ¿No lo ha hecho? ¡*Debería* hacerlo! Tendrá mucho más sentido si lo prueba usted mismo.

Es importante que haga lo que acaba de leer en este capítulo: *dude* de sus programas. ¿En *realidad* hacen lo que quería que hicieran? La forma de verificarlo es mediante la *prueba*. De esto trata esta sección. Acaba de ver varias formas de realizar pruebas:

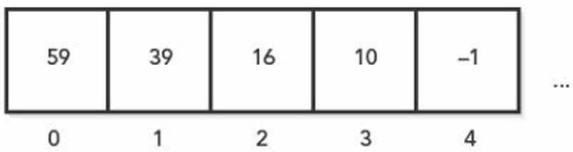
- Verificando piezas de los resultados (usando la herramienta de sonido de JES).
- Escribiendo instrucciones de código adicionales que verifican los resultados del programa original.

### Cómo funciona

Vamos a recorrer el código lentamente, para ver cómo funciona este programa.

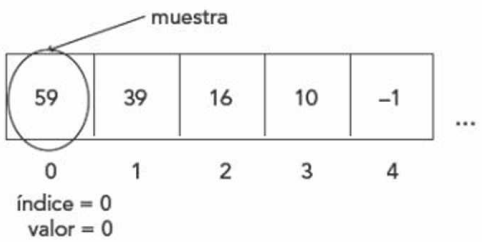
```
def aumentarVolumen(sonido):
  for muestra in getSamples(sonido):
    valor = getSampleValue(muestra)
    setSampleValue(muestra,valor * 2)
```

Recuerde nuestra imagen de cómo podrían verse las muestras en un arreglo de sonido. La siguiente imagen muestra primero los valores en el sonido creados a partir del archivo **gettysburg.wav**.



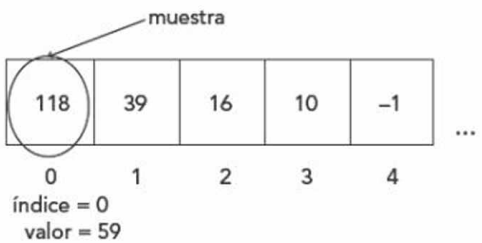

Esto es lo que regresaría `getSamples(sonido)`: un arreglo de valores de muestra, cada uno numerado. El ciclo `for` nos permite recorrer cada muestra, una a la vez. El nombre `muestra` se asignará a cada muestra en turno.

Cuando empiece el ciclo `for`, `muestra` será el nombre de la primera muestra.

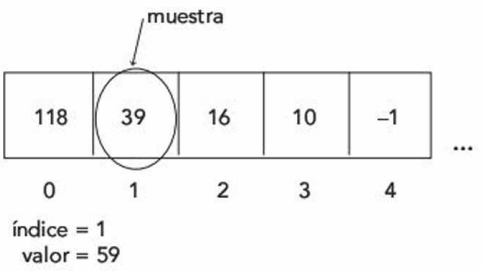



La variable valor recibirá el valor de 59 cuando se ejecute la instrucción `valor=getSampleValue(muestra)`

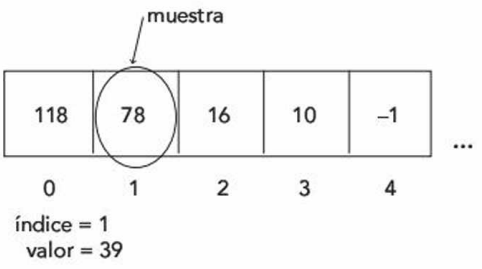
La muestra a la que hace referencia el nombre muestra se duplicará entonces con `setSampleValue(muestra,valor*2)`



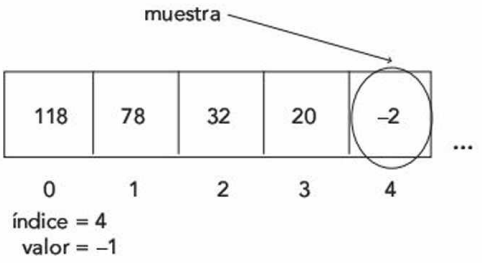
Ése es el final de la primera pasada por el cuerpo del ciclo `for`. A continuación Python empezará el ciclo de nuevo y moverá `muestra` para que apunte al *siguiente* elemento en el arreglo.




El `valor` se establece de nuevo al valor de la muestra y luego ésta se duplica.




Así es como se verá después de recorrer el ciclo cinco veces.




El ciclo `for` *sigue* recorriendo todas las muestras: ¡decenas de miles de ellas! ¡Gracias a Dios que la *computadora* es la que ejecuta este programa!

Una manera de pensar en lo que ocurre aquí es que, en realidad, el ciclo `for` no *hace* nada, en cuanto a cambiar algo en el sonido. Sólo el *cuerpo* del ciclo realiza el trabajo. El ciclo `for` indica a la computadora *qué* hacer. Es un manejador. Lo que la computadora hace es algo como lo siguiente:

```
muestra = muestra #0
valor = valor de la muestra, 59
cambiar muestra a 118
muestra = muestra #1
valor = 39
cambiar muestra a 78
muestra = muestra #2
...
```

```
muestra = muestra #4
valor = -1
cambiar muestra a -2
...
```

El ciclo `for` sólo dice: "Haz todo esto para cada elemento en el arreglo". Es el *cuerpo* del ciclo el que contiene los comandos de Python que se ejecutan.

Lo que acaba de leer en esta sección se conoce como **rastrear** el programa. Recorrimos con lentitud la forma en que se ejecuta cada paso en el programa. Dibujamos imágenes para describir los datos en el programa. Usamos números, flechas, ecuaciones e incluso español cotidiano para explicar lo que ocurría en el programa. Ésta es la técnica más importante en la programación. Es parte de la *depuración*. Su programa *no* siempre funcionará. Se lo garantizamos en definitiva, sin sombra de duda; usted escribirá código que no haga lo que desea. Pero la computadora *hará algo*. ¿Cómo podemos averiguar qué *está* haciendo? Hay que depurar, y la forma más poderosa de hacerlo es mediante un rastreo del programa.

### 6.3.3 Reducir el volumen

La reducción del volumen es el inverso del proceso que describimos antes.

**Programa 57: reducir el volumen de un sonido, reduciendo la amplitud a la mitad**

```
def reducirVolumen(sonido):
  for muestra in getSamples(sonido):
    valor = getSampleValue(muestra)
    setSampleValue(muestra,valor * 0.5)
```

■

#### Cómo funciona

- Nuestra función recibe un objeto de sonido como entrada. Dentro de la función `reducirVolumen`, al sonido de entrada se le llamará `sonido`; no importa el nombre que tenga en el área de comandos.
- La variable `muestra` representará a todas y cada una de las muestras en el sonido de entrada.
- Cada vez que se asigna una nueva muestra a la variable `muestra`, obtenemos el *valor* de la muestra y lo colocamos en la variable `valor`.
- Después establecemos el valor de muestra al 50% de su valor actual, multiplicando `valor` por 0.5, y asignando el resultado al valor de muestra.

Podemos usarlo así:

```
>>> f=pickAFile()
>>> print f
C:/ip-book/mediasources/test-masfuerte.wav
>>> sonido=makeSound(f)
>>> explore(sonido)
>>> play(sonido)
>>> reducirVolumen(sonido)
>>> explore(sonido)
>>> play(sonido)
```

Podemos incluso hacerlo de nuevo y reducir el volumen aún más.

```
>>> reducirVolumen(sonido)
>>> explore(sonido)
>>> play(sonido)
```

### 6.3.4 Interpretación de las funciones en los sonidos

Las lecciones sobre cómo trabajan las funciones con las imágenes (de la sección 3.4.1) se aplican aquí en los sonidos también. Por ejemplo, podríamos colocar todas las llamadas a `pickAFile` y `makeSound` directamente en nuestras funciones, como `aumentarVolumen` y `reducirVolumen`, pero eso significaría que las funciones realizarían más de *una y sólo una cosa*. Si tuviéramos que aumentar o reducir el volumen a un grupo de sonidos, nos resultaría molesto tener que estar eligiendo archivos.

Podemos escribir funciones que reciban varias entradas. Por ejemplo, he aquí un programa para `cambiarVolumen`. Acepta un factor que se multiplica por el valor de cada muestra. Esta función puede usarse para aumentar o reducir la amplitud (y por ende, el volumen).

**Programa 58: cambiar el volumen de un sonido por un factor dado**

```
def cambiarVolumen(sonido, factor):
  for muestra in getSamples(muestra):
    valor = getSampleValue(muestra)
    setSampleValue(muestra, valor * factor)
```

■

Este programa es sin duda más flexible que `aumentarVolumen` o `reducirVolumen`. ¿Acaso eso lo hace mejor? En definitiva es para ciertos propósitos (por ejemplo, si escribe software para realizar procesamiento de audio en general), pero para otros fines es mejor tener funciones separadas con nombres claros para aumentar y reducir el volumen. Recuerde que el software está escrito para humanos; escriba software comprensible para las personas que leerán y usarán su software.

En este libro reutilizamos el nombre `sonido` muchas veces. Lo usamos para nombrar sonidos que leemos del disco en el área de comandos y lo usamos como marcador de posición para las entradas a las funciones. *Eso está bien.* Los nombres pueden tener distintos significados dependiendo del contexto. El interior de una función es un contexto distinto al área de comandos. Si crea una variable en un contexto de función (como `valor` en el programa 58), entonces esa variable no existirá cuando regrese al área de comandos. Podemos devolver valores de un contexto de función al área de comandos (o de otra función invocadora) mediante el uso de la instrucción `return`, de la cual hablaremos más adelante.

## 6.4 NORMALIZACIÓN DE SONIDOS

Si piensa en ello, tal vez le parezca extraño que los dos últimos programas funcionen. Podemos multiplicar los números que representan un sonido, y éste parecerá (en esencia) igual ante nuestros oídos, sólo que más fuerte. La forma en que experimentamos un sonido depende menos de los números específicos que de la *relación* entre ellos. Recuerde que la forma general de la forma de onda de un sonido depende de *muchas* muestras. En general, si multiplicamos todas las muestras por el mismo multiplicador, sólo afectaremos la forma en que detectamos el

volumen (intensidad) y no al sonido en sí (trabajaremos para cambiar el sonido en sí en futuras secciones).

Una operación común que las personas desean llevar a cabo con los sonidos es hacerlos lo **más fuertes posibles**. A esto se le conoce como **normalización**. En realidad no es difícil hacerlo, pero se requieren unas cuantas variables más. He aquí la receta, en español, que necesitamos dar a la computadora para que la lleve a cabo.

- Tenemos que averiguar cuál es la muestra más grande en el sonido. Si ya está en el valor máximo (32767), entonces no podemos aumentar el volumen y obtener todavía lo que se asemeje al mismo sonido. Recuerde que tenemos que multiplicar todas las muestras por el mismo multiplicador.

  Es una receta (*algoritmo*) fácil encontrar el valor más grande; algo así como una *subreceta* dentro de la receta de normalización general. Defina un nombre (por decir, `masgrande`) y asígnele un pequeño valor (el 0 estará bien). Ahora revise todas las muestras. Si encuentra una muestra mayor a `masgrande`, cambie `masgrande` para que tenga ese valor mayor. Siga revisando las muestras, pero ahora compárelas con el *nuevo* valor más grande. En un momento dado, el valor más grande de todos en el arreglo estará en la variable `masgrande`.

  Para ello, necesitamos una forma de averiguar el valor máximo de dos valores. Python cuenta con una función integrada llamada `max` que puede hacer esto.

```
>>> print max(8,9)
9
>>> print max(3,4,5)
5
```

- A continuación necesitamos averiguar por qué valor vamos a multiplicar todas las muestras. Queremos que el valor más grande se convierta en 32767. Por lo tanto, debemos encontrar un *multiplicador* de tal forma que

  (*multiplicador*)(*masgrande*) = 32767.

  Si resolvemos para el multiplicador:

  *multiplicador* = 32767/*masgrande*. El multiplicador tendrá que ser un número de punto flotante (con un componente decimal), por lo que necesitamos convencer a Python de que no todo aquí es un entero. Resulta ser que esto es fácil: use 32767.0. Simplemente agregue el ".0".

- Ahora hay que iterar por todas las muestras, como hicimos para `aumentarVolumen`, y multiplicar el valor por el multiplicador.

  He aquí un programa para normalizar sonidos.

**Programa 59: normalizar el sonido hasta una amplitud máxima**

```
def normalizar(sonido):
    masgrande = 0
    for s in getSamples(sonido):
        masgrande = max(masgrande,getSampleValue(s) )
    multiplicador = 32767.0 / masgrande
    print "El valor de muestra más grande en el sonido original fue",masgrande
    print "El multiplicador es", multiplicador

    for s in getSamples(sonido):
        masfuerte = multiplicador * getSampleValue(s)
        setSampleValue(s,masfuerte)
```

■

Hay varias cosas que tener en cuenta sobre este programa.

- ¡Hay líneas en blanco! A Python no le importan esas líneas. Puede ser útil agregar líneas en blanco para dividir y mejorar el grado de comprensión de los programas más extensos.
- ¡Hay instrucciones `print`! Estas instrucciones pueden ser *realmente* útiles. En primer lugar, nos proporcionan cierta retroalimentación de que el programa está ejecutándose; algo útil en los programas que se ejecutan por periodos prolongados. En segundo lugar, nos muestran lo que el programa está encontrando, que puede ser interesante. En tercer lugar, es un método de prueba grandioso, además de una manera de depurar sus programas. Imaginemos que los resultados de `print` mostraran que el multiplicador es menor a 1.0. Sabemos que este tipo de multiplicador *reduce* el volumen. Esto nos indicaría que es muy probable que algo saliera mal.
- Algunas de las instrucciones en este programa son bastante extensas, por lo que se dividen en varias líneas en el texto. *¡Escríbalas como una sola línea!* Python no le permitirá teclear INTRO hasta el final de la instrucción; asegúrese de que todas sus instrucciones `print` estén cada una en una sola línea.

He aquí cómo se ve el programa en ejecución.

```
>>> archivo = "c:/ip-book/mediasources/test.wav"
>>> sonido = makeSound(archivo)
>>> explore(sonido)
>>> normalizar(sonido)
El valor de muestra más grande en el sonido original fue 11702
El multiplicador es 2.8001196376687747
>>> explore(sonido)
>>> play(sonido)
```

Emocionante, ¿verdad? Sin duda, la parte interesante es escuchar el volumen mucho más fuerte, lo cual es bastante difícil de demostrar en un libro.

### 6.4.1 Generación del recorte

En párrafos anteriores hablamos sobre el *recorte*, el efecto cuando las curvas normales del sonido se rompen debido a las limitaciones del tamaño de la muestra. Una forma de generar el recorte es seguir aumentando el volumen. Otra forma es forzarlo de manera explícita.

¿Qué pasaría si usted *sólo* tuviera los valores de muestra más grande y más pequeño posibles? ¿Qué pasaría si todos los valores positivos fueran los valores *máximos* y todos los valores negativos fueran los valores *mínimos*? Pruebe este programa, en especial en sonidos que contengan palabras.

**Programa 60: establecer todas las muestras a valores máximos**

```
def soloMaximizar(sonido):
  for muestra in getSamples(sonido):
     valor = getSampleValue(muestra)
     if valor >= 0:
        setSampleValue(muestra,32767)
     if valor < 0:
        setSampleValue(muestra,-32768)
```

■

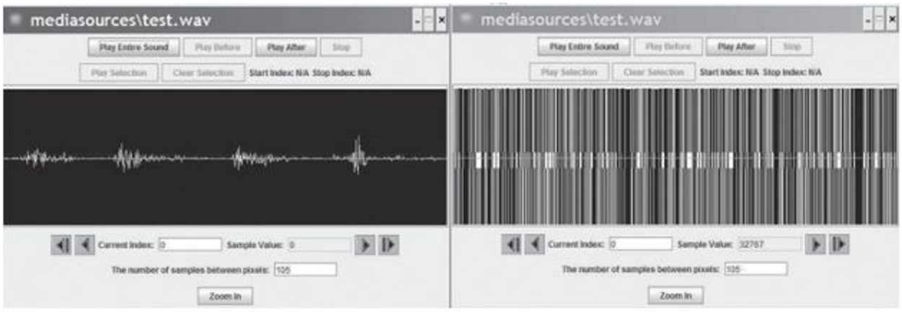



**FIGURA 6.18**
El sonido original y el sonido sólo con los valores máximos.

El resultado final se ve bastante extraño (figura 6.18). Al reproducir el sonido, escuchará algunos ruidos espantosos. Parte de ese sonido espantoso es el recorte. Otras partes de ese sonido espantoso se deben a que cada pequeño ruido de fondo ahora se ha vuelto *¡LO MÁS FUERTE POSIBLE!* Lo que de verdad sorprende es que *todavía* podemos distinguir las palabras en los sonidos que manipulamos con esta función. ¿A qué se debe esto? Hay que tener en cuenta que la información de la *frecuencia* es idéntica tanto en el sonido original como en el maximizado. Nuestra habilidad para descifrar palabras del ruido se basa en su mayor parte en la información de frecuencia, y es en extremo poderosa.

**Idea de ciencias computacionales: sólo necesitamos un bit por muestra para una expresión oral legible**

¿Cuántos bits hay en cada muestra en nuestro sonido maximizado? Sabemos que la computadora almacena 16 bits de valor por cada muestra, pero el término *bit* en realidad se refiere a una cantidad de información. Cada muestra en el sonido maximizado tiene sólo dos *estados* o *valores* posibles. Una pieza de información con sólo dos valores es un bit. Por lo tanto, sólo usamos un bit de información para grabar el sonido maximizado; y es una expresión oral legible. Ahora sabemos que, si nuestra velocidad de muestreo es lo bastante alta, podemos grabar una expresión oral legible con sólo un bit por muestra.

■

## RESUMEN DE PROGRAMACIÓN

En este capítulo, hablamos sobre varios tipos de codificaciones de datos (u objetos).

| | |
|---|---|
| Sonidos | Codificaciones de sonidos, que por lo general provienen de un archivo WAV. |
| Muestras | Colecciones de objetos de muestra, cada uno indexado por un número (por ejemplo, muestra #0, muestra #1). `muestras[0]` es el primer objeto de muestra. Puede manipular cada muestra en las muestras de la siguiente forma: `for s in muestras:`. |
| Muestra | Un valor entre –32 000 y 32 000 (aproximadamente) que representa el voltaje que un micrófono generaría en un instante dado al grabar un sonido. Por lo general la longitud del instante es 1/44 100 de un segundo (para sonido con calidad de CD), o de 1/22 050 de un segundo (para un sonido con calidad suficiente en la mayoría de las computadoras). Un objeto de muestra recuerda de cuál sonido proviene, por lo que si modifica su valor, sabe cómo regresar y modificar la muestra correcta en el sonido. |

Éstas son las funciones utilizadas e introducidas en este capítulo:

| | |
|---|---|
| `int` | Devuelve la parte entera del valor de entrada. |
| `max` | Recibe todos los números que usted quiera y devuelve el valor más grande. |

## FUNCIONES Y PIEZAS DE ARCHIVOS DE SONIDO

| | |
|---|---|
| `pickAFile` | Deja que el usuario seleccione un archivo y devuelve el nombre de ruta completo como una cadena. No hay entrada. |
| `makeSound` | Recibe un nombre de archivo como entrada, lee el archivo y crea un sonido a partir de él. Devuelve el sonido. |

## FUNCIONES Y PIEZAS DE UN OBJETO DE SONIDO

| | |
|---|---|
| `play` | Reproduce un sonido que se proporciona como entrada. Sin valor de retorno. |
| `getLength` | Recibe un sonido como entrada y devuelve el número de muestras en el sonido. |
| `getSamples` | Recibe un sonido como entrada y devuelve las muestras en el sonido en un arreglo unidimensional. |
| `blockingPlay` | Reproduce el sonido que se proporciona como entrada y se asegura que ningún otro sonido se reproduzca exactamente al mismo tiempo (pruebe con dos `play`, uno después del otro). |
| `playAtRate` | Recibe un sonido y una proporción (1.0 indica una velocidad normal, 2.0 es el doble de rápido y 0.5 es la mitad de rápido) y reproduce el sonido con esa proporción. La duración siempre es *la misma* (por ejemplo, si lo reproduce con el doble de rapidez, el sonido se reproduce *dos veces* para llenar el tiempo dado). |
| `playAtRateDur` | Recibe un sonido, una proporción y una duración como el número de muestras a reproducir. |
| `writeSoundTo` | Recibe un sonido y un nombre de archivo (una cadena); luego escribe el sonido en el archivo como WAV (asegúrese de que el nombre de archivo termine en ".wav" si desea que el sistema operativo lo maneje en forma correcta). |
| `getSamplingRate` | Recibe un sonido como entrada y devuelve el número que representa la cantidad de muestras en cada segundo para el sonido. |
| `getLength` | Devuelve la longitud del sonido como un número de muestras. |

## FUNCIONES Y PIEZAS ORIENTADAS A MUESTRAS

| | |
|---|---|
| `getSampleValueAt` | Recibe un sonido y un índice (un valor entero) y devuelve el valor de la muestra (entre –32000 y 32000) para ese objeto. |
| `setSampleValueAt` | Recibe un sonido, un índice y un valor (debe estar entre –32000 y 32000) y establece el valor de la muestra en el índice dado, en el sonido dado, al valor dado. |
| `getSampleObjectAt` | Recibe un sonido y un índice (un valor entero) y devuelve el objeto de muestra en ese índice. |
| `getSampleValue` | Recibe un objeto de muestra y devuelve su valor (entre –32000 y 32000). También funciona con `getValue`. |
| `setSampleValue` | Recibe un objeto de muestra y un valor, y establece la muestra con el valor. También funciona con `setValue`. |
| `getSound` | Recibe un objeto muestra y devuelve el sonido del que forma parte. |

## PROBLEMAS

6.1 Defina los siguientes términos.

1. Recorte
2. Normalización
3. Amplitud
4. Frecuencia
5. Rarefacciones

6.2 Escriba el número –9 en complemento a dos. Escriba el número 4 en complemento a dos. Escriba el número –27 en complemento a dos.

6.3 Abra la vista de SONOGRAMA y diga algunos sonidos de vocales. ¿Hay algún patrón distintivo? ¿Acaso las "o" siempre suenan igual? ¿Y las "a"? ¿Acaso importa si cambia la voz a la de otra persona? ¿Son los patrones iguales?

6.4 Consiga un par de instrumentos diferentes y reproduzca la misma nota en ellos en el editor de sonido de la aplicación de MediaTools con la vista de sonograma abierta. ¿Acaso todos los "do" son iguales? Si utiliza la visualización de MediaTools, ¿puede *ver* las diferencias entre los sonidos?

6.5 Pruebe una variedad de archivos WAV como instrumentos, usando el teclado de piano en el editor de sonido de la aplicación MediaTools. ¿Qué tipos de grabaciones funcionan mejor como instrumentos?

6.6 La receta para aumentar el volumen (programa 56, página 161) recibe un sonido como entrada. Escriba una función llamada `aumentarVolumenConNombre` que reciba un nombre de archivo como entrada y después reproduzca el sonido más fuerte con `play`.

6.7 Escriba una función para aumentar el volumen para todos los valores positivos y reducir el volumen para todos los valores negativos. ¿Puede todavía entender las palabras en el sonido?

6.8 Escriba una función para encontrar el valor más pequeño en un sonido e imprimirlo en pantalla.

6.9 Escriba una función para contar el número de veces que el valor en un sonido es 0 e imprimir el total en pantalla.

6.10 Escriba una función para establecer valores mayores que 0 al máximo valor positivo (32767) en un sonido.

6.11 Vuelva a escribir la función para aumentar el volumen (programa 56) de modo que reciba dos entradas: el sonido al que se va a aumentar el volumen y un nombre de archivo en donde deberá almacenarse el nuevo sonido más fuerte. Luego aumente el volumen y escriba el sonido en el archivo con el nombre. También podría intentar volver a escribirlo de modo que reciba un nombre de archivo como entrada en vez del sonido, de modo que ambas entradas sean nombres de archivos.

6.12 Vuelva a escribir la función para aumentar el volumen (programa 56) de modo que reciba dos entradas: un sonido al que se va a aumentar el volumen y un *multiplicador*. Use el multiplicador para indicar *cuánto* aumentar la amplitud de las muestras de sonido. ¿Podemos usar la misma función para aumentar y reducir el volumen? Muestre los comandos que ejecutaría para realizar cada acción.

6.13 En la sección 6.3.2 vimos cómo funcionaba el programa 56 paso a paso. Dibuje las imágenes para mostrar cómo funciona el programa 57 (página 166) en la misma forma.

6.14 ¿Qué ocurre si aumenta demasiado un volumen? Para explorar esto, cree un objeto de sonido, luego aumente el volumen una vez, luego otra y otra. ¿Acaso siempre seguirá haciéndose más fuerte? ¿O pasa alguna otra cosa? ¿Puede explicar por qué?

6.15 Pruebe a esparcir algunos valores específicos en sus sonidos. ¿Qué ocurre si establece el valor de unos cuantos cientos de muestras a la mitad de un sonido a 32767? ¿O unos cuantos cientos a –32768? ¿O unos cuantos ceros? ¿Qué ocurre con el sonido?

6.16 En vez de multiplicar muestras por un multiplicador (como 2 o 0.5), trate de *sumarles* un valor. ¿Qué ocurre con un sonido si suma 100 a todas las muestras? ¿Y si suma 1000?

## PARA PROFUNDIZAR

Existen muchos libros maravillosos sobre psicoacústica y música de computadora. Uno de los favoritos de Mark por su facilidad de comprensión es *Computer Music: Synthesis, Composition, and Performance* por Dodge y Jerse [11]. La *biblia* de la música de computadora es *The Computer Music Tutorial* de Curtis Roads [35].

Cuando utiliza la aplicación de MediaTools, en realidad usa un lenguaje de programación llamado *Squeak*, desarrollado en un principio y principalmente por Alan Kay, Dan Ingalls, Ted Kaehler, John Maloney y Scott Wallace [25]. Squeak ahora es de código fuente abierto[2] y una excelente herramienta multimedia multiplataforma. Hay un libro que introduce Squeak incluyendo sus herramientas de sonido [19], además de otro libro sobre Squeak [20] que incluye un capítulo sobre *Siren*, una variación de Squeak por Stephen Pope, diseñada en especial para exploración y composición de música de computadora.

[2] `http://www.squeak.org`

CAPÍTULO 7

# Modificación de muestras en un rango

7.1 MANIPULACIÓN DE SECCIONES DIFERENTES DEL SONIDO EN FORMA DISTINTA
7.2 EMPALME DE SONIDOS
7.3 RECORTE Y COPIA EN GENERAL
7.4 INVERSIÓN DE SONIDOS
7.5 REFLEJO
7.6 SOBRE LAS FUNCIONES Y EL ALCANCE

### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Empalmar sonidos para crear composiciones.
- Invertir sonidos.
- Reflejar sonidos.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Iterar una variable índice para un arreglo a través de un rango.
- Usar comentarios en los programas y entender por qué.
- Identificar algunos algoritmos que cruzan los límites de los medios.
- Describir y usar el alcance con más cuidado.

## 7.1 MANIPULACIÓN DE SECCIONES DIFERENTES DEL SONIDO EN FORMA DISTINTA

En el último capítulo describimos algunas cosas útiles de hacer con los sonidos en general, pero los efectos que son de verdad interesantes surgen al cortar los sonidos y manipular cada pieza de manera distinta: algunas palabras por aquí, otros sonidos por allá. ¿Cómo podemos hacer eso? Necesitamos la capacidad de iterar a través de *porciones* de la muestra, sin recorrer todo. Esto es fácil de hacer, pero necesitamos manipular las muestras de una manera algo distinta (por ejemplo, tenemos que usar números de índice) y tenemos que usar nuestro ciclo `for` de una forma un poco distinta.

Si recuerda, cada muestra tiene un número asociado (un *índice*) que podemos considerar como una dirección, o una descripción de la posición de esa muestra en el sonido en general. Podemos obtener cada muestra individual con `getSampleValueAt` (con un sonido y un

número de índice como entrada). Podemos establecer cualquier muestra con `setSampleValueAt` (con entradas de un sonido, un número de índice y un nuevo valor). Así es como podemos manipular muestras sin usar getSamples y los objetos de muestra. Pero de todas formas no queremos escribir código como el siguiente:

```
setSampleValueAt(sonido,1,12)
setSampleValueAt(sonido,2,28) ...
```

¡No para decenas de miles de muestras!

Mediante el uso de `range` podemos hacer todo lo que hacemos con `getSamples`, pero ahora podemos hacer referencia directa a los números de índice. Aquí está el programa 56 (página 161) modificado para usar `range`.

**Programa 61: aumentar el volumen de un sonido de entrada mediante range**

```
def aumentarVolumenPorRango(sonido):
  for indiceMuestra in range(0,getLength(sonido)):
    valor = getSampleValueAt(sonido,indiceMuestra)
    setSampleValueAt(sonido,indiceMuestra,valor * 2)
```

■

Pruébelo y verá que funciona justo igual que el programa anterior.

Pero ahora podemos hacer cosas realmente divertidas con los sonidos, ya que podemos controlar con cuáles muestras vamos a trabajar. El siguiente programa *aumenta* el sonido para la primera mitad del mismo, y luego lo *reduce* en la segunda mitad. Vea si puede rastrear cómo funciona.

**Programa 62: aumentar el volumen y luego reducirlo**

```
def aumentarYReducir(sonido):
  for indiceMuestra in range(0,getLength(sonido)/2):
    valor = getSampleValueAt(sonido,indiceMuestra)
    setSampleValueAt(sonido,indiceMuestra,valor * 2)
  for indiceMuestra in range(getLength(sonido)/2, getLength(sonido)):
    valor = getSampleValueAt(sonido,indiceMuestra)
    setSampleValueAt(sonido,indiceMuestra,valor * 0.2)
```

■

### Cómo funciona

Hay dos ciclos en `aumentarYReducir`, cada uno de los cuales se encarga de una mitad del sonido.

- El primer ciclo lidia con las muestras desde 0 hasta la mitad del sonido. Estas muestras se multiplican por 2 para duplicar su amplitud.
- El segundo ciclo va desde la mitad hasta el final del sonido. Aquí multiplicamos cada muestra por 0.2, con lo cual se reduce el volumen en un 80%.

### Otra forma de escribir referencias a arreglos

Vale la pena señalar que en muchos lenguajes, los corchetes (`[ ]`) son la notación estándar para acceder a los elementos de los arreglos. En Python funciona de esta manera. Para

cualquier arreglo, `arreglo[indice]` devuelve el i-ésimo elemento en el arreglo. El número dentro de los corchetes es siempre una variable índice, pero algunas veces se conoce como *subíndice* debido a la forma en que los matemáticos hacen referencia al *i*-ésimo elemento de *a*; por ejemplo, $a_i$.

Usaremos algunos ejemplos aquí para demostrarlo.

```
>>> archivo = "C:/ip-book/mediasources/a.wav"
>>> sonido = makeSound(archivo)
>>> muestras = getSamples(sonido)
>>> print muestras[0]
Sample at 0 with value -90
>>> print muestras[1]
Sample at 1 with value -113
>>> print getLength(sonido)
9508
>>> muestras[9507]
Sample at 9507 with value -147
>>> muestras[9508]
The error was: 9508
Sequence index out of range.
The index you're using goes beyond the size of that
data (too low or high).
For instance, maybe you tried to access OurArray[10]
and OurArray only has 5 elements in it.
```

El primer elemento en un arreglo está en el índice 0. El último elemento en un arreglo está en la longitud del arreglo menos 1. Observe lo que ocurre si hacemos referencia a un índice *más allá* del final del arreglo. Obtenemos un error indicando que el "índice de la secuencia está fuera de rango".

Vamos a usar `range` para crear un arreglo[1] y luego referenciarlo de la misma forma.

```
>>> miArreglo = range(0,100)
>>> print miArreglo[0]
0
>>> print miArreglo[1]
1
>>> print miArreglo[99]
99
>>> miSegundoArreglo = range(0,100,2)
>>> print miSegundoArreglo[35]
70
```

El código `range(0, 100)` crea un arreglo[2] que contiene 100 elementos. El primer elemento está en el índice 0 y el último en el índice 99. El arreglo contiene todos los valores desde 0 hasta 99. Recuerde que cuando especificamos un rango mediante `range(inicio, fin)`, el número `inicio` está en el arreglo devuelto pero el número `fin` no.

Si usamos un número de índice *debajo* del final del arreglo (es decir, un número negativo), Python calculará el índice como una desviación desde el final del arreglo. Observe que la referencia a `miArreglo[-1]` que se muestra a continuación es la misma que la referencia a `miArreglo[99]` anterior. Cualquier referencia *por encima* del final del arreglo genera el mismo error que vimos antes.

[1]Técnicamente una *secuencia*, que puede indexarse como un arreglo pero no tiene todas sus características.
[2]Técnicamente ésta es una *secuencia* de Python, pero el comportamiento es el mismo para nuestros fines.

```
>>> miArreglo[-1]
99
>>> miArreglo[100]
The error was: 100
Sequence index out of range.
The index you're using goes beyond the size of that data
    (too low or high). For instance, maybe you tried to
    access OurArray[10] and OurArray only has 5 elements
    in it.
>>> miArreglo[101]
The error was: 101
Sequence index out of range.
The index you're using goes beyond the size of that data
    (too low or high). For instance, maybe you tried to
    access OurArray[10] and OurArray only has 5 elements
    in it.
```

## 7.2 EMPALME DE SONIDOS

El *empalme de sonidos* es un término que se remonta a la época en la que los sonidos se grababan en cinta; para cambiar el orden de las cosas en la cinta había que cortarla en segmentos y después pegarla en el orden correcto. A esto se le conoce como "empalmar". Cuando todo es digital, es *mucho más* sencillo.

Para empalmar sonidos, tan sólo tenemos que copiar elementos y moverlos en el arreglo. Es más fácil hacer esto con dos (o más) arreglos, en vez de realizar la copia dentro del mismo arreglo. Si copia todas las muestras que representan a alguien diciendo la palabra "el" al principio de un sonido (empezando en el índice número 0), entonces hará que el sonido empiece con la palabra "el". Los empalmes le permiten crear todo tipo de sonidos, discursos, tonterías y arte.

El tipo más sencillo de empalme es cuando los sonidos están en archivos separados. Todo lo que necesita hacer es copiar cada sonido, en orden, en un sonido de destino. He aquí un programa que crea el inicio de una oración en inglés: "Guzdial is..." (invitamos a los lectores a completar la oración).

**Programa 63: mezclar palabras en una sola oración**

```
def mezclar():
  sonidoGuzdial = makeSound(getMediaPath("guzdial.wav"))
  sonidoIs = makeSound(getMediaPath("is.wav"))
  destino = makeSound(getMediaPath("sec3silence.wav"))
  indice = 0
  # Copiar "Guzdial"
  for origen in range(0,getLength(sonidoGuzdial)):
    valor = getSampleValueAt(sonidoGuzdial,origen)
    setSampleValueAt(destino,indice,valor)
    indice = indice + 1
  # Copiar pausa de 0.1 segundos (silencio) (0)
  for origen in range(0,int(0.1*getSamplingRate(destino))):
    setSampleValueAt(destino,indice,0)
    indice = indice + 1
  # Copiar "is"
```

```
for origen in range(0,getLength(sonidoIs)):
  valor = getSampleValueAt(sonidoIs,origen)
  setSampleValueAt(destino,indice,valor)
  indice = indice + 1
normalizar(destino)
play(destino)
return destino
```

■

### Cómo funciona

Hay tres ciclos en la función `mezclar`, cada uno de los cuales copia un segmento en el sonido de destino; un segmento puede ser una palabra o un silencio entre palabras.

- La función empieza por crear objetos de sonido para la palabra "Guzdial" (`sonidoGuzdial`), la palabra "is" (`sonidoIs`) y el silencio de destino (`destino`).
- Tenga en cuenta que hicimos `indice` (para el destino) igual a 0 antes del primer ciclo. Luego lo aumentamos en cada ciclo, pero nunca lo volvemos a establecer a un valor específico. Esto se debe a que `indice` siempre es el índice para la *siguiente muestra vacía* en el sonido de destino. Como cada ciclo sigue del anterior, sólo seguimos pegando muestras al final del destino.
- En el primer ciclo copiamos todas y cada una de las muestras de `sonidoGuzdial` en el `destino`. Hacemos que el índice origen vaya desde 0 hasta la longitud de `sonidoGuzdial`. Obtenemos el valor de la muestra en `origen` de `sonidoGuzdial` y luego establecemos el valor de muestra en `indice` en el sonido `destino` al valor que obtuvimos de `sonidoGuzdial`. Luego aumentamos `indice` de modo que apunte al siguiente índice de la muestra vacía.
- En el segundo ciclo creamos 0.1 segundos de silencio. Como `getSamplingRate(destino)` nos proporciona el número de muestras en 1 segundo de `destino`, 0.1 veces nos indica el número de muestras en 0.1 segundos. Aquí no obtenemos ningún valor de origen; tan sólo establecemos la i-ésima muestra a 0 (para silencio) y luego aumentamos el `indice`.
- Por último copiamos todas las muestras de `sonidoIs`, justo igual que en el primer ciclo en donde copiamos `sonidoGuzdial`.
- Ahora normalizamos el sonido con la función `normalizar` para hacerlo más fuerte. Esto significa que la función `normalizar` *debe* estar en el área del programa junto con `mezclar`, aun y cuando no la mostramos aquí. Luego usamos `play` para reproducir el sonido y `return` para devolverlo.

  Devolvemos el sonido de destino usando `return destino` debido a que creamos el sonido en la función. El sonido de destino no se entregó como entrada para la función. Si no regresamos el sonido que creamos en `mezclar`, desaparecería al terminar el contexto de la función `mezclar` (*alcance*). Al regresarlo, permitimos que otra función use el sonido resultante.

En vez de usar un sonido de silencio de 3 segundos como el destino, podemos crear un sonido vacío de una longitud específica mediante la función `makeEmptySound(longitudEnMuestras)`. Por ejemplo, podemos crear un sonido que sea lo bastante extenso como para dar cabida a lo que queremos copiar en él, si hacemos lo siguiente:

```
longGuz = getLength(sonidoGuzdial)
longSilencio = int(0.1 * 22050)
longIs = getLength(sonidoIs)
destino = makeEmptySound(longGuz + longSilencio + longIs)
```

La velocidad de muestreo predeterminada para un nuevo sonido es de 22050 muestras por segundo. Entonces, si queremos una décima de segundo de silencio entre las palabras, la longitud de eso en muestras es (0.1 * 22050). Debemos convertir este valor de vuelta en un entero mediante `int(0.1 * 22050)`. También podemos crear un nuevo sonido de silencio usando `makeEmptySound(longitud,velocidadMuestreo)`.

El tipo más común de empalme es cuando las palabras están en medio de un sonido existente y necesitamos extraerlas de ahí. Lo primero por hacer en un empalme así es averiguar los números de índice que delimiten las piezas en las que estamos interesados. Si usamos la herramienta de sonido de JES, es algo muy fácil de hacer.

- Abra su archivo WAV en la herramienta de sonido de JES. Para ello, puede crear el sonido y explorarlo.
- Desplácese y mueva el cursor (arrastrándolo por el gráfico) hasta que crea que el cursor está antes o después de un sonido de interés.
- Verifique su posición reproduciendo el sonido antes y después del cursor, usando los botones en la herramienta de sonido.

Mark usó este proceso para encontrar los puntos finales de las primeras palabras en **preamble10.wav** (figura 7.1). Asumió que la primera palabra empezaba en el índice 0.

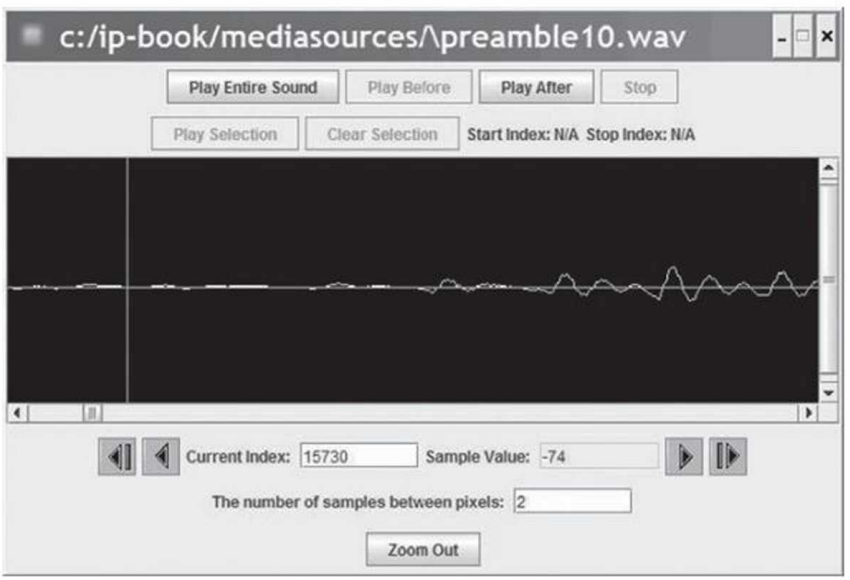



**FIGURA 7.1**
Explorar un sonido para encontrar el silencio entre las palabras.

| Palabra | Índice final |
|---|---|
| We | 15730 |
| the | 17407 |
| People | 26726 |
| of | 32131 |
| the | 33413 |
| United | 40052 |
| States | 55510 |

Para escribir un ciclo que copie cosas de un arreglo a otro se requiere un poco de manipulación. Es necesario llevar la cuenta de dos índices: la posición en el arreglo *desde* donde vamos a copiar y la posición en el arreglo *hacia* dónde vamos a copiar. Éstas son dos variables diferentes, que rastrean dos índices distintos. Pero ambas se incrementan de la misma forma.

La manera en que haremos esto (una *subreceta* o *subprograma*) será usar una variable índice para apuntar en la entrada correcta en el arreglo de *destino* (*hacia* el que vamos a copiar), usar un ciclo `for` para que la segunda variable índice se desplace por las entradas correctas en el arreglo de *origen* cada vez que realicemos una copia y (*¡muy importante!*) mover la variable índice de destino cada vez que realicemos una copia. Eso es lo que mantiene sincronizadas a las dos variables índice.

Para mover el índice de destino, hay que sumarle 1. Esto es muy sencillo: vamos a indicar a Python que realice la operación `indiceDestino = indiceDestino + 1`. Recuerde que esto cambia el valor de `indiceDestino` al valor actual *más* 1, con lo cual se mueve (aumenta) el índice de destino. Si colocamos esto en el cuerpo del ciclo en donde vamos a cambiar el índice de origen, haremos que se muevan en sincronía.

La forma general del subprograma es:

```
indiceDestino = En-donde-debería-iniciar-el-sonido-entrante
for indiceOrigen in range(puntoInicial,puntoFinal)
  setSampleValueAt(destino, indiceDestino,
    getSampleValueAt(origen, indiceOrigen))
  indiceDestino = indiceDestino + 1
```

A continuación veremos el programa que cambia el preámbulo de "We the people of the United States" a "We the *united* people of the United States".

**Programa 64: empalme del preámbulo para incluir "united"**

Asegúrese de cambiar la variable `archivo` antes de probar esto en su computadora.

```
# Empalmes
# Uso del sonido de preámbulo,
#  crear "We the united people"
def empalmarPreambulo():
    archivo = "C:/ip-book/mediasources/preamble10.wav"
    origen = makeSound(archivo)
    # Éste será el nuevo sonido empalmado
    destino = makeSound(archivo)
```

```
    # indiceDestino empieza justo antes de
    # "We the" en el nuevo sonido
    indiceDestino=17408
    # En dónde está la palabra "United" en el sonido
    for indiceOrigen in range(33414, 40052):
        valor = getSampleValueAt(origen, indiceOrigen)
        setSampleValueAt(destino, indiceDestino, valor)
        indiceDestino = indiceDestino + 1

    # En dónde está la palabra "People" en el sonido
    for indiceOrigen in range(17408, 26726):
        valor = getSampleValueAt(origen, indiceOrigen)
        setSampleValueAt(destino, indiceDestino, valor)
        indiceDestino = indiceDestino + 1

    # Pegar cualquier espacio silencioso después de eso
    for indice in range(0,1000):
        setSampleValueAt(destino, indiceDestino,0)
        indiceDestino = indiceDestino + 1

    # Escuchemos y devolvamos el resultado
    play(destino)
    return destino
```

■

Usamos esta función de la siguiente manera:

```
>>> nuevoSonido=empalmarPreambulo()
```

### Cómo funciona

En este programa suceden muchas cosas. Vamos a recorrerlo lentamente.

Tenga en cuenta que hay muchas líneas que empiezan con "#". Este símbolo significa que lo que viene después de ese carácter en la línea es una nota para el programador y *Python debe ignorarla.* Se conoce como un *comentario.*

**Tip de funcionamiento: ¡los comentarios son buenos!**

Los comentarios son excelentes formas de explicar lo que hacemos a los demás, ¡incluso a usted mismo! La realidad es que es difícil recordar todos los detalles de un programa, por lo que a menudo es *muy* útil dejar notas sobre lo que hicimos en caso de que alguna vez volvamos a usar el programa, o si alguien más está tratando de comprenderlo.

■

La función `empalmarPreambulo` no recibe parámetros. Sin duda sería estupendo escribir una función que pudiera realizar cualquier tipo de empalme que necesitemos, de la misma forma como las funciones generalizadas para aumentar volumen y para normalizar. Pero ¿cómo hacemos esto? ¿Cómo generalizamos todos los puntos iniciales y finales? Es más fácil, cuando menos al principio, crear programas individuales que manejen tareas específicas de empalmes.

Aquí podemos ver tres de los ciclos de copia como los que establecimos antes. En realidad sólo hay dos. El primero copia la palabra "united" en su posición. El segundo copia la

palabra "people" en su posición. "Pero espera", tal vez esté usted pensando. "¡La palabra 'people' *ya estaba* en el sonido!" Eso es cierto, pero cuando copiamos "united" sobrescribimos parte de la palabra "people", por lo que volvimos a copiarla.

Al final del programa regresamos el sonido de `destino`. Este sonido se creó en la función, no se recibe como entrada. Si no lo regresamos no podríamos hacer referencia a él de nuevo. Al regresarlo es posible darle un nombre y reproducirlo (e incluso manipularlo todavía más) después de que la función termine su ejecución.

He aquí una forma más simple. Pruebe el programa y escuche el resultado:

```
def empalmarMasSimple():
  archivo = "C:/ip-book/mediasources/preamble10.wav"
  origen = makeSound(archivo)
  # Éste será el nuevo sonido empalmado
  destino = makeSound(archivo)
  # indiceDestino empieza justo después de "We the" en el nuevo sonido
  indiceDestino=17408
  # En dónde está la palabra "United" en el sonido
  for indiceOrigen in range(33414, 40052):
    valor = getSampleValueAt(origen, indiceOrigen)
    setSampleValueAt(destino, indiceDestino, valor)
    indiceDestino = indiceDestino + 1
  # Escuchemos y devolvamos el resultado
  play(destino)
  return destino
```

Veamos si podemos averiguar lo que sucede en el sentido matemático. Recordemos la tabla que está en la página 180. Vamos a empezar insertando muestras en el índice 17408. La palabra "united" tiene (40052 – 33414) 6638 muestras (ejercicio para el lector: ¿cuánto es eso en segundos?). Esto significa que escribiremos en el destino desde 17408 hasta (17408 +6638) el índice de muestra 24046. Sabemos de la tabla que la palabra "people" termina en el índice 26726. Si la palabra "people" tiene más de (26726 – 24046) 2680 muestras, entonces empezará antes de 24046 y nuestra inserción de "united" invadirá una parte. Si la palabra "united" tiene más de 6000 muestras, dudamos que la palabra "people" tenga menos de 2000. Esto explica por qué suena quebrado. ¿Por qué funciona con la parte en donde está "of"? Tal vez el orador se haya detenido ahí. Si revisa la tabla de nuevo, verá que la palabra "of" termina en el índice de muestra 32131 y la palabra anterior a esta termina en 26726. La palabra "of" ocupa menos de (32131 – 26726) 5405 muestras, razón por la cual funciona el programa original.

El tercer ciclo en el programa 64 original (página 180) se ve igual que el mismo tipo de ciclo de copia, pero en realidad sólo coloca unos cuantos ceros. Las muestras con un valor de 0 representan el silencio. Al colocar unos cuantos se crea una pausa que suena mejor (hay un ejercicio que sugiere extraer las pausas y ver cómo se escucha).

La figura 7.2 muestra el archivo **preamble10.wav** original en el editor de sonido superior y el nuevo sonido empalmado (el que se guardó con `writeSoundTo`) en la parte inferior. Las líneas se dibujan de modo que la sección empalmada esté entre ellas, mientras que el resto de los sonidos son idénticos.

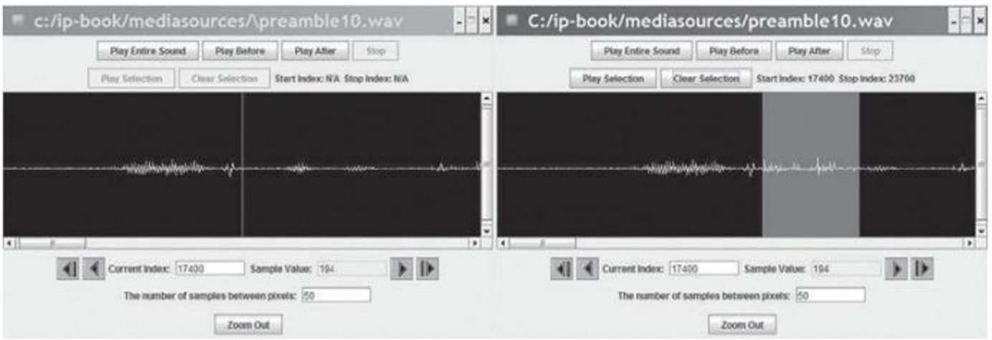



**FIGURA 7.2**
Comparación del sonido original (*izquierda*) con el sonido empalmado (*derecha*).

## 7.3 RECORTE Y COPIA EN GENERAL

Las funciones anteriores eran un poco complicadas. ¿Qué las podría hacer más sencillas? Sería agradable tener un método de recorte general que tomara un sonido, un índice inicial y final, y regresara un nuevo recorte de sonido con sólo esa parte del sonido original. Entonces sería fácil crear un recorte que sólo contuviera la palabra "united".

**Programa 65: crear un recorte de sonido**

```
def recortar(origen,inicio,fin):
  destino = makeEmptySound(fin - inicio)
  indiceDestino = 0
  for indiceOrigen in range(inicio,fin):
    valorOrigen = getSampleValueAt(origen,indiceOrigen)

    setSampleValueAt(destino,indiceDestino,valorOrigen)
    indiceDestino = indiceDestino + 1
  return destino
```

■

Ahora podemos crear un recorte de sonido que sólo contenga la palabra "united", haciendo lo siguiente:

```
>>> preambulo = makeSound(getMediaPath("preamble10.wav"))
>>> explore(preambulo)
>>> united = recortar(preambulo,33414,40052)
>>> explore(united)
```

También sería bueno tener un método de copia general que reciba un sonido de origen y un sonido de destino, y que copie todo el origen en el destino en una ubicación inicial en el destino que se pase como parámetro.

**Programa 66: copia general**

```
def copiar(origen,destino,inicio):
  indiceDestino = inicio
  for indiceOrigen in range(0,getLength(origen)):
    valorOrigen = getSampleValueAt(origen,indiceOrigen)
    setSampleValueAt(destino,indiceDestino,valorOrigen)
    indiceDestino = indiceDestino + 1
```

■

Ahora podemos insertar "united" en el preámbulo de nuevo, usando las nuevas funciones.

**Programa 67: uso de las funciones recortar y copiar generales**

```
def crearNuevoPreambulo():
  archivo = "C:/ip-book/mediasources/preamble10.wav"
  preambulo = makeSound(archivo)
  united = recortar(preambulo,33414,40052)
  inicio = recortar(preambulo,0,17407)
  fin = recortar(preambulo,17408,55510)
  longitud = getLength(inicio) + getLength(united) + getLength(fin)
  nuevoPre = makeEmptySound(longitud)
  copiar(inicio,nuevoPre,0)
  copiar(united,nuevoPre,17407)
  copiar(fin,nuevoPre,getLength(inicio) + getLength(united))
  return nuevoPre
```

■

Observe cómo esta función llama a las nuevas funciones generales `recortar` y `copiar`. Puede colocar las tres en el mismo archivo. Sería bueno si pudiera tener un archivo de funciones de sonido generales que otras funciones pudieran usar, sin necesidad de tener todas las funciones en el mismo archivo.

Para ello puede *importar* funciones de otros archivos. Tendrá que agregar `from media import *` como la primera línea en el archivo con las funciones de sonido generales para poder usar las funciones multimedia que JES proporciona, como `getMediaPath o getRed`. JES importa las funciones de medios de manera automática por usted, pero como no vamos a cargar este archivo general por medio de JES, debemos importar de manera explícita las funciones de medios. Al archivo de funciones generales de sonido le llamaremos **miSonido.py** para que no cause conflicto con otras cosas en JES que utilicen el nombre Sonido.

```
from media import *

def recortar(origen,inicio,fin):
  destino = makeEmptySound(fin - inicio)
  indiceDestino = 0
  for indiceOrigen in range(inicio,fin):
    valorOrigen = getSampleValueAt(origen,indiceOrigen)
    setSampleValueAt(destino,indiceDestino,valorOrigen)
    indiceDestino = indiceDestino + 1
  return destino
```

```
def copiar(origen,destino,inicio):
  indiceDestino = inicio
  for indiceOrigen in range(0,getLength(origen)):
    valorOrigen = getSampleValueAt(origen,indiceOrigen)
    setSampleValueAt(destino,indiceDestino,valorOrigen)
    indiceDestino = indiceDestino + 1
```

Para usar las nuevas funciones generales de sonido, debe utilizar primero `setLibPath(ruta)`. Esta función indica a Python en dónde buscar los archivos que va a importar. La ruta es el nombre de ruta completo que indica el directorio con el archivo que contiene las funciones generales.

```
>>> setLibPath("c:/ip-book/programas/")
```

Para usar las funciones generales de sonido copiar y pegar en otro archivo, agregamos `from miSonido import *` como la primera línea en el archivo. La instrucción `from miSonido import *` es como copiar las funciones generales en el mismo archivo que `crearNuevoPreambulo`, sólo que en realidad es mejor que copiar las funciones. Si copiamos las funciones generales a muchos archivos y *después* modificamos esas funciones, tendremos que cambiarlas en todos los lugares en donde las copiamos. Pero, si importamos funciones generales y después modificamos el archivo de la función general, las funciones mejoradas serán las que se utilicen en todas partes.

```
from miSonido import *

def crearNuevoPreambulo():
  archivo = "C:/ip-book/mediasources/preamble10.wav"
  preambulo = makeSound(archivo)
  united = recortar(preambulo,33414,40052)
  inicio = recortar(preambulo,0,17407)
  fin = recortar(preambulo, 17408, 55510)
  longitud = getLength(inicio) + getLength(united) + getLength(fin)
  nuevoPre = makeEmptySound(longitud)
  copiar(inicio,nuevoPre,0)
    copiar(united,nuevoPre,17407)
    copiar(fin,nuevoPre,getLength(inicio) + getLength(united))
    return nuevoPre
```

Las funciones generales son también una forma de *abstracción* como la función general `copiar` que usamos con las imágenes. Al crear un archivo de funciones generales que importamos, permitimos a otros (incluso a nosotros, si se nos olvida) usar la abstracción sin tener que comprender los detalles acerca de cómo se implementan estas funciones. Así es como hemos estado usando `getRed` y `getMediaPath` sin necesidad de conocer los detalles sobre cómo funcionan.

**Tip de funcionamiento: un directorio de funciones generales**

Si coloca todos los archivos que contienen funciones generales en el mismo directorio, puede usar `setLibPath` una vez para tener acceso a todos los archivos de funciones generales mediante el uso de `import`. Puede tener archivos de funciones generales para sonidos, imágenes, películas, etcétera.

■

## 7.4 INVERSIÓN DE SONIDOS

En el ejemplo de empalme, copiamos las muestras de las palabras justo como estaban en el sonido original. No siempre tenemos que ir en el mismo orden. Podemos invertir las palabras, o hacerlas más rápidas, lentas, fuertes o suaves. Como ejemplo, he aquí un programa que invierte un sonido (figura 7.3).

**Programa 68: reproducir el sonido dado al revés (invertirlo)**

```
def invertir(origen):
  destino = makeEmptySound(getLength(origen))
  indiceOrigen = getLength(origen)-1
  for indiceDestino in range(0,getLength(destino)):
    valorOrigen = getSampleValueAt(origen,indiceOrigen)
    setSampleValueAt(destino,indiceDestino,valorOrigen)
    indiceOrigen = indiceOrigen - 1
  return destino
```

■

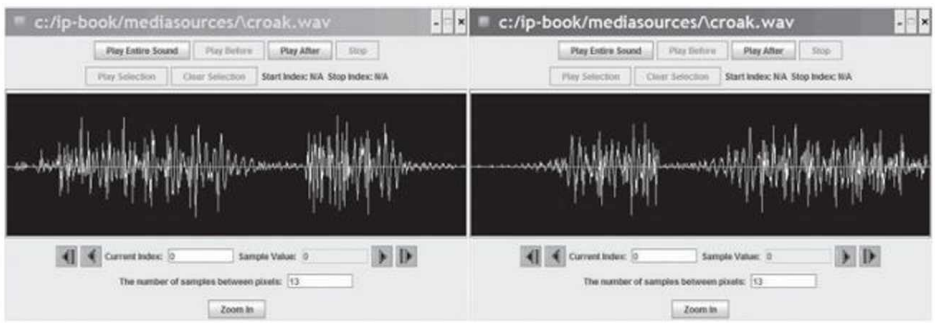



**FIGURA 7.3**
Comparación del sonido original (*izquierda*) con el sonido invertido (*derecha*).

Para probar el programa anterior, escriba lo siguiente en el área de comandos:

```
>>> croak = makeSound(getMediaPath("croak.wav"))
>>> explore(croak)
>>> croakInv = invertir(croak)
>>> explore(croakInv)
```

### Cómo funciona

Este programa usa otra variante del subprograma que copia elementos de un arreglo que ya vimos antes.

- Primero crea el sonido `destino` como un sonido vacío con la misma longitud que el sonido `origen`.
- Inicia el `indiceOrigen` al *final* del arreglo, en vez de hacerlo al principio (la longitud menos 1).

- El `indiceDestino` se mueve desde 0 hasta la longitud menos 1, tiempo durante el cual el programa
  - obtiene el valor de muestra en el `origen` en el `indiceOrigen`:
  - copia ese valor en el destino en el `indiceDestino:` y
  - *reduce* el `indiceOrigen` por 1, lo que significa que el `indiceOrigen` se mueve desde el final del arreglo, hacia atrás, hasta el principio.

## 7.5 REFLEJO

Una vez que sabemos cómo reproducir sonidos hacia delante y hacia atrás, *reflejar* un sonido es el mismo proceso exacto que reflejar una imagen (figura 7.4). Compare esto con el programa 20 (página 79). ¿Está de acuerdo en que éste es el mismo *algoritmo*, aun y cuando estamos lidiando con un medio diferente?

**Programa 69: reflejar un sonido de adelante hacia atrás**

```
def reflejarSonido(sonido):
  longitud = getLength(sonido)
  puntoreflejo = longitud/2
  for indice in range(0,puntoreflejo):
    izquierda = getSampleObjectAt(sonido,indice)
    derecha = getSampleObjectAt(sonido,longitud-indice-1)
    valor = getSampleValue(izquierda)
    setSampleValue(derecha,valor)
```

■

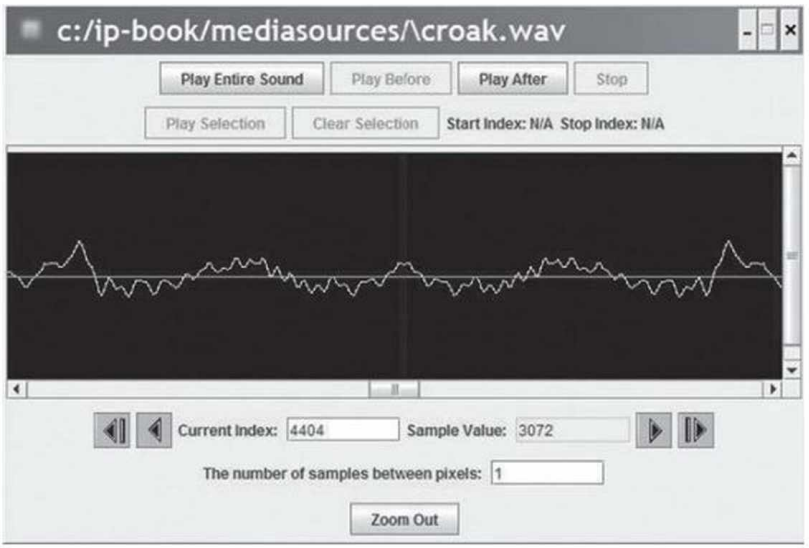



**FIGURA 7.4**
Reflejar un sonido de adelante (*izquierda*) hacia atrás (*derecha*).

## 7.6 SOBRE LAS FUNCIONES Y EL ALCANCE

Los parámetros de las funciones y el alcance pueden ser confusos. El mismo *nombre* puede tener diferentes significados (es decir, distintos valores), dependiendo de si el nombre se crea dentro de la función o fuera de la misma. Imagine que va a usar este conjunto (estúpido, sin significado) de funciones en JES (figura 7.5):

```
def fun1(a):
  print a
  a = 12
  c = "George"
  print a,c

def fun2(b):
  print b
  b = 13
  print b,c
```

Ahora vamos a definir en el área de comandos tres variables: a, b y c.

```
>>> a = "Hola"
>>> b = "Adiós"
>>> c = "Mark"
>>> print a,b,c
Hola Adiós Mark
```

A continuación, vamos a llamar a fun1 con b como entrada.

```
>>> fun1(b)
Adiós
12 George
>>> print a,b,c
Hola Adiós Mark
```

Cuando llamamos a la función fun1, el valor de b del área de comandos se conoce como a. El nombre a *dentro* de fun1 recibirá una copia del valor de b; esto es, "Adiós". **La variable de parámetro a en fun1 no tiene nada que ver con la variable a en el área de comandos.** Ésta tiene su propio *alcance*.

Al cambiar a dentro de fun1, sólo cambiamos el parámetro, que es una variable *local*. Por lo tanto, fun1 imprimirá el mensaje "Adiós". Luego asignamos la variable c, que es *local* para la función fun1. Una variable local sólo existe dentro del *alcance* de la función. Al asignar un valor a c no cambiamos **nada** con respecto a las variables en el área de comandos. Desde el interior de fun1 imprimimos a y c, y vemos "12 George". Pero de vuelta en el área de comandos, cuando imprimimos a, b y c vemos (de nuevo) "Hola Adiós Mark".

Ahora, ¿qué ocurre si llamamos a fun1 con a, el mismo nombre que el parámetro? Ocurre algo muy parecido, excepto que imprimimos "Hola" en vez de "Adiós". La variable a dentro de fun1 es una variable totalmente distinta de la que tiene el mismo nombre en el área de comandos.

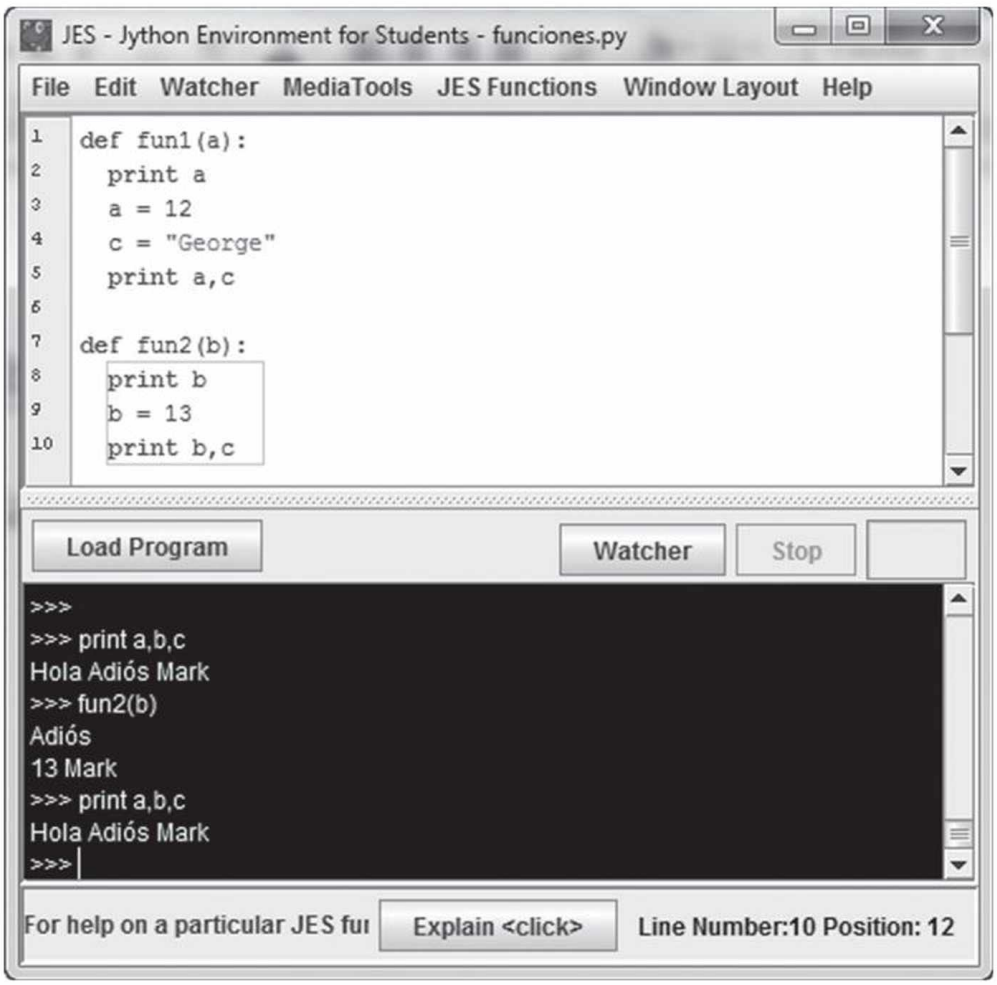



**FIGURA 7.5**
Dos funciones de muestra para explorar parámetros y alcances.

```
>>> print a,b,c
Hola Adiós Mark
>>> fun1(a)
Hola
12 George
>>> print a,b,c
Hola Adiós Mark
```

¿Qué hay sobre `fun2`? Recibe `b` como entrada. Si le proporcionamos la `b` del área de comandos, de todas formas no podremos modificar la `b` en el área de comandos, ya que el parámetro es *local* para la función.

```
>>> print a,b,c
Hola Adiós Mark
>>> fun2(b)
Adiós
13 Mark
>>> print a,b,c
Hola Adiós Mark
```

Observe que `fun2` *hace referencia* a la variable `c`. No asigna a `c`, y no tiene a `c` como parámetro. ¿Qué ocurre cuando ejecutamos la instrucción `print b,c`? La variable `c` será la variable del área de comandos, ya que no hay una copia local dentro de `fun2`. Imprimimos el parámetro `b` y la variable `c` del área de comandos, por lo que imprimimos "13 Mark". Decimos que el alcance de `fun2` está *dentro* del alcance del área de comandos. Si no definimos una variable local con el mismo nombre, podemos acceder a las variables del alcance externo. Podemos considerar que el alcance del área de comandos es *global* y abarca todas las funciones. Si dentro de una función definimos una variable local con el mismo nombre, la variable local redefine a la variable global, bloqueando efectivamente el acceso.

He aquí las tres ideas que debemos cosechar de todo esto:

- Cuando se invoca una función, los valores de entrada se copian en las variables de parámetros.[3] Si modificamos las variables de parámetros no cambia la variable de entrada, ni cambian las variables en otros alcances.
- Todas las variables que son locales (incluyendo a las variables de parámetros) desaparecen al terminar la función. Si modificamos las variables locales no cambian las variables en otros alcances.
- Podemos hacer referencia a las variables en el alcance externo de una función, pero sólo si no tenemos una variable local con el mismo nombre.

## RESUMEN DE PROGRAMACIÓN

En este capítulo hablamos sobre varios tipos de codificaciones de datos (u objetos).

| | |
|---|---|
| Sonidos | Codificaciones de sonidos, por lo general provenientes de un archivo WAV. |
| Muestras | Colecciones de objetos de muestras, cada uno indexado por un número (por ejemplo, muestra #0, muestra #1). `muestras[0]` es el primer objeto de muestra. Puede manipular cada muestra dentro de las muestras de la siguiente manera: `for s in muestras:`. |
| Muestra | Un valor entre –32 000 y 32 000 (aproximadamente) que representa el voltaje que generaría un micrófono en un instante dado al grabar un sonido. Por lo general la longitud del instante es 1/44 100 de un segundo (para el sonido con calidad de CD) o de 1/22 050 de un segundo (para un sonido con calidad suficiente en la mayoría de las computadoras). Un objeto muestra recuerda de qué sonido proviene, por lo que si modificamos su valor sabe cómo regresar y modificar la muestra correcta en el sonido. |

[3] Si el valor de entrada es un objeto, la variable de parámetro se convierte en un alias: otro nombre para ese objeto.

Éstas son las funciones utilizadas e introducidas en este capítulo:

| | |
|---|---|
| `range` | Recibe dos números y devuelve un arreglo de enteros, empezando en el primer número y deteniéndose antes del último número. |
| `range` | También puede recibir tres números y entonces devuelve un arreglo de todos los enteros desde el primero hasta, pero sin incluir, el segundo, usando un incremento indicado por el tercer número. |

## PROBLEMAS

7.1 Vuelva a escribir el programa 62 (página 175) de modo que se proporcionen dos valores a la función: el sonido y un *porcentaje* de qué tanto hay que avanzar en el sonido antes de aumentar y reducir el volumen.

7.2 Vuelva a escribir el programa 62 para normalizar el primer segundo de un sonido y luego reducir con lentitud el sonido en intervalos de 1/5 por cada segundo subsiguiente (¿cuántas muestras hay en un segundo? `getSamplingRate` es el número de muestras por segundo para un sonido dado).

7.3 Intente modificar el programa 62 de modo que tenga un aumento lineal en volumen durante la primera mitad del sonido y después haya una reducción lineal del volumen hasta cero en la segunda mitad.

7.4 ¿Qué ocurre si quitamos el pedacito de silencio que se agregó al sonido de destino en el ejemplo de empalme (programa 64, página 180)? ¿Desea averiguarlo? ¿Puede escuchar alguna diferencia?

7.5 Escriba una nueva versión del programa 64 para copiar "We the" en un nuevo sonido, después copiar "united" y por último copiar "people". Asegúrese de agregar 2250 muestras con un valor de 0 entre las palabras. Devuelva el nuevo sonido.

7.6 Pensamos que si vamos a decir "We the united people" en el empalme (programa 64), deberíamos enfatizar la palabra "united" con un volumen bastante alto. Cambie el programa de modo que la palabra "united" se escuche lo más fuerte posible (normalizada) en la frase "united people".

7.7 Trate de usar un cronómetro para medir el tiempo de la ejecución de los programas en este capítulo. Mida desde que se oprime Intro en el comando hasta que aparece el siguiente indicador. ¿Cuál es la relación entre el tiempo de ejecución y la longitud del sonido? ¿Es una relación lineal (es decir, que los sonidos más largos tardan más tiempo en procesarse y los sonidos más cortos tardan menos tiempo)? ¿O es alguna otra cosa? Compare los programas individuales. ¿Acaso normalizar un sonido tarda más tiempo que aumentar (o reducir) la amplitud por una cantidad constante? ¿Qué tanto tiempo más? ¿Acaso importa si el sonido es más largo o más corto?

7.8 Cree un collage de audio. Asegúrese de que tenga por lo menos 5 segundos de duración e incluya como mínimo dos sonidos distintos (es decir, que provengan de archivos diferentes). Haga una copia de uno de esos sonidos diferentes y modifíquelo usando cualquiera de las técnicas descritas en este capítulo (reflejo, empalme, manipulaciones de volumen). Empalme los dos sonidos originales y el sonido modificado para crear el collage completo.

7.9 Componga una oración que nadie haya dicho, combinando palabras de otros sonidos en un nuevo sonido gramaticalmente correcto. Escriba una función llamada `oracionAudio` para generar una oración a partir de palabras individuales. Use por lo menos tres palabras en su oración. Puede usarlas en la carpeta mediasource del sitio Web, o grabar sus propias palabras. Asegúrese de incluir una pausa de una décima (1/10) de segundo entre las palabras. (*Sugerencia 1:* recuerde que los ceros para los valores de muestra generan silencio. *Sugerencia 2:* recuerde que la velocidad de muestreo es el número de muestras por segundo. A partir de esto, debe ser capaz de averiguar cuántas muestras deben ser de cero para generar una décima de segundo de silencio). Asegúrese de acceder a sus sonidos en su carpeta de medios a través de `getMediaPath`, de modo que funcione para los usuarios de su programa siempre y cuando ejecuten `setMediaPath`.

7.10 Escriba una función para agregar 1 segundo de silencio al principio de un sonido. Debe recibir el sonido como entrada, crear el nuevo sonido de destino vacío y luego copiar el sonido original a partir del primer segundo en el destino.

7.11 Busque en Web canciones que tengan mensajes ocultos que pueda escuchar al invertir el sonido.

7.12 Escriba una función que empalme palabras y música.

7.13 Escriba una función que intercale dos sonidos. Debe empezar con 2 segundos del primer sonido y luego con 2 segundos del segundo sonido. Después debe continuar con los siguientes 2 segundos del primer sonido y los siguientes 2 segundos del segundo sonido y así en lo sucesivo, hasta que se hayan copiado ambos sonidos completos en el sonido de destino.

7.14 Escriba una función llamada `borrarParte` para establecer todas las muestras en el segundo segundo de **thisisatest.wav** a 0; en esencia, hay que hacer que el 2o segundo sea un silencio. (*Sugerencia:* recuerde que `getSamplingRate(sonido)` le indica el número de muestras en un solo segundo en un sonido). Reproduzca y devuelva el sonido que se borró en forma parcial.

7.15 ¿Puede escribir una función que encuentre el silencio entre palabras? ¿Qué debería devolver?

7.16 Escriba una función para aumentar el volumen en el sonido que se pase como parámetro, justo entre los índices inicial y final que se pasen como parámetros.

7.17 Escriba una función para invertir parte del sonido que se pase como parámetro, justo entre los índices inicial y final que se pasen como parámetros.

7.18 Escriba una función llamada `reflejarAtrasHaciaDelante` que refleje la segunda mitad de un sonido en la primera mitad.

7.19 Escriba una función llamada `invertirSegundaMitad` que reciba un sonido como entrada, invierta sólo la segunda mitad del sonido y devuelva el resultado. Por ejemplo, si el sonido dice "MarkBark", el sonido devuelto debe decir "MarkkraB".

7.20 Hay una palabra clave de Python llamada `global`. Aprenda lo que hace y luego cambie la función `fun1` de la última sección para que acceda a una variable `c` en el área de comandos *y la modifique* (también puede cambiar la forma en que define la variable en el área de comandos).

CAPÍTULO 8

# Creación de sonidos mediante la combinación de piezas


8.1 COMPOSICIÓN DE SONIDOS POR MEDIO DE LA ADICIÓN

8.2 MEZCLA DE SONIDOS

8.3 CREACIÓN DE UN ECO

8.4 CÓMO FUNCIONAN LOS TECLADOS DE MUESTREO

8.5 SÍNTESIS ADITIVA

8.6 SÍNTESIS DE MÚSICA MODERNA


### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Mezclar sonidos de modo que uno se desvanezca en forma gradual, al tiempo que el otro va apareciendo en forma gradual.
- Crear ecos.
- Cambiar la frecuencia (tono) de un sonido.
- Crear sonidos que no existen en la naturaleza, al componer sonidos más básicos (ondas senoidales).
- Elegir entre los formatos de sonido, como MIDI y MP3, para distintos fines.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar rutas de archivos para hacer referencia a los archivos en distintos lugares del disco.
- Explicar la mezcla como un algoritmo que traspasa los límites de los medios.
- Crear programas a partir de varias funciones.

## 8.1 COMPOSICIÓN DE SONIDOS POR MEDIO DE LA ADICIÓN

Es muy divertido crear sonidos en forma digital que no existían antes. En vez de sólo copiar valores de muestras de un lado a otro o multiplicarlos, en realidad cambiamos sus valores y agregamos ondas. El resultado es un conjunto de sonidos que nunca habían existido antes de que los creáramos.

En la física, para sumar sonidos hay que cancelar ondas y aplicar otros factores. En matemáticas, se trata de matrices. En ciencias computacionales, es uno de los procesos más sencillos que podamos imaginar. Digamos que tiene un sonido llamado `origen` que desea agregar al `destino`. *Sólo tiene que sumar los valores en los mismos números de índice* (como en la figura 8.1). ¡Eso es todo!

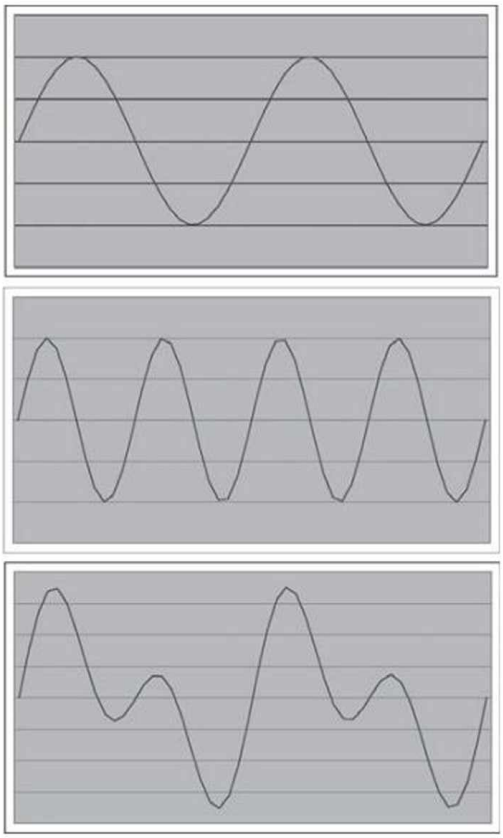


**FIGURA 8.1**
Las ondas superior y media se suman para crear la onda inferior.

```
for indiceOrigen in range(0,getLength(origen)):
  valorDestino=getSampleValueAt(destino,indiceOrigen)
  valorOrigen=getSampleValueAt(origen,indiceOrigen)
  setSampleValueAt(destino,indiceDestino,valorOrigen+valorDestino)
```

La función `setSampleValueAt` recibe un sonido, el índice de la muestra a cambiar en el sonido y el nuevo valor con el que se va a establecer la muestra. Es similar a `setSampleValue`, sólo que no hay que extraer primero la muestra del sonido.

Para hacer algunas de nuestras manipulaciones más sencillas, vamos a usar `setMediaPath` y `getMediaPath`. JES sabe cómo *establecer* un directorio (carpeta) de medios y luego hacer referencia a los archivos de medios dentro de esa carpeta. Esto hace mucho más fácil el proceso de hacer referencia a los archivos de medios: no tenemos que escribir toda la ruta completa. Las funciones que usaremos son `setMediaPath` y `getMediaPath`. La función `setMediaPath()` nos permitirá seleccionar el directorio de medios usando un selector de archivos. La función `getMediaPath(nombreBase)` recibe un nombre de archivo base, agrega el directorio de medios en frente del nombre base y lo devuelve.

```
>>> setMediaPath()
New media folder: C:/ip-book/mediasources/
>>> print getMediaPath("barbara.jpg")
C:/ip-book/mediasources/barbara.jpg
>>> print getMediaPath("sec1silence.wav")
C:/ip-book/mediasources/sec1silence.wav
```

**Error común: no es un archivo, es una cadena**

Sólo porque `getMediaPath` devuelve algo que parece una ruta, no significa que de verdad haya un archivo ahí. Usted tiene que conocer el nombre base correcto y, si lo hace, es más fácil de usar en su código. Pero si usa un nombre que no existe, obtendrá una ruta a un archivo no existente y `getMediaPath` se lo advertirá.

■

```
>>> print getMediaPath("blah-blah-blah")
Note: There is no file at C:/ip-
book/mediasources/blah-blah-blah
C:/ip-book/mediasources/blah-blah-blah
```

## 8.2 MEZCLA DE SONIDOS

En este ejemplo recibimos dos sonidos (alguien que dice "¡Aah!" y el sonido de un instrumento llamado fagot de Do en la cuarta octava) y los *mezclamos*. Para ello copiamos parte del "¡Aah!", después agregamos el 50% de cada uno y luego copiamos el Do. Esto es muy parecido a mezclar el 50% de cada uno en una mezcladora de sonidos. También es muy parecido a la forma en que mezclamos imágenes en el programa 45 (página 123).

**Programa 70: mezclar dos sonidos**

```
def mezclarSonidos():
  bajo = makeSound(getMediaPath("bassoon-c4.wav"))
  aah = makeSound(getMediaPath("aah.wav"))
  lienzo = makeSound(getMediaPath("sec3silence.wav"))
  #Cada uno de estos tiene más de 40k muestras
  for indice in range(0,20000):
    setSampleValueAt(lienzo,indice,getSampleValueAt(aah,indice))
  for indice in range(0,20000):
    aahMuestra = getSampleValueAt(aah,indice+20000)
    bajoMuestra = getSampleValueAt(bajo,indice)
    nuevaMuestra = 0.5*aahMuestra + 0.5 * bajoMuestra
    setSampleValueAt(lienzo,indice+20000,nuevaMuestra)
  for indice in range(20000,40000):
    setSampleValueAt(lienzo,indice+20000,getSampleValueAt(bajo,indice))
  play(lienzo)
  return lienzo
```

■

### Cómo funciona

Al igual que en la mezcla de las imágenes (programa 45, página 123), hay ciclos en esta función para cada segmento del sonido mezclado.

- Primero creamos los sonidos `bajo` y `aah` para mezclar, además de un sonido de silencio llamado `lienzo`, en donde vamos a realizar la mezcla. La longitud de estos sonidos es de más de 40000 muestras, pero sólo vamos a usar las primeras 40000 como ejemplo.
- En el primer ciclo, sólo obtenemos 20000 muestras de `aah` y las copiamos en `lienzo`. Observe que no usamos una variable índice separada para el `lienzo`; en vez de ello usamos la misma variable índice llamada `indice` para ambos sonidos.
- En el siguiente ciclo copiamos 20000 muestras de `aah` y `bajo` mezcladas en `lienzo`. Usamos la variable índice llamada `indice` para indexar los tres sonidos: usamos el `indice` básico para acceder a `bajo` y sumamos 20000 a `indice` para acceder a `aah` y `lienzo` (puesto que ya copiamos 20000 muestras de `aah` en `lienzo`). Obtenemos una muestra de `aah` y una de `bajo`, luego multiplicamos cada una por 0.5 y sumamos los resultados. El resultado es una muestra que representa el 50% de cada entrada.
- Por último copiamos otras 20000 muestras de `bajo`. El sonido resultante se devuelve (ya que de otra forma desaparecería), el cual suena como "Aah", luego un poco de cada sonido y después sólo una nota de fagot.

## 8.3 CREACIÓN DE UN ECO

La creación de un efecto de eco es similar a la receta del empalme (programa 64, página 180) que vimos en el capítulo anterior, sólo que involucra la creación de sonidos que no existían antes. Para ello *sumamos* formas de onda. Lo que hacemos aquí es sumar muestras de un número de muestras de `retraso` en el sonido pero las multiplicamos por 0.6, de modo que sean más tenues.

**Programa 71: crear un sonido y un solo eco del mismo**

```
def eco(retraso):
  f = pickAFile()
  s1 = makeSound(f)
  s2 = makeSound(f)
  for indice in range(retraso, getLength(s1)):
    # establece el valor de retraso al valor original + valor retrasado * .6
    ecoMuestra = 0.6*getSampleValueAt(s2, indice-retraso)
    comboMuestra = getSampleValueAt(s1,indice) + ecoMuestra
    setSampleValueAt(s1,indice,comboMuestra)
  play(s1)
  return s1
```

■

### Cómo funciona

La función `eco` recibe como entrada la cantidad de retraso entre ecos y devuelve el sonido con eco. Pruebe esto con distintas cantidades de retraso. Con valores bajos de retraso, el eco sonará más como *vibrato*. Los valores más altos (pruebe 10000 o 20000) le proporcionarán un eco real.

- Esta función le pide un nombre de archivo al que se va a agregar un eco (no es una gran idea si queremos usar la función generadora de ecos para otros fines), luego crea dos copias del sonido. `s1` es donde crearemos el sonido con eco. `s2` es en donde obtendremos las muestras originales sin adulterar para crear el eco (podría probar esto con sólo un sonido para obtener ecos interesantes en capas).
- Nuestro ciclo `indice` omite las muestras de `retraso` y luego itera hasta el final del sonido.
- El sonido con eco se atrasa una cantidad de muestras definidas por `retraso`, por lo que `indice-retraso` es la muestra que necesitamos. Lo multiplicamos por 0.6 para suavizar el volumen.
- A continuación sumamos la muestra con eco a la muestra actual en `comboMuestra`, y luego la almacenamos en el `indice`.
- Al final, reproducimos el sonido con `play` y lo regresamos con `return`.

La figura 8.2 proporciona un ejemplo de cómo funciona este programa. Piense en la primera fila ("100,200,1000,-150,-350,200,500,10...") como el sonido original. Imagine un retraso de 3 (que es demasiado pequeño como para escucharlo, pero funciona para un ejemplo). Creamos una copia del sonido y lo multiplicamos por 0.6 para obtener la nueva fila "60, 120, 600, -90, -210, 120, 300, 6...". Sumamos estas dos filas entre sí, pero con una desviación con base en el retraso. Así que mantenemos 100, 200, 1000, luego sumamos –150+60 para obtener 90, luego –350+120 para obtener –230, y así en lo sucesivo. El resultado es que sumamos el sonido a sí mismo, pero retrasado y menos fuerte (multiplicado por 0.6).

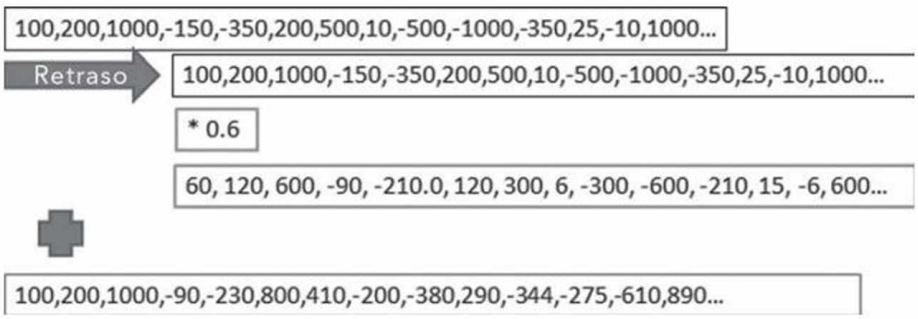



**FIGURA 8.2**
Un ejemplo de cómo funciona el eco.

### 8.3.1 Creación de múltiples ecos

Este programa le permite establecer el número de ecos que obtendrá. Puede generar algunos efectos sorprendentes con él.

**Programa 72: creación de múltiples ecos**

```
def ecos(archSonido, retraso, num):
  # Crea un nuevo sonido, que agrega eco al archivo de sonido de entrada
  # num es el número de ecos, cada uno separado por retraso
  s1 = makeSound(archSonido)
  fins1 = getLength(s1)
```

```
fins2 = fins1 + (retraso * num)
s2 = makeEmptySound(fins2)

ecoAmplitud = 1.0
for ecoCuenta in range(1, num):
  # 60% más pequeño cada vez
  ecoAmplitud = ecoAmplitud * 0.6
  for posns1 in range(0,fins1):
    posns2 = posns1 + (retraso * ecoCuenta)
    valores1 = getSampleValueAt(s1,posns1)*ecoAmplitud
    valores2 = getSampleValueAt(s2,posns2)
    setSampleValueAt(s2,posns2,valores1+valores2)
play(s2)
return s2
```

■

### 8.3.2 Creación de acordes

Un acorde musical está formado por tres o más notas que crean un sonido armonioso cuando se reproducen en conjunto. Un acorde de Do mayor es una combinación de las notas Do (C), Mi (E) y Sol (G). Para crear el acorde, podemos sumar los valores en el mismo índice. El directorio **mediasources** contiene archivos de sonido para las notas Do (C), Mi (E) y Sol (G), cada una de las cuales se reproduce en un fagot.

**Programa 73: creación de un acorde**

```
def crearAcorde():
  c = makeSound(getMediaPath("bassoon-c4.wav"))
  e = makeSound(getMediaPath("bassoon-e4.wav"))
  g = makeSound(getMediaPath("bassoon-g4.wav"))
  acorde = makeEmptySound(c.getLength())
  for indice in range(0,c.getLength()):
    cValor = getSampleValueAt(c,indice)
    eValor = getSampleValueAt(e,indice)
    gValor = getSampleValueAt(g,indice)
    total = cValor + eValor + gValor
    setSampleValueAt(acorde,indice,total)
  return acorde
```

■

## 8.4 CÓMO FUNCIONAN LOS TECLADOS DE MUESTREO

Los teclados de muestreo son teclados que usan grabaciones de sonidos (como pianos, harpas, trompetas) para crear música, reproduciendo esas grabaciones en el tono deseado. Los teclados modernos de música y sonido (y los sintetizadores) permiten a los músicos grabar sonidos en sus vidas diarias y convertirlos en "instrumentos" al desplazar sus frecuencias originales. ¿Cómo hacen esto los sintetizadores? En realidad no es complicado. La parte interesante es que le permiten usar cualquier sonido que desee como instrumento.

Los teclados de muestreo usan enormes cantidades de memoria para grabar muchos instrumentos diferentes en distintos tonos. Cuando presiona una tecla se selecciona la grabación

*más cercana* (en tono) a la nota que presionó y después se cambia al tono exacto que usted solicitó.

Este primer programa funciona creando un sonido que *omite* una de cada dos muestras. Leyó bien: después de tener tanto cuidado en tratar a todas las muestras de la misma forma, ¡ahora vamos a omitir la mitad de ellas! En el directorio **mediasources** encontrará un sonido llamado **c4.wav**. Ésta es la nota en C, en la cuarta octava de un piano, que se reproduce durante un segundo. Es un buen sonido con el que podemos experimentar, aunque cualquier sonido funcionará.

**Programa 74: duplicar la frecuencia de un sonido**

```
def duplicar(origen):
  longitud = getLength(origen) / 2 + 1
  destino = makeEmptySound(longitud)
  indiceDestino = 0
  for indiceOrigen in range(0, getLength(origen), 2):
    valorOrigen = getSampleValueAt(origen,indiceOrigen)
    setSampleValueAt(destino, indiceDestino, valorOrigen)
    indiceDestino = indiceDestino + 1
  play(destino)
  return destino
```

■

A continuación le mostramos cómo usar el programa anterior:

```
>>> archivo = pickAFile()
>>> print archivo
C:/ip-book/mediasources/c4.wav
>>> c4 = makeSound(archivo)
>>> play(c4)
>>> c4duplicado=duplicar(c4)
```

Este programa parece como si usara el subprograma de copia de arreglos que vimos antes, pero observe que `range` usa el tercer parámetro: estamos incrementando por 2. Si incrementamos por 2, sólo llenamos la mitad de las muestras en el destino, por lo que el segundo ciclo sólo llena el resto con ceros.

¡Pruébelo![1] Podrá escuchar con facilidad que en verdad el sonido duplica su frecuencia. Si explora el primer sonido y el nuevo sonido duplicado (usando **c4.wav**) mediante la función `explore`, se verá como en la figura 8.3. Ambos sonidos *se ven* iguales, pero observe "el número de píxeles entre las muestras". En una es de 43 y en la otra es de 86. En verdad el sonido se duplicó: la forma de onda se ve igual, y así es como debería ser exactamente.

¿Cómo ocurrió esto? En realidad no es tan complicado. Piense de esta forma: la frecuencia del archivo básico es en realidad el número de ciclos que pasan en cierta cantidad de tiempo. Si omite una de cada dos muestras, el nuevo sonido tiene los mismos ciclos sólo que ocurren en la mitad del tiempo.

Ahora probemos de la otra forma. Vamos a tomar cada muestra dos veces. ¿Qué ocurre entonces?

Para ello, necesitamos usar la función de Python llamada `int` para devolver la porción entera de la entrada.

[1]Está probando esto a medida que va leyendo, ¿verdad?

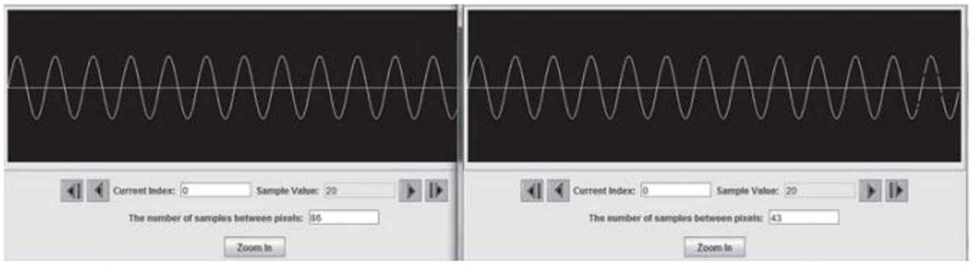



**FIGURA 8.3**
Una onda de sonido duplicada.

```
>>> print int(0.5)
0
>>> print int(1.5)
1
```

He aquí el programa que convierte la frecuencia a *la mitad*. Usamos la subreceta para copiar arreglos otra vez, sólo que haremos algo así como invertirla. El ciclo `for` mueve el `indiceDestino` a lo largo del sonido. El `indiceOrigen` ahora va a incrementarse, pero sólo por 0.5. El efecto es que tomamos cada muestra en el origen dos veces. El `indiceOrigen` será 1, 1.5, 2, 2.5 y así en lo sucesivo, pero como vamos a usar la parte `int` de ese valor, tomaremos las muestras 1, 1, 2, 2 y así en lo sucesivo.

**Programa 75: convertir la frecuencia a la mitad**

```
def mitad(origen):
  destino = makeEmptySound(getLength(origen) * 2)
  indiceOrigen = 0
  for indiceDestino in range(0, getLength(destino)):
    valor = getSampleValueAt(origen, int(indiceOrigen))
    setSampleValueAt(destino, indiceDestino, valor)
    indiceOrigen = indiceOrigen + 0.5
  play(destino)
  return destino
```

■

## Cómo funciona

La función `mitad` toma el sonido de origen como entrada. Crea un sonido de destino que es del doble de largo que el sonido de origen. El `indiceOrigen` se establece en 0 (ahí es de donde vamos a copiar desde el `origen`) y tenemos un ciclo para `indiceDestino` desde 0 hasta el final del sonido `destino`. Obtenemos un valor de muestra de `origen` en el valor *entero* (`int`) del `indiceOrigen`. Establecemos el valor de muestra en `indiceDestino` con el valor que obtuvimos de la muestra de `origen`. Luego sumamos 0.5 al `indiceOrigen`. Esto significa que cada vez que pase por el ciclo, el `indiceOrigen` tomará los valores 0, 0.5, 1, 1.5, 2, 2.5 y así sucesivamente. Pero la parte entera de esta secuencia es 0, 0, 1, 1, 2, 2 y así en lo sucesivo. El resultado es que tomamos cada muestra del sonido `origen` *dos veces*.

Piense en lo que estamos haciendo aquí. Imagine que el número 0.5 anterior es en realidad 0.75, o 2, o 3. ¿Funcionaría esto? Tendríamos que cambiar el ciclo `for`, pero en esencia la

idea es la misma en todos estos casos. Estamos *muestreando* los datos de origen para crear los datos de destino. Si usamos un *índice de muestra* de 0.5 reducimos la velocidad del sonido y cambiamos la frecuencia a la mitad. Un índice de muestra mayor a 1 agiliza el sonido y aumenta la frecuencia.

Ahora tratemos de generalizar este muestreo con el siguiente programa (tenga en cuenta que éste *no* funcionará bien).

**Programa 76: desplazar la frecuencia de un sonido: ¡DESCOMPUESTO!**

```
def desplazar(origen,factor):
  destino = makeEmptySound(getLength(origen))
  indiceOrigen = 0

  for indiceDestino in range(0, getLength(destino)):
    valorOrigen = getSampleValueAt(origen,int(indiceOrigen))
    setSampleValueAt(destino, indiceDestino, valorOrigen)
    indiceOrigen = indiceOrigen + factor

  play(destino)
  return destino
```

■

Así es como podríamos usar el programa anterior:

```
>>> cF=getMediaPath("c4.wav")
>>> print cF
C:/ip-book/mediasources/c4.wav
>>> c4 = makeSound(cF)
>>> c4Inferior=desplazar(c4,0.75)
```

Esto parece funcionar, pero ¿qué pasa si el factor de muestreo es *mayor* a 1.0?

```
>>> pruebaMasAlta=desplazar(c4,1.5)
You are trying to access the sample at index: 67584,
but the last valid index is at 67584
The error was:
Inappropriate argument value (of correct type).
An error occurred attempting to pass an argument to a function.
Please check line 218 of C:\ip-book\
programas\miSonido.py
```

¿Por qué? ¿Qué está ocurriendo? He aquí cómo puede verlo: imprima el `indiceOrigen` justo antes de `setSampleValueAt`. Así podrá ver que el `indiceOrigen` se hace *más grande* que la longitud del sonido de origen. Desde luego que tiene sentido. Si cada vez que recorremos el ciclo incrementamos el `indiceDestino` en 1, pero incrementamos el `indiceOrigen` en *más de* 1, pasaremos el final del sonido de origen antes de llegar al final del sonido de destino. Pero ¿cómo evitamos esto?

He aquí lo que queremos que ocurra: si el `indiceOrigen` se hace más grande que la longitud del origen, deseamos restablecer el `indiceOrigen` de vuelta a cero. En Python usamos una instrucción `if` que sólo ejecute el código en el siguiente bloque si la prueba es verdadera.

**Programa 77: desplazar la frecuencia de un sonido**

```
def desplazar(origen,factor):
  destino = makeEmptySound(getLength(origen))
  indiceOrigen = 0

  for indiceDestino in range(0, getLength(destino)):
    valorOrigen = getSampleValueAt(origen,int(indiceOrigen))
    setSampleValueAt(destino, indiceDestino, valorOrigen)
    indiceOrigen = indiceOrigen + factor
    if (indiceOrigen >= getLength(origen)):
      indiceOrigen = 0

  play(destino)
  return destino
```

■

En realidad podemos establecer el factor para obtener la frecuencia que deseemos. A este factor le llamamos el *intervalo de muestreo*. Para una frecuencia deseada $f_0$, el intervalo de muestreo debería ser:

$$intervaloMuestreo = (tamañoDelSonidoDeOrigen)\frac{f_0}{velocidadMuestreo}$$

Así es como funciona un sintetizador de teclado. Tiene grabaciones de pianos, voces, campanas, tambores, lo que sea. Al *muestrear* esos sonidos a distintos intervalos de muestreo, puede desplazar el sonido a la frecuencia deseada.

El último programa de esta sección reproduce un sonido individual a su frecuencia original, luego a dos veces la frecuencia original, tres veces, cuatro veces y cinco veces.

**Programa 78: reproducir un sonido en un rango de frecuencias**

```
def reproducirUnaSecuencia(archivo):
  # Reproduce el sonido cinco veces, aumentando la frecuencia
  for factor in range(1,6):
    sonido = makeSound(archivo)
    destino = desplazar(sonido,factor)
    blockingPlay(destino)
```

■

### 8.4.1 El muestreo como un algoritmo

Debería ser capaz de reconocer una similitud entre la receta para reducir la frecuencia a la mitad, programa 75 (página 200) y la receta para aumentar la escala de una imagen (hacerla más grande), programa 35 (página 102). Para reducir la frecuencia a la mitad, tomamos cada muestra dos veces al incrementar por 0.5 y usamos la función `int()` para obtener la parte entera de eso. Para hacer más grande la imagen, tomamos cada pixel dos veces, sumamos 0.5 a nuestras variables índice y usamos la función `int()`en ellas. Estas recetas utilizan el

mismo *algoritmo*: se utiliza el mismo proceso básico en cada una. Los detalles de imágenes comparadas con sonidos no son importantes. El punto es que se utiliza el mismo proceso básico en cada receta.

Hemos visto otros algoritmos que cruzan los límites de los medios. Sin duda, nuestras funciones para aumentar el rojo y aumentar el volumen (además de las versiones para reducir estos elementos) hacen en esencia lo mismo. La forma en que mezclamos imágenes o sonidos es igual. Tomamos los canales de componentes de colores (píxeles) o las muestras (sonidos) y las sumamos usando porcentajes para determinar la cantidad de cada una que deseamos en el producto final. Mientras que los porcentajes den un total de 100%, obtendremos un resultado razonable que refleje los sonidos o las imágenes de entrada en los porcentajes correctos.

Es útil identificar algoritmos como éstos por varias razones. Si comprendemos el algoritmo en general (por ejemplo, cuándo es lento y cuándo rápido, con qué funciona y con qué no, cuáles son sus limitaciones), entonces las lecciones aprendidas se aplican en las instancias específicas de la imagen o el sonido. También es útil que los diseñadores conozcan los algoritmos. Al diseñar un nuevo programa, tenga los algoritmos en cuenta de modo que pueda usarlos cuando se apliquen.

Al duplicar o reducir a la mitad la frecuencia de sonido, también estamos encogiendo o duplicando su longitud. Tal vez desee un sonido de destino cuya longitud sea *exactamente* la misma del sonido de origen, en vez de tener que borrar cosas adicionales de un sonido más largo. Puede hacer eso con `makeEmptySound`. La función `makeEmptySound(22050*10)` devuelve un nuevo sonido vacío de 10 segundos de longitud, con una velocidad de muestreo de 22050.

## 8.5 SÍNTESIS ADITIVA

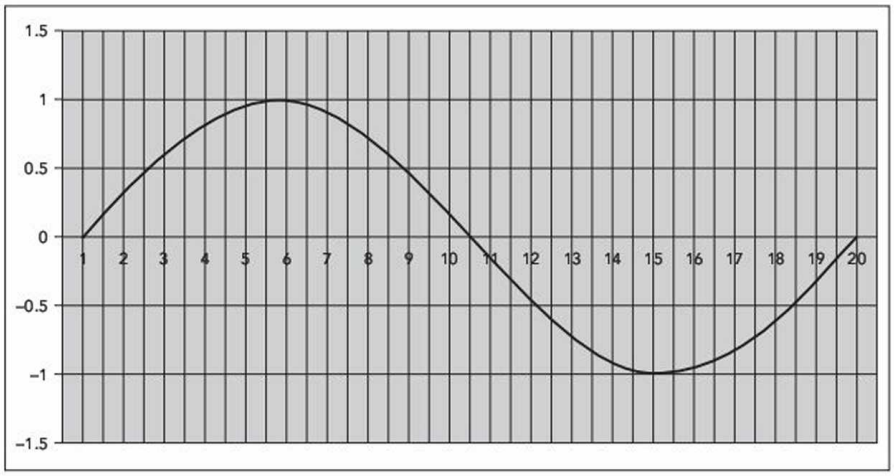
La síntesis aditiva crea sonidos al sumar ondas senoidales. En párrafos anteriores vimos que es bastante sencillo sumar sonidos. Con la síntesis aditiva podemos dar forma a las ondas nosotros mismos, establecer sus frecuencias y crear "instrumentos" que nunca antes habían existido.

### 8.5.1 Creación de ondas senoidales

Vamos a averiguar cómo producir un conjunto de muestras para generar un sonido a una frecuencia y amplitud dadas. Una forma sencilla de hacer esto es crear una onda senoidal. Por ejemplo, un silbato es una onda senoidal casi perfecta. El truco es crear una onda senoidal con la frecuencia deseada.

Si tomáramos valores desde 0 hasta $2\pi$, calculáramos el seno de cada valor y graficáramos los valores calculados, obtendríamos una onda senoidal. En los cursos de matemáticas más básicos vemos que hay una infinidad de números entre 0 y 1. Las computadoras no manejan muy bien el infinito, por lo que en realidad sólo tomamos *algunos* valores entre 0 y $2\pi$.

Para crear el gráfico que se muestra a continuación, Mark llenó 20 filas (un número totalmente arbitrario) de una hoja de cálculo con valores desde 0 hasta $2\pi$ (alrededor de 6.28). Mark sumó el valor aproximado de 0.314 (6.28/20) a cada fila precedente. En la siguiente columna tomó el seno de cada valor de la primera columna y luego lo graficó.




Si deseamos crear un sonido a una frecuencia dada, por decir de 440 Hz, tenemos que ajustar todo un ciclo como el del gráfico en 1/440 de un segundo (440 ciclos por segundo, lo que significa que cada ciclo cabe en 1/440 de segundo, o 0.00227 segundos). Para crear el gráfico, Mark usó 20 valores. Vamos a llamarles 20 *muestras*. ¿En cuántas muestras tenemos que partir el ciclo de 440 Hz? Es lo mismo que preguntar cuántas muestras deben pasar en 0.00227 segundos. Conocemos la velocidad de muestreo: es el número de muestras en un segundo. Digamos que es de 22050 muestras por segundo (nuestra velocidad de muestreo predeterminada). Cada muestra es entonces de 1/22050, lo cual equivale a 0.0000453 segundos. ¿Cuántas muestras caben en 0.00227? Esto es 0.00227/0.0000453, o cerca de 50. Lo que hicimos en sentido matemático es:

$$intervalo = 1/frecuencia$$

$$muestrasPorCiclo = \frac{intervalo}{1/velocidadMuestreo} = (velocidadMuestreo)(intervalo)$$

Ahora vamos a escribir esto como Python. Para obtener una forma de onda a una frecuencia dada, digamos de 440 Hz, necesitamos 440 de estas ondas en un solo segundo. Cada una debe caber en el intervalo de 1/*frecuencia*. El número de muestras que necesitamos producir durante el intervalo es la velocidad de muestreo dividida entre la frecuencia, o intervalo (1/*f*) * (*velocidad de muestreo*). Llamemos a esto las *muestrasPorCiclo*.

En cada entrada del sonido *indiceMuestra*, queremos:

- Obtener la fracción de *indiceMuestra/muestrasPorCiclo*.
- Multiplicar esa fracción por 2π. Ése es el número de radianes que necesitamos. Luego obtener el *seno* de (*indiceMuestra/muestrasPorCiclo*) * 2π.
- Multiplicar el resultado por la amplitud deseada y colocarlo en el `indiceMuestra`.

Para crear sonidos, existen algunos sonidos de *silencio* en las fuentes de medios. Nuestro generador de ondas senoidales usará un segundo de silencio para crear una onda senoidal de un segundo. Proveeremos una amplitud como entrada: será la amplitud *máxima* del sonido (como el seno genera entre −1 y 1, el rango de amplitudes estará entre −*amplitud* y *amplitud*).

**Programa 79: generar una onda senoidal a una frecuencia y amplitud dadas**

```
def ondaSenoidal(frec,amplitud):

  # Obtener un sonido en blanco
  miSonido = getMediaPath('sec1silence.wav')
  crearSen = makeSound(miSonido)

  # Establecer un sonido constante
  vm = getSamplingRate(crearSen)              # velocidad de muestreo

  intervalo = 1.0/frec      # asegurarse de que sea punto flotante
  muestrasPorCiclo =  intervalo * vm # muestras por ciclo
  cicloMax = 2 * pi

  for pos in range(0,getLength(crearSen)):
    muestraOriginal = sin((pos / muestrasPorCiclo) * cicloMax)
    valMuestra = int(amplitud*muestraOriginal)
    setSampleValueAt(crearSen,pos,valMuestra)

  return crearSen
```

■

Observe que usamos 1.0 dividido entre `frec` para calcular el intervalo. Usamos 1.0 en vez de 1 para asegurarnos de que el resultado sea punto flotante y no entero. Si Python nos ve usando sólo enteros, nos dará un resultado entero y descartará todo lo que esté después del punto decimal. Si usamos al menos un número de punto flotante (1.0), nos proporcionará el resultado como punto flotante.

A continuación crearemos una onda senoidal de 880 Hz con una amplitud de 4000:

```
>>> f880=ondaSenoidal(880,4000)
>>> play(f880)
```

### 8.5.2 Sumar ondas senoidales

Ahora vamos a sumar ondas senoidales. Como dijimos al principio del capítulo, es bastante sencillo: sólo hay que sumar las muestras en los mismos índices. He aquí una función que suma un sonido a un segundo sonido.

**Programa 80: sumar dos sonidos**

```
def sumarSonidos(sonido1,sonido2):
  for indice in range(0,getLength(sonido1)):
    s1Muestra = getSampleValueAt(sonido1,indice)
    s2Muestra = getSampleValueAt(sonido2,indice)
    setSampleValueAt(sonido2,indice,s1Muestra+s2Muestra)
```

■

Vamos a sumar 440 Hz, 880 Hz (dos veces 440) y 1 320 Hz (880 + 440) pero haremos que aumenten las amplitudes. Duplicaremos la amplitud cada vez: 2000, después 4000, luego 8000. Sumaremos todo y colocaremos el resultado en el nombre `f440`; después exploraremos el resultado. Al final generaremos un sonido de 440 Hz, para poder escuchar ambos y compararlos.

```
>>> f440=ondaSenoidal(440,2000)
>>> f880=ondaSenoidal(880,4000)
>>> f1320=ondaSenoidal(1320,8000)
>>> sumarSonidos(f880,f440)
>>> sumarSonidos(f1320,f440)
>>> play(f440)
>>> explore(f440)
>>> solo440=ondaSenoidal(440,2000)
>>> play(solo440)
>>> explore(f440)
```

**Error común: tenga cuidado al sumar amplitudes superiores a 32767**

Al sumar sonidos, también sumamos sus amplitudes. Un máximo de 2000 + 4000 + 8000 nunca será mayor a 32767, pero no se preocupe por eso. Recuerde lo que ocurrió cuando la amplitud aumentó demasiado en el capítulo anterior...

■

### 8.5.3 Comprobación de nuestro resultado

¿Cómo sabemos si en verdad obtuvimos lo que queríamos? Si exploramos el sonido `f440` original y el sonido `f440` modificado, que en realidad es la combinación de los tres sonidos, observará que las formas de onda se ven muy diferentes (figura 8.4). Eso nos indica que hicimos *algo* con el sonido... ¿pero qué fue?

Para evaluar nuestro código podemos usar las herramientas de sonido en MediaTools. Primero guardamos una onda de muestra (sólo 400 Hz) y la onda combinada.

```
>>> writeSoundTo(solo440,"C:/ip-book/mediasources/solo440.wav")
>>> writeSoundTo(f440,"C:/ip-book/mediasources/440combinada.wav")
```

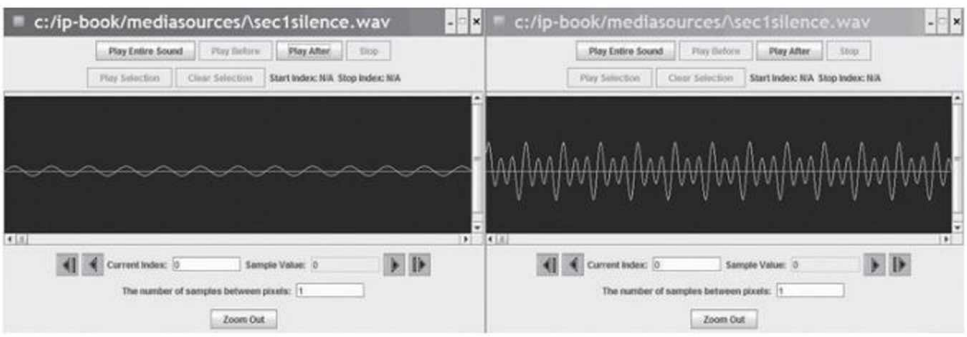



**FIGURA 8.4**
La señal de 440 Hz pura (*izquierda*) y la señal de 440 + 800 + 1 320 Hz (*derecha*).

La forma en que podemos comprobar de verdad la síntesis aditiva es con una FFT (transformada rápida de Fourier). Puede generar la FFT para cada señal mediante la aplicación de MediaTools. Descubrirá que la señal de 440 Hz tiene un solo pico (figura 8.5). Eso es lo que podríamos esperar: se supone que debe ser una sola onda senoidal. Ahora observe la FFT de la forma de onda combinada. Es lo que se supone que debe ser. Podemos ver tres picos ahí; además cada pico subsiguiente tiene el doble de altura del anterior.

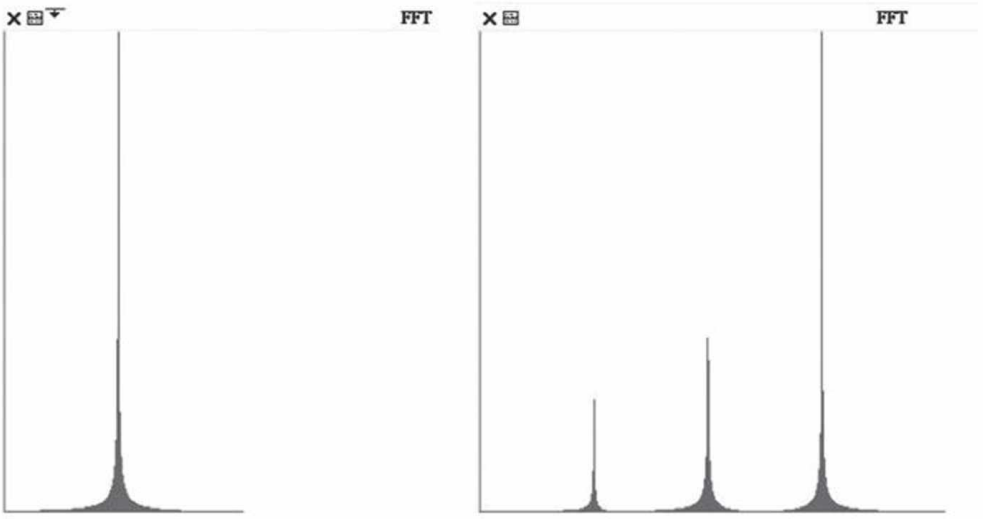



**FIGURA 8.5**
FFT del sonido de 440 Hz (*izquierda*) y el sonido combinado (*derecha*).

### 8.5.4 Ondas cuadradas

No tenemos que sumar sólo ondas senoidales. También podemos sumar *ondas cuadradas*. En sentido literal, éstas son ondas con forma cuadrada, que se mueven entre +1 y −1. La FFT se verá bastante distinta, y el *sonido* será muy diferente. En realidad puede ser un sonido mucho más rico.

Trate de cambiar este programa por el generador de ondas senoidales y juzgue por usted mismo. Observe el uso de una instrucción `if` para alternar entre los lados positivo y negativo de la onda a la mitad de un ciclo.

Podemos usar la función anterior de la siguiente forma:

**Programa 81: generador de ondas cuadradas para una frecuencia y amplitud dadas**

```
def ondaCuadrada(frec,amplitud):

  # Obtener un sonido en blanco
  miSonido = getMediaPath("sec1silence.wav")
  cuadrada = makeSound(miSonido)

  # Establecer constantes de música
  velocidadMuestreo = getSamplingRate(cuadrada)     # velocidad de muestreo
  segundos = 1   # reproducir por 1 segundo
```

```
# Crear herramientas para esta onda
# segundos por ciclo: asegurar que sea punto flotante
intervalo = 1.0 * segundos / frec
# crea punto flotante ya que el intervalo es punto flotante
muestrasPorCiclo = intervalo * velocidadMuestreo
# necesitamos cambiar cada mitad de ciclo
muestrasPorMedioCiclo = int(muestrasPorCiclo / 2)
valMuestra = amplitud
s = 1
i = 1


for s in range(0,getLength(cuadrada)):
  # si es el final de medio ciclo
  if (i > muestrasPorMedioCiclo):
    # invertir la amplitud cada medio ciclo
    valMuestra = valMuestra * -1
    # y reinicializar el contador de medio ciclo
    i = 0
  setSampleValueAt(cuadrada,s,valMuestra)
  i = i + 1


return(cuadrada)
```

■

```
>>> cuad440=ondaCuadrada(440,4000)
>>> play(cuad440)
>>> cuad880=ondaCuadrada(880,8000)
>>> cuad1320=ondaCuadrada(1320,10000)
>>> writeSoundTo(cuad440,getMediaPath("cuadrada440.wav"))
Note: There is no file at C:/ip-book/mediasources/
cuadrada440.wav
>>> sumarSonidos(cuad880,cuad440)
>>> sumarSonidos(cuad1320,cuad440)
>>> play(cuad440)
>>> writeSoundTo(cuad440,getMediaPath("cuadradacombinada440.wav"))
Note: there is no file at C:/ip-book/mediasources/
cuadradacombinada440.wav
```

### Cómo funciona

Este programa crea ondas senoidales con forma cuadrada y todos los valores de muestra son la amplitud que se pasa, –1 por la amplitud. Veamos ahora paso a paso lo que ocurre al ejecutar `cuad440 = ondaCuadrada(440,4000)`.

- Primero, creamos un sonido de silencio con duración de 1 segundo, llamado `cuadrada`.
- Después calculamos el número de muestras por ciclo con base en la velocidad de muestreo, la frecuencia y la longitud en segundos. Esto es (1.0 * 1/440) * 22050, que nos da un resultado aproximado de 50.11363.

- Calculamos el número de muestras en la mitad de un ciclo como un entero (25), de modo que la mitad de los valores en un ciclo sean positivos y la mitad negativos. Usamos `i` para rastrear cuántos valores se establecieron en `cuadrada`, para comprobar si llevamos medio ciclo. Establecemos el `valMuestra` a la `amplitud` que se pasó como parámetro.
- Si llegamos al final de un medio ciclo (`i == muestrasPorMedioCiclo`), multiplicamos el `valMuestra` por −1 para negarlo. Si `valMuestra` es positivo, se hará negativo. Y si es negativo se hará positivo. También restablecemos `i` a cero para empezar a contar el siguiente medio ciclo.
- Establecemos el valor de muestra en `cuadrada` a `valMuestra`. Incrementamos `i`.
- Una vez que termina el ciclo, devolvemos el sonido `cuadrada`.

Encontrará que las ondas en realidad sí se ven cuadradas (figura 8.6), pero lo más sorprendente son todos los picos adicionales en FFT (figura 8.7). En verdad las ondas cuadradas producen un sonido mucho más complejo.

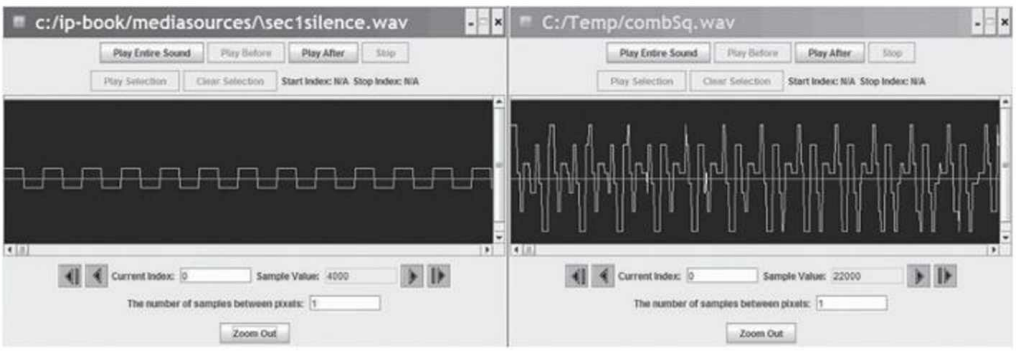



**FIGURA 8.6**
La onda cuadrada de 440 Hz (*izquierda*) y una combinación aditiva de ondas cuadradas (*derecha*).

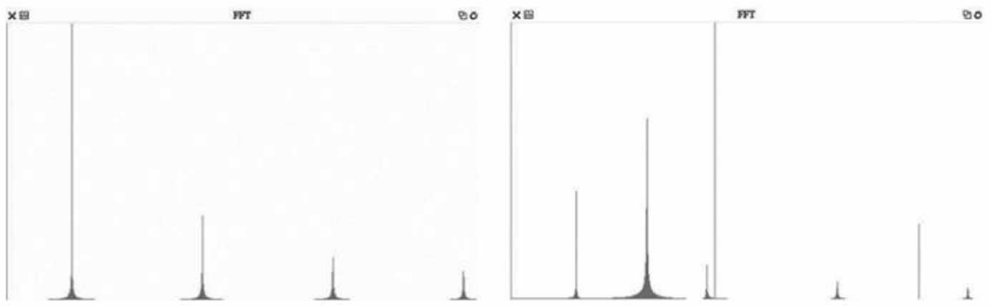



**FIGURA 8.7**
Las FFT de la onda cuadrada de 440 Hz (*izquierda*) y la combinación aditiva de ondas cuadradas (*derecha*).

### 8.5.5 Ondas triangulares

Podemos crear ondas triangulares en vez de ondas cuadradas con el siguiente programa.

**Programa 82: generación de ondas triangulares**

```
def ondaTriangular(frec, laAmplitud):

  # Obtener un sonido en blanco
  miSonidoF = getMediaPath("sec1silence.wav")
  triangulo = makeSound(miSonidoF)

  # Establecer constantes de música
  # usar la amplitud que se pasó
  amplitud = laAmplitud
  # velocidad de muestreo (muestras por segundo)
  velocidadMuestreo = 22050
  # reproducir durante 1 segundo
  segundos = 1

  # Crear herramientas para esta onda
  # segundos por ciclo: asegurarse de que sea punto flotante
  intervalo = 1.0 * segundos / frec
  # crea punto flotante ya que el intervalo es punto flotante
  muestrasPorCiclo = intervalo * velocidadMuestreo
  # necesitamos cambiar cada medio ciclo
  muestrasPorMedioCiclo = int(muestrasPorCiclo / 2)
  # valor a sumar por cada muestra subsiguiente; debe ser entero
  incremento = int(amplitud / muestrasPorMedioCiclo)
  # empieza desde abajo e incrementa o decrementa según sea necesario
  valMuestra = -amplitud
  i = 0

  # crear sonido de 1 segundo
  for s in range(0,velocidadMuestreo):

    # si es el final de medio ciclo
    if (i == muestrasPorMedioCiclo):
      # invertir el incremento cada medio ciclo
      incremento = incremento * -1
      # y reiniciar el contador de medio ciclo
      i = 0

    valMuestra = valMuestra + incremento
    setSampleValueAt(triangulo,s,valMuestra)
    i = i + 1

  play(triangulo)
  return triangulo
```

■

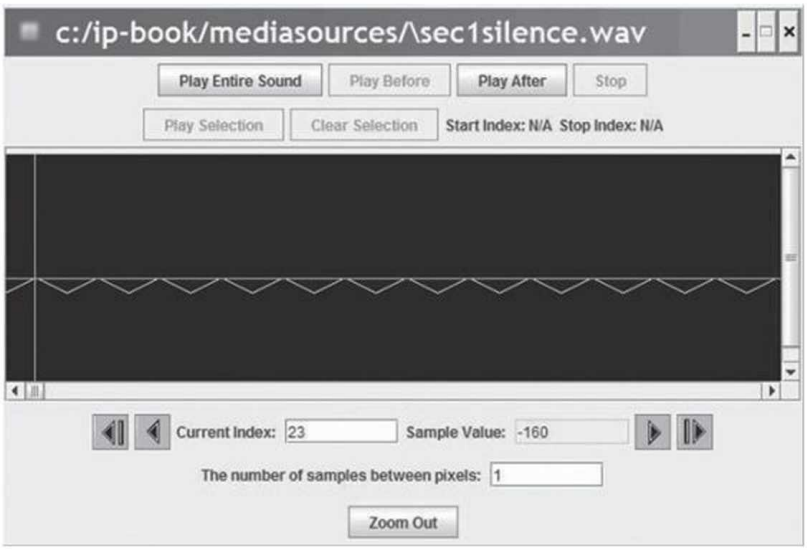



**FIGURA 8.8**
Exploración de una onda triangular.

Puede usar el programa anterior de la siguiente forma:

```
>>> tri440=ondaTriangular(440,4000)
>>> explore(tri440)
```

### Cómo funciona

Esta receta es similar a la que crea ondas cuadradas, sólo que estas ondas tienen forma triangular. El `valMuestra` se inicializará con la negación de la amplitud que se pasa como parámetro. Se suma un incremento al `valMuestra` cada vez que se itera por el ciclo. La variable `i` rastrea en dónde nos encontramos en el ciclo y cuando llegamos a la mitad de uno, niega el `incremento` y restablece el valor de `i` a cero.

## 8.6 SÍNTESIS DE MÚSICA MODERNA

Los primeros sintetizadores de música funcionaban por medio de la síntesis aditiva. Hoy en día, esta técnica no es muy común debido a que los sonidos que genera no suenan naturales. La sintetización a partir de sonidos grabados es bastante común.

La técnica de síntesis más común en la actualidad tal vez sea la **síntesis por modulación de frecuencias o síntesis FM**. En la síntesis FM, un oscilador (un objeto programado que genera una serie regular de salidas) controla (modula) las frecuencias con otras frecuencias. El resultado es un sonido más rico, menos metalizado o computarizado.

Otra técnica común es la **síntesis sustractiva**. En esta técnica el *ruido* se usa como entrada y luego se aplican *filtros* para eliminar las frecuencias no deseadas. De nuevo, el resultado es un sonido más rico, aunque por lo general no tanto como la síntesis FM.

¿Cuál es el objetivo de crear sonidos o música con computadoras? ¿De qué sirve cuando hay muchos sonidos, música y músicos excelentes en el mundo? El punto es que si desea

decirle a alguien más *cómo* obtuvo ese sonido, para que pueda replicar el proceso o incluso modificar el sonido de alguna forma (tal vez mejorarlo), un programa es la forma de hacerlo. En esencia, el programa captura y comunica un proceso: la forma en que se genera un sonido o una pieza de música.

### 8.6.1 MP3

En la actualidad, la mayoría de los archivos de audio en las computadoras son archivos MP3 (o tal vez MP4, o uno de sus tipos de archivos relacionados o descendientes). Los archivos MP3 son codificaciones de sonido (y video en algunos casos) que se basan en el estándar MPEG-3. Son archivos de audio, sólo que comprimidos de maneras especiales.

Una forma en que se comprimen los archivos MP3 es la **compresión sin pérdida**. Como sabemos, existen técnicas para almacenar datos que usan menos bits. Por ejemplo, sabemos que cada muestra tiene una anchura básica de dos bytes. ¿Qué pasaría si en vez de almacenar cada muestra almacenáramos la *diferencia* de la última muestra con respecto a la actual? La diferencia entre muestras es por lo general mucho más pequeña que de 32 776 a –32 768; podría ser +/– 1 000. Esto requiere menos bits para almacenarse.

Pero MP3 también usa la **compresión con pérdidas**. En realidad descarta una parte de la información del sonido. Por ejemplo, si hay un sonido muy suave justo después o al mismo tiempo que con un sonido bastante fuerte, no podrá escuchar el sonido suave. Una grabación *análoga* (el tipo que se usa en los discos) mantiene todas esas frecuencias. MP3 descarta las que realmente no podemos escuchar. Las grabaciones análogas son distintas de las grabaciones digitales en cuanto a que graban el sonido en forma continua, mientras que las digitales toman muestras en intervalos de tiempo.

Los archivos WAV están comprimidos, pero no tanto como los MP3, y sólo usan técnicas sin pérdidas. Los archivos MP3 tienden a ser mucho más pequeños que el mismo sonido en formato WAV. Los archivos AIFF son similares a los archivos WAV.

### 8.6.2 MIDI

MIDI es la *Interfaz digital de instrumentos musicales*. En realidad es un conjunto de acuerdos entre los fabricantes de dispositivos de música de computadora (secuenciadores, sintetizadores, cajas de ritmos, teclados, etcétera.) para definir cómo trabajarán sus dispositivos en conjunto. Mediante el uso de MIDI podemos controlar varios sintetizadores y cajas de ritmo desde distintos teclados.

MIDI no se usa para codificar sonido tanto como para codificar música. MIDI no graba cómo suena algo, sino cómo se reproduce. En sentido literal, MIDI codifica información como: "Presione la tecla en el instrumento sintetizado *X* en el tono *Y*", y después, "Libere la tecla *Y* en el instrumento *X*". La calidad del sonido MIDI depende totalmente del sintetizador, el dispositivo que genera el instrumento sintetizado.

Los archivos MIDI tienden a ser muy pequeños. Las instrucciones como "Reproducir clave #42 en pista 7" sólo ocupan unos cinco bytes de espacio. Esto hace a los archivos MIDI más atractivos en comparación con los archivos de sonido más grandes. MIDI ha sido bastante popular para las máquinas de karaoke.

Una ventaja de MIDI en comparación con los archivos MP3 o WAV es que puede especificar mucha música en muy pocos bytes. Pero MIDI no puede grabar sonidos. Por ejemplo, si desea grabar el estilo de una persona específica al tocar un instrumento, o grabar a *alguien* cantando, no es conveniente usar MIDI. Para capturar sonidos reales necesita grabar las muestras actuales, por lo que debe usar MP3 o WAV.

La mayoría de los sistemas operativos modernos tienen sintetizadores integrados muy buenos. Podemos usarlos desde Python. JES tiene integrada una función llamada `playNote`, la cual recibe como entrada una nota MIDI, una duración (cuánto tiempo hay que reproducir el sonido) en milisegundos y una intensidad (con qué fuerza golpear la tecla) de 0 a 127. `playNote` siempre usará un instrumento similar al piano. Las notas MIDI corresponden a las teclas y no a las frecuencias. Do (C) en la primera octava es 1, DO sostenido (C#) es 2. Do (C) en la cuarta octava es 60, Re (D) es 62 y Mi (E) es 64.

He aquí un ejemplo simple de cómo reproducir algunas notas MIDI desde JES. Podemos usar ciclos `for` para especificar ciclos en la música.

**Programa 83: reproducción de notas MIDI (ejemplo)**

```
def cancion():
  playNote(60,200,127)
  playNote(62,500,127)
  playNote(64,800,127)
  playNote(60,600,127)
```

■

## RESUMEN DE PROGRAMACIÓN

| | |
|---|---|
| `if` | Permite a Python tomar decisiones. `if` evalúa si una expresión es verdadera o falsa (en esencia, todo lo que se evalúa como 0 es falso y todo lo demás es verdadero). El bloque que va después del `if` se ejecuta si la prueba es verdadera. |
| `int` | Devuelve la parte entera del valor de entrada, descartando cualquier cosa después del decimal. |
| `setMediaPath()` | Le permite elegir una carpeta para obtener y almacenar medios. |
| `getMediaPath(nombreArchivoBase)` | Recibe un nombre de archivo base como entrada, luego devuelve la ruta completa a ese archivo (suponiendo que esté en la carpeta de medios que usted estableció usando `setMediaPath()`). |
| `playNote` | Recibe como entrada la nota, duración e intensidad. Cada nota se representa como un valor entero de 0 a 127; el Do (C) central es 60. La duración se especifica en milisegundos. La intensidad también puede variar de 0 a 127. Si omite este parámetro, JES usará 64 para la intensidad. |

## PROBLEMAS

8.1 ¿Qué son cada uno de los siguientes términos?

1. MIDI
2. MP3

3. Análogo
4. Amplitud
5. Velocidad de muestreo

8.2 ¿Cuál es la diferencia entre compresión sin pérdidas y compresión con pérdidas? ¿Qué tipo de archivos de sonidos usan cada tipo de compresión?

8.3 Vuelva a escribir la función de eco (programa 71, pág. 196) para generar *dos* ecos de vuelta, cada `retraso` muestras anteriores. *Sugerencia:* comience su ciclo de índices en `2*retraso + 1`, luego acceda a una muestra de eco en `indice-retraso` y a otra en `indice - 2*retraso`.

8.4 Escriba una función de mezcla general que reciba los dos sonidos a mezclar como entrada y devuelva un nuevo sonido. También puede tomar el número de muestras a usar del primer sonido antes de que empiece la mezcla, además del número de muestras a mezclar.

8.5 ¿Qué tan largo es un sonido comparado con el original, cuando se duplica su frecuencia (programa 74, pág. 199)? ¿Qué tan largo es un sonido comparado con el original, cuando se copia sólo uno de cada cuatro valores de sonidos en el sonido de destino?

8.6 Escriba una función que cree un nuevo sonido con la mitad de un sonido, que después sume los dos sonidos y luego agregue la segunda mitad del segundo sonido. Esto es más fácil de hacer si los sonidos tienen la misma longitud.

8.7 Escriba una función que mezcle tres sonidos entre sí. Empiece con una parte del primer sonido, luego una mezcla de sonido1 y sonido2, y después una mezcla de sonido2 y sonido3, para finalizar con el resto de sonido3.

8.8 Mezcle unas palabras sobre algo de música. Empiece con la música al 75% y las palabras al 25%, para luego pasar gradualmente hasta que las palabras representen el 75% y la música el 25%.

8.9 Escriba una función para crear una onda de sonido tipo pirámide.

8.10 Escriba una función para crear una onda de sonido tipo diente de sierra.

8.11 Escriba una función para cambiar la frecuencia de un sonido 10 veces.

8.12 Genere otra versión de la función de desplazamiento para crear el destino de modo que sea tan grande como el sonido resultante. Por lo tanto, si el factor es menor a uno, crearía un sonido más grande y si es mayor a uno, un sonido más pequeño.

8.13 ¿Trabaja la función de desplazamiento con un factor de 0.3? Si no puede, ¿puede corregirla para que copie cada valor de origen 3 veces en el destino?

8.14 Los DJ de hip-hop giran tornamesas hacia delante y hacia atrás de modo que las secciones de sonido se reproduzcan hacia delante y hacia atrás con rapidez. Pruebe a combinar la reproducción hacia atrás (programa 68, pág. 186) y el desplazamiento de frecuencia (programa 74, pág. 199) para obtener el mismo efecto. Reproduzca un segundo de un sonido con rapidez hacia delante, después con rapidez hacia atrás, dos o tres veces (tal vez tenga que moverse con una rapidez mayor a la del doble de la velocidad).

8.15 Considere cambiar el bloque `if` en la receta de desplazamiento de frecuencia (programa 77, pág. 202) a `indiceOrigen = indiceOrigen - getLength(origen)`. ¿Cuál es la diferencia si sólo se establece el `indiceOrigen` a 0? ¿Es mejor o peor? ¿Por qué?

8.16 Si usa la receta de desplazamiento (programa 77) con un factor de 2.0 o 3.0, el sonido se repetirá o incluso se triplicará. ¿Por qué? ¿Puede corregir esto? Escriba la función `desplazarDur` que reciba varias muestras (o incluso segundos) para reproducir el sonido.

8.17 Use las herramientas de sonidos para averiguar el patrón característico de distintos instrumentos. Por ejemplo, los pianos tienden a tener un patrón opuesto a lo que hemos creado: las amplitudes *disminuyen* a medida que alcanzamos ondas senoidales más altas. Pruebe a crear una variedad de patrones y vea cómo suenan, además de cómo se ven.

8.18 Cuando los músicos trabajan con síntesis aditiva, lo común es que coloquen *envolventes* alrededor de los sonidos, e incluso alrededor de cada onda senoidal agregada. Una envolvente *cambia* la amplitud con el tiempo: podría empezar pequeña, luego crecer (con rapidez o lentitud), mantenerse en cierto valor durante el sonido y después bajar antes de que termine el sonido. A este tipo de patrón se le conoce algunas veces como *envoltura de ataque, caída y sostenido (ASD)*. Los pianos tienden a tener un ataque rápido y luego decaen con rapidez. Las flautas tienden a tener un ataque lento y se sostienen tanto como uno desee. Pruebe a implementar eso para los generadores de ondas senoidales y cuadradas.

8.19 Cree una función para reproducir una canción mediante el uso de MIDI.

## PARA PROFUNDIZAR

Los buenos libros sobre música de computadora dicen mucho acerca de cómo crear sonidos desde cero, como en este capítulo. Uno de los favoritos de Mark por su facilidad de comprensión es *Computer Music: Synthesis, Composition, and Performance* de Dodge y Jerse [11]. La "biblia" de música de computadora es *The Computer Music Tutorial* de Curtis Roads [35].

Una de las herramientas más poderosas para reproducción con este nivel de música de computadora es *CSound*. Es un sistema de síntesis de música de software, gratuito y totalmente multiplataforma. El libro de Richard Boulanger [7] tiene todo lo necesario para jugar con CSound.

CAPÍTULO 9

# Creación de programas más grandes


9.1 DISEÑO DE PROGRAMAS DE ARRIBA HACIA ABAJO (TOP-DOWN)

9.2 DISEÑO DE PROGRAMAS DE ABAJO HACIA ARRIBA (BOTTOM-UP)

9.3 PRUEBA DE SU PROGRAMA

9.4 TIPS SOBRE DEPURACIÓN

9.5 ALGORITMOS Y DISEÑO

9.6 EJECUCIÓN DE PROGRAMAS FUERA DE JES


### Objetivos de aprendizaje del capítulo

- Demostrar dos estrategias de diseño distintas: arriba-abajo (top-down) y abajo-arriba (bottom-up).
- Demostrar distintas estrategias de prueba, como la caja negra y la caja de cristal.
- Demostrar el uso de varias estrategias de depuración a la hora de detectar los problemas de programación.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar métodos para aceptar la entrada del usuario y generar resultados para éste.
- Usar una nueva estructura de iteración, el ciclo `while`.

Para muchos de los problemas que enfrentamos hasta ahora, los hemos resuelto con programas más o menos sencillos. Programas relativamente pequeños, con tal vez una docena de líneas de código pueden resolver muchos problemas además de lograr muchas creaciones interesantes. Pero existen muchos, muchos problemas y creaciones más que *podrían* resolverse con programas más grandes y complejos. De esto es de lo que trata este capítulo. Éstas son las cuestiones que se resuelven mediante el campo de la *ingeniería de software*.

Para escribir programas más grandes hay que resolver problemas relacionados con la administración de la actividad de programación en sí.

- ¿Qué líneas de programas vamos a escribir? ¿Cómo vamos a decidir qué funciones necesitamos? Ése es el proceso de *diseño*. Hay muchas metodologías para diseñar, pero las dos más comunes son **arriba-abajo (top-down)** y **abajo-arriba (bottom-up)**. En

el diseño arriba-abajo, debemos averiguar lo que se debe hacer, refinar los *requerimientos* hasta poder identificar las piezas y luego escribirlas (por lo general desde el nivel más alto primero). En el diseño abajo-arriba, empezamos con lo que sabemos y seguimos agregando elementos hasta obtener nuestro programa.

- Las cosas no funcionan a la primera. La programación se define como "el arte de depurar una hoja de papel en blanco".[1] La depuración es averiguar qué es lo que *no* funciona, por qué y cómo corregirlo. La programación y la *depuración* están conectadas de una manera intrincada. Aprender a depurar es una habilidad importante que conduce a averiguar cómo hacer programas que realmente se *ejecuten*.
- Incluso aunque las cosas funcionen la primera vez, es poco probable que todo haya salido perfecto. Los programas grandes constan de muchas partes. Necesitamos usar técnicas de *prueba* para asegurarnos de haber detectado todos (o incluso la mayoría de) los errores (*bugs*) en el programa.
- Aun después de probar y depurar, la mayoría de los programas grandes no están "terminados". Muchos programas grandes se usan durante un largo periodo para resolver problemas que surgen a menudo (como contabilidad y rastreo de inventario). Estos programas grandes *nunca* se terminan. Por el contrario, se agregan nuevas características o hay que eliminar los errores recién descubiertos. La etapa de *mantenimiento* del desarrollo de programas continúa mientras el programa esté en uso. En general, durante el transcurso de la vida de un programa, el mantenimiento es *por mucho* la parte más costosa.

## 9.1 DISEÑO DE PROGRAMAS DE ARRIBA HACIA ABAJO (TOP-DOWN)

El diseño arriba-abajo es la forma como la mayoría de las disciplinas de ingeniería recomiendan realizar el diseño. Para empezar hay que desarrollar una lista de *requerimientos*: lo que se debe hacer, en idioma español o mediante matemáticas, que pueda refinarse de manera iterativa. Refinar los requerimientos significa hacerlos más claros y específicos. El objetivo de refinar los requerimientos en el diseño arriba-abajo es llegar al punto en donde las instrucciones de los requerimientos puedan implementarse de manera directa como código de programa.

El proceso arriba-abajo es preferido debido a su facilidad de comprensión, además de que podemos realizar una planeación: es en verdad lo que hace posible el negocio del software. Imagine trabajar con un cliente que desea que usted programe algo. Usted recibe el planteamiento del problema y luego trabaja con el cliente para refinar ese planteamiento en un conjunto de requerimientos. Luego se pone a crear el programa. Si el cliente no está contento con ese programa, puede probar para ver si el software cumple con los requerimientos. Si lo hace y el cliente aceptó esos requerimientos, entonces usted cumplió con su contrato. Si no lo hace, entonces usted necesita hacer que cumpla con los requerimientos, pero no necesariamente debe cumplir con las necesidades del cliente que hayan cambiado.

A continuación le mostramos los detalles sobre el proceso:

- Empiece con el planteamiento del problema. Si no tiene uno, escríbalo. ¿Qué está tratando de hacer?

[1]Definición de "programación" de The Jargon File v. 4.4.8. The Jargon File © 2003 por Eric S. Raymond. Reimpreso con permiso.

- Empiece a refinar el planteamiento del problema. ¿El programa consta de varias partes? Tal vez debería usar la **descomposición jerárquica** para definir subfunciones. ¿Está consciente de que tiene que abrir algunas imágenes o establecer algunos valores constantes? ¿Hay algunos ciclos? ¿Cuántos necesita?
- Siga refinando el planteamiento del problema hasta obtener instrucciones, comandos o funciones que conozca (o que sepa cómo escribir).
- Cuando trabaje en un programa más grande, es muy probable que vaya a usar funciones que llamen a otras funciones (*subfunciones*). Empiece por escribir las funciones que podría llamar desde el área de comandos y después las funciones de nivel inferior hasta que llegue a las subfunciones.

### 9.1.1 Un ejemplo de diseño arriba-abajo

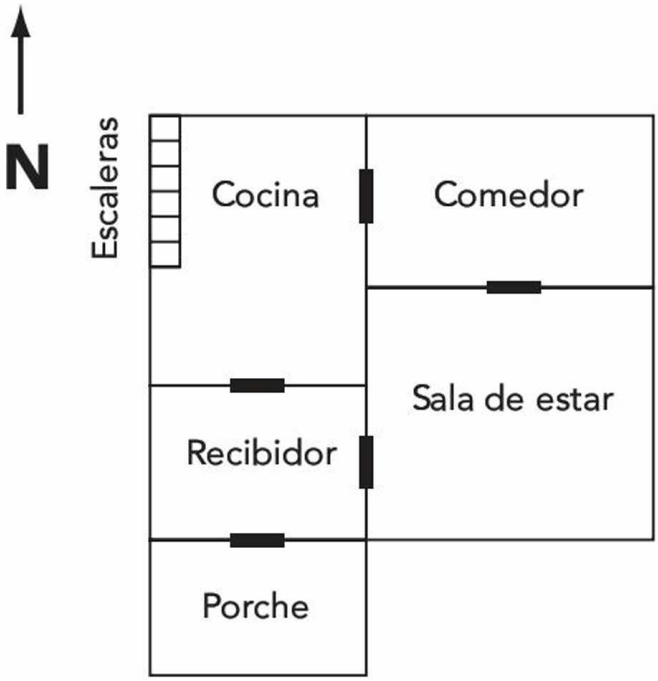
Si seguimos este proceso y definimos funciones (*procedimientos*) en vez de líneas de código, estaríamos usando lo que se conoce como **abstracción procedural**. Aquí definimos funciones de alto nivel que llaman a funciones de nivel inferior. Estas funciones de nivel inferior son fáciles de escribir y probar, al tiempo que las funciones de nivel superior se hacen más fáciles de leer, ya que lo único que hacen es llamar a las funciones de nivel inferior.

En los primeros capítulos del libro vimos algo de abstracción procedural. Un ejemplo es cuando redefinimos el collage en términos de funciones de nivel inferior que hicieron la copia de los píxeles por nosotros. Estas funciones de nivel inferior son una forma de abstracción. Al asignar a esas líneas de código el nombre de una función, podemos dejar de pensar en términos de esas líneas individuales y usar un nombre significativo para esas líneas. Al proveer parámetros a la función, la hacemos reutilizable.

Ahora vamos a crear un *juego de aventuras* sencillo. Un juego de aventuras es un tipo de videojuego en donde el jugador explora un mundo, se mueve entre los salones y espacios en el juego mediante el uso de comandos como "ir al norte" y "tomar la llave". A menudo hay acertijos por resolver y batallas por entablar. William Crowther escribió el juego de aventuras original en 1970; Don Woods lo extendió más tarde. Este juego estaba basado en la exploración de un conjunto de cuevas. El género se hizo popular en la década de 1980 con los juegos de Infocom como *Zork* y *Hitchiker's Guide to the Galaxy*. Los juegos de aventuras modernos tienden a ser gráficos, como *Dreamfall* y *Portal*. Los juegos de aventuras originales basados en texto mezclaron las técnicas de jugar y de contar historias, que es lo que todos buscamos.

En este punto, necesitamos definir nuestro problema. ¿Qué tipo de juego de aventura vamos a construir? ¿De qué tamaño? ¿Qué es lo que podrá hacer el usuario? Queremos definir el problema dentro de las limitaciones de lo que podemos desarrollar.

Vamos a construir un juego de aventuras en donde el usuario pueda recorrer los salones y eso será todo. No habrá acertijos ni un juego de verdad. Queremos sólo un ejemplo simple. He aquí un bosquejo de cómo podríamos distribuir un conjunto simple de salones. Para hacerlo interesante, vamos a pretender que es un escenario de una historia de terror o de suspenso.




### 9.1.2 Diseño de la función de nivel superior

¿Cómo funcionará nuestro programa? Podemos idear una descripción general de cómo podría ejecutarse:

1. Para empezar, decimos al jugador las bases del juego.
2. Describimos el salón al jugador. Para empezar vamos a colocar al jugador en un salón específico.
3. Esperamos a que el jugador escriba un comando ("norte" o "salir").
4. Averiguamos a cuál salón va a entrar el usuario, con base en el comando (dirección) que eligió.
5. Repetimos desde el paso 2 hasta que el usuario diga que desea salir.

Podemos escribir la función que hace estas cosas ahora mismo. Pero necesitamos conocer un par de funciones adicionales en Python que no hemos visto todavía.

- `printNow` es una función[1] que recibe una cadena como entrada y después la imprime en el área de comandos *tan pronto como se ejecuta*. Esto contrasta con `print`, que no muestra nada en el área de comandos sino hasta que el programa termina de ejecutarse. `printNow` es útil para mostrar cosas durante el juego.
- Para obtener entrada del usuario, podemos usar `requestString`. Esta función recibe como entrada un indicador que aparecerá en la ventana solicitando la entrada del

[2]En Python 3.0, incluso `print` es una función, por lo que no es tan extraño como podría parecer.

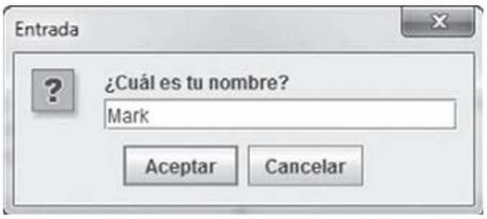



**FIGURA 9.1**
La apariencia del cuadro de diálogo de `requestString`.

usuario (figura 9.1) (en Python, la habilidad de leer una entrada de cadena del usuario se conoce como `raw_input`). La función devuelve la cadena que escribió el usuario.

```
>>> print requestString("¿Cuál es tu nombre?")
Mark
```

- El mayor desafío es cómo seguir repitiendo algunas líneas de código de manera indefinida hasta que ocurra algo específico. No podemos hacer esto con un ciclo `for`.

  En cambio usaremos un nuevo tipo de ciclo, conocido como ciclo `while`. Un ciclo while realiza una prueba (al igual que un `if`). Lo diferente en comparación con el `if` es que el ciclo `while` repite el cuerpo del ciclo de manera *indefinida* hasta que la prueba se vuelva falsa. El siguiente código se ejecuta cuando `x` es 0, 1 y 2, pero cuando `x` es 3, no es cierto que `x < 3`, por lo que el ciclo no se ejecuta.

```
>>> x = 0
>>> while (x < 3):
...         print "Contando..."
...         x = x + 1
...
Contando...
Contando...
Contando...
```

Con estas tres piezas nuevas, podemos escribir la función que corresponda a la descripción que vimos antes.

**Programa 82: función de nivel superior del juego de aventuras**

```
def jugarJuego():
  ubicacion = "Porche"
  mostrarIntroduccion()
  while not (ubicacion == "Salir") :
    mostrarSalon(ubicacion)
    direccion = requestString("¿Cuál dirección?")
    ubicacion = elegirSalon(direccion, ubicacion)
```

■

Esta función es bastante parecida a la descripción que presentamos antes, línea por línea. Lo que puede parecer extraño aquí es que no hemos visto ni escrito algunas de estas funciones, como `mostrarIntroduccion`, `mostrarSalon` y `elegirSalon`. Éste es uno de los puntos del diseño arriba-abajo: podemos hacer el *plan* de cómo debe funcionar todo el

programa y las funciones que necesitaremos en el programa completo, mucho antes de escribir todas las partes.

### Cómo funciona

- Usaremos la variable `ubicacion` para almacenar el salón en donde se encuentra el jugador en un momento dado. Empezaremos en el "Porche".
- La función `mostrarIntroduccion` mostrará las instrucciones al usuario.
- Usaremos la ubicación "Salir" para representar la solicitud del usuario de salir del juego. Mientras que la ubicación *no sea* "Salir", seguiremos jugando.
- Al empezar cada turno, mostramos la descripción del salón actual. La función `mostrarSalon` mostrará la descripción del salón, usando la `ubicacion` del jugador como entrada para indicar el salón que se va a mostrar.
- Obtenemos la solicitud del usuario de una nueva `direccion` a través de `requestString`.
- Elegimos un nuevo salón para el usuario mediante la función `elegirSalon`, con base en las entradas de la `direccion` solicitada y la `ubicacion` del salón actual.

Observe que no sabemos *nada* sobre la forma en que trabajarán estas subfunciones. No es posible saber cómo funcionarán justo ahora, ya que no las hemos escrito todavía. Lo que esto significa es que esta función de nivel superior, `jugarJuego`, está *desacoplada* de las subfunciones de nivel inferior. Las definimos en términos de las entradas, salidas y lo que deben hacer. Alguien más podría escribir estas funciones por nosotros ahora. Ésa es otra razón por la que el diseño de nivel superior se utiliza con tanta frecuencia para ingeniería.

- Facilita el mantenimiento, ya que las distintas piezas pueden cambiar sin tener que cambiar el todo en general.
- Planeamos las funciones antes de escribirlas.
- Podemos dejar que distintos programadores trabajen juntos en el mismo programa, con base en el plan.

## 9.1.3 Escritura de las subfunciones

Ahora que tenemos el plan, podemos empezar a escribir el resto de las subfunciones para hacer que esto funcione. Por ahora colocaremos todas estas funciones juntas en el archivo, para que funcione correctamente. También podríamos colocar funciones útiles en un archivo separado e importarlas mediante la instrucción `import`. Por ahora vamos a completar nuestro diseño y la implementación del juego de aventuras.

**Programa 83: `mostrarIntroduccion` para el juego de aventuras**

```
def mostrarIntroduccion():
  printNow("¡Bienvenido a la casa de aventura!")
  printNow("En cada salón le indicaremos hacia
     dónde puede ir.")
  printNow("Puede avanzar hacia el norte, sur, este u oeste, escribiendo
     esa dirección.")
  printNow("Escriba ayuda para volver a mostrar esta introducción.")
  printNow("Escriba terminar o salir para terminar el programa.")
```

■

### Cómo funciona

Lo único que hace la introducción es mostrar información para el usuario/jugador, por medio de la función `printNow`. Le decimos al jugador cómo moverse (escribiendo una dirección) y cómo obtener ayuda o salir. Observe que, al hacer esto, estamos definiendo aún más las funciones anteriores. En nuestra función `elegirSalon`, tenemos que manejar entradas como "ayuda", "terminar" y "salir".

**Programa 84: `mostrarSalon` para el juego de aventuras**

```
def mostrarSalon(salon):
  if salon == "Porche":
    mostrarPorche()
  if salon == "Recibidor":
    mostrarRecibidor()
  if salon == "Cocina":
    mostrarCocina()
  if salon == "SalaDeEstar":
    mostrarSE()
  if salon == "Comedor":
    mostrarComedor()
```

■

### Cómo funciona

Podríamos convertir a `mostrarSalon` en una función extensa con muchas llamadas a la función `printNow`. Sin embargo, esto sería tedioso de escribir y desafiante de mantener. ¿Qué tal si tuviéramos que cambiar la descripción de la sala de estar? Tendríamos que recorrer toda una extensa función y encontrar la instrucción `printNow` correcta para modificarla. O, podríamos simplemente modificar la función `mostrarSE`. Así es como lo hicimos en nuestro ejemplo.

**Programa 85: `elegirSalon` para el juego de aventuras**

```
def elegirSalon(direccion, salon):
  if (direccion == "terminar") or (direccion == "salir"):
    printNow("¡Adiós!")
    return "Salir"
  if direccion == "ayuda":
    mostrarIntroduccion()
    return salon
  if salon == "Porche":
    if direccion == "norte":
      return "Recibidor"
  if salon == "Recibidor":
    if direccion == "norte":
      return "Cocina"
    if direccion == "este":
      return "SalaDeEstar"
    if direccion == "sur":
      return "Porche"
  if salon == "Cocina":
```

```
    if direccion == "este":
      return "Comedor"
    if direccion == "sur":
      return "Recibidor"
  if salon == "SalaDeEstar":
    if direccion == "oeste":
      return "Recibidor"
    if direccion == "norte":
      return "Comedor"
  if salon == "Comedor":
    if direccion == "oeste":
      return "Cocina"
    if direccion == "sur":
      return "SalaDeEstar"
```

■

### Cómo funciona

Esta función está definida por el mapa. Dados el salón actual y la dirección deseada, esta función devuelve el nombre del nuevo salón en el que se ubicará el jugador.

**Programa 86: mostrar salones en el juego de aventuras**

```
def mostrarPorche():
  printNow("Usted está en el porche de una escalofriante casa.")
  printNow("Las ventanas están rotas. Es una noche oscura y tempestuosa.")
  printNow("Puede ir hacia el norte y entrar en la casa. Si se atreve.")

def mostrarRecibidor():
  printNow("Usted está en el recibidor de la casa. Hay telarañas¬
  en el rincón.")
  printNow("De pronto tiene una sensación de temor.")
  printNow("Hay un pasaje hacia el norte y otro hacia el este.")
  printNow("El porche queda detrás de usted hacia el sur.")

def mostrarCocina():
  printNow("Usted está en la cocina. Todas las superficies están cubiertas con
  ollas, sartenes, pedazos de comida y charcos de sangre.")
  printNow("Cree escuchar algo proveniente de las escaleras que van hacia el
  lado oeste del cuarto.")
  printNow("Se escucha que algo raspa, como si se arrastrara por el¬
  piso.")
  printNow("Puede ir hacia el sur o el este.")

def mostrarSE():
  printNow("Usted se encuentra en una sala de estar. Hay sofás, sillas y mesas¬
  pequeñas.")
  printNow("Todo está cubierto con polvo y telarañas.")
  printNow("De repente, escucha un ruido estremecedor en otro salón.")
  printNow("Puede ir hacia el norte o hacia el oeste.")
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un comando individual en Python no puede dividirse entre varias líneas.

```
def mostrarComedor():
  printNow("Usted está en el comedor.")
  printNow("Hay restos de una comida en la mesa. No puede identificar lo que¬
  es, y tal vez no quiera hacerlo.")
  printNow("¿Acaso se escuchó un golpe proveniente del oeste?")
  printNow("Puede ir hacia el sur o hacia el oeste")
```

¬ Estas líneas del programa deben continuar con las siguientes líneas.
Un comando individual en Python no puede dividirse entre varias líneas.

■

## Cómo funciona

Para cada salón, hay sólo unas cuantas invocaciones a `printNow` para describir ese salón. Desde el punto de vista programático, éstas son funciones muy simples. Desde la perspectiva del autor, aquí es donde están la diversión y la creatividad.

Ahora tenemos suficiente para jugar nuestro juego. Para empezar, vamos a invocar a nuestra función de nivel superior, escribiendo `jugarJuego()`. Las descripciones aparecen en el área de comandos y los indicadores aparecen en un cuadro de diálogo por encima del juego (figura 9.2). A continuación le mostramos un ejemplo de la ejecución del programa:

```
>>> jugarJuego ()
¡Bienvenido a la casa de aventura!
En cada salón le indicaremos hacia dónde puede ir.
Puede avanzar hacia el norte, sur, este u oeste, escribiendo esa dirección.
Escriba ayuda para volver a mostrar esta
    introducción.
Escriba terminar o salir para terminar el programa.
```

**FIGURA 9.2**
Pantalla del juego de aventuras

```
Usted está en el porche de una escalofriante casa.
Las ventanas están rotas. Es una noche oscura y tempestuosa.
Puede ir hacia el norte y entrar en la casa. Si se atreve.
Usted está en el recibidor de la casa. Hay telarañas en el rincón.
De pronto tiene una sensación de temor.
Hay un pasaje hacia el norte y otro hacia el este.
El porche queda detrás de usted hacia el sur.
Usted se encuentra en una sala de estar. Hay sofás, sillas
    y mesas pequeñas.
Todo está cubierto con polvo y telarañas.
De repente, escucha un ruido estremecedor en otro salón.
Puede ir hacia el norte o hacia el oeste.
¡Adiós!
```

## 9.2 DISEÑO DE PROGRAMAS DE ABAJO HACIA ARRIBA (BOTTOM-UP)

El diseño abajo-arriba es un proceso distinto que básicamente termina en el mismo lugar. Empezamos con una idea de lo que deseamos hacer; podríamos llamarla un planteamiento del problema. Pero en vez de refinar el problema, nos enfocamos en crear el programa de solución. Es conveniente reutilizar código de otros programas lo más que sea posible.

Lo más importante que debemos hacer en el diseño abajo-arriba es *probar* nuestro programa con *mucha* frecuencia. ¿Hace lo que queremos? ¿Hace lo que *esperamos*? ¿Tiene sentido? Si no es así, hay que agregar instrucciones `print` y explorar el código hasta comprender lo que hace. Si no sabemos lo que hace, no podemos cambiarlo en lo que queremos que haga.

He aquí una descripción común del proceso abajo-arriba, partiendo del planteamiento de un problema:

- ¿Qué tanto del problema sabe ya cómo solucionar? ¿Cuánto puede lograr mediante otros programas que ha escrito? Por ejemplo, ¿el problema pide manipular sonido? Pruebe un par de recetas del libro para que recuerde cómo se hace eso. ¿Acaso le pide cambiar los niveles de rojo? ¿Puede encontrar una receta que haga eso y probarla?
- Ahora, ¿puede juntar algunas de estas piezas (que pueda escribir o tomar prestadas de algunos otros programas)? ¿Puede reunir un par de recetas que hagan una *parte* de lo que desea?
- Siga aumentando el tamaño del programa. ¿Se acerca más a lo que usted necesita? ¿Qué más tiene que agregar?
- Ejecute su programa con *frecuencia*. Asegúrese de que funcione y de comprender todo lo que tiene hasta ahora.
- Repita el proceso hasta quedar satisfecho con el resultado.

### 9.2.1 Ejemplo de un proceso abajo-arriba

La mayoría de los ejemplos de este libro están desarrollados mediante un proceso abajo-arriba. La forma en que realizamos la sustracción del fondo y la técnica chromakey son un buen ejemplo. Empezamos con la idea de quitar el fondo de alguien y colocarlo en una nueva imagen. ¿En dónde empezamos?

Podemos imaginar que una parte del problema es buscar todos los píxeles que sean parte de la persona o del fondo. Hemos hecho cosas así antes, como cuando buscamos todo el café

en el cabello de Katie para convertirlo en rojo (programa 36, página 110). Eso nos dice que es conveniente verificar si hay una `distancia` lo bastante grande entre el color de una persona y un color de fondo, y de ser así, queremos traer un nuevo color en el pixel de fondo: el pixel en el mismo punto. También hemos hecho cosas así, como cuando copiamos imágenes en lienzos.

En este punto es probable que podamos escribir algo como esto:

```
if distancia(colorPersona,colorFondo)<unValorDeUmbral:
  colorFondo = getColor(getPixel(nuevoFondo,x,y))
  setColor(getPixel(imagenPersona,x,y),colorFondo)
```

Ésta es la parte fundamental de la receta de sustracción de fondo. El resto simplemente trata sobre establecer las variables (programa 46, página 125). Pero luego, cuando probamos esa receta, no funcionaba muy bien por varias razones. Eso es lo que nos condujo a la técnica de chromakey (programa 47, página 128). Chromakey es una mejor forma de averiguar qué píxeles forman parte del fondo y cuáles pertenecen al primer plano, pero el proceso básico es el mismo que intercambiar el fondo; por lo tanto, la mayor parte del programa es reutilizable en el nuevo contexto.

Aquí el proceso clave es tomar ideas (o incluso líneas de código) de otros proyectos y combinarlas, *probando* lo que estamos haciendo todo el tiempo. La programación abajo-arriba es muy parecida al proceso de "depurar usando una hoja de papel en blanco". La depuración es una habilidad imprescindible en el diseño o programación abajo-arriba.

## 9.3 PRUEBA DE SU PROGRAMA

Es bastante difícil probar bien los programas, en especial para un nuevo programador. Usted escribió el código, ¡por lo que supone que va a hacer lo que usted escribió! Para convertirse en un excelente probador es necesario contar con una gran porción de humildad. Debe aceptar el hecho de que tal vez haya escrito algo en forma incorrecta, o que tal vez no haya comprendido bien *qué* debería escribir.

Hay dos metodologías para probar programas. Una se conoce como *prueba de la caja de cristal*, que es en donde hay que probar toda ruta posible a través de nuestro programa. La metodología se llama "caja de cristal" debido a que en verdad analizamos el programa y pensamos cómo probar cada una de las líneas del mismo. Puesto que conocemos la estructura de nuestro programa, lo probamos de acuerdo con ella.

Para usar la prueba de la caja de cristal en nuestro juego de aventuras, debemos recorrer todas las puertas en ambas direcciones. Por ejemplo, vaya del porche al norte hacia el recibidor, luego regrese al porche por el sur. ¿Llegó al lugar correcto en ambas ocasiones? ¿Aparecieron correctamente los salones? Debemos probar lo que ocurre en respuesta a los comandos "ayuda", "terminar" y "salir". Esto nos asegurará que hayamos probado todas y cada una de las líneas de nuestro programa.

La segunda metodología se llama *prueba de la caja negra*, en donde no consideramos cómo está escrito el programa, sino la forma en que se supone debe comportarse y, en especial, en respuesta a las entradas tanto válidas como *inválidas*. El jugador no siempre escribirá los comandos correctos. Puede llegar a cometer un error y escribir mal algo, o probar un comando que crea que es razonable pero que usted no consideró en la programación.

Por ejemplo, usemos la prueba de la caja negra con la función `elegirSalon`. Primero evaluaremos todas las entradas correctas a `elegirSalon`, como se indica a continuación:

```
>>> elegirSalon("norte","Porche")
"Recibidor"
>>> elegirSalon("norte","Recibidor")
"Cocina"
```

Ahora probemos algunas entradas inválidas: palabras mal escritas e ideas incorrectas.

```
>>> elegirSalon("nrte","Porche")
>>> elegirSalon("Recibidor","Porche")
```

Éste es un verdadero problema. En respuesta a una entrada inválida, `elegirSalon` no devuelve nada. Cuando tratemos de establecer la variable `ubicacion` a partir de la respuesta, no podremos obtener un salón válido. La variable `ubicacion` estará vacía. Además, el jugador no obtendrá ninguna respuesta de que algo salió mal.

Necesitamos modificar `elegirSalon` de modo que, si ninguna otra respuesta coincide, se realice una contestación razonable y se devuelva un valor razonable. Quizás la respuesta más razonable sea devolver el mismo salón: dejar al jugador en donde está.

Ahora, nuestro programa trabajará de una forma un poco más razonable en respuesta a una entrada incorrecta.

```
>>> elegirSalon("nrte","Porche")
```

**Programa 87: función `elegirSalon` mejorada para el juego de aventuras**

```
def elegirSalon(direccion, salon):
  if (direccion == "terminar") or (direccion == "salir"):
    printNow("¡Adiós!")
    return "Salir"
  if direccion == "ayuda":
    mostrarIntroduccion()
    return salon
  if salon == "Porche":
    if direccion == "norte":
      return "Recibidor"
  if salon == "Recibidor":
    if direccion == "norte":
      return "Cocina"
    if direccion == "este":
      return "SalaDeEstar"
    if direccion == "sur":
      return "Porche"
  if salon == "Cocina":
    if direccion == "este":
      return "Comedor"
    if direccion == "sur":
      return "Recibidor"
  if salon == "SalaDeEstar":
    if direccion == "oeste":
      return "Recibidor"
    if direccion == "norte":
      return "Comedor"
```

```
if salon == "Comedor":
  if direccion == "oeste":
    return "Cocina"
  if direccion == "sur":
    return "SalaDeEstar"
printNow("No puede (o no es conveniente) ir en esa dirección.")
return salon
```

■

```
No puede (o no es conveniente) ir en esa dirección.
'Porche'
>>> elegirSalon("Recibidor","Porche")
No puede (o no es conveniente) ir en esa dirección.
"Porche"
```

El primer enunciado aquí dice que no está permitida la dirección introducida y que la segunda línea ('Porche') es el salón en el que se encuentra el jugador en ese momento.

### 9.3.1 Prueba de las condiciones límite

Los programadores profesionales prueban cada programa en forma exhaustiva, para asegurarse de que funcione de la manera esperada. Uno de los elementos de la caja negra que se concentran en evaluar se relaciona con las *condiciones límite*. ¿Cuál es la entrada más pequeña con la que debe trabajar el programa? ¿Cuál es la entrada más grande con la que debe trabajar el programa? Asegurarse de que el programa pueda trabajar con las entradas más pequeña y más grande posibles es a lo que nos referimos con evaluar las condiciones límite.

También puede usar esta estrategia para evaluar sus programas que manipulen medios. Digamos que su programa de manipulación de imágenes falla (genera un error o no parece detenerse) con una imagen específica y usted ha intentado rastrear el programa, pero no puede averiguar por qué está fallando. Pruebe con una imagen distinta. ¿Funciona con una imagen más pequeña? ¿Qué tal con una imagen vacía (todo blanco o todo negro)? Tal vez descubra que su programa funciona, pero es tan lento que no pensó que funcionara con la imagen más grande.

Algunas veces las funciones que manipulan índices (por ejemplo, los programas para aumentar o reducir escalas) pueden fallar en programas de cierto tamaño, pero no en otros. Por ejemplo, los programas de reflejo pueden funcionar con índices de números impares pero no con números pares. Pruebe con entradas de distintos tipos de tamaños para ver cuáles tienen éxito o fracasan.

## 9.4 TIPS SOBRE DEPURACIÓN

¿Cómo puede averiguar lo que está haciendo su programa, si se ejecuta pero no hace lo que usted quiere? Éste es el proceso de *depuración*: averiguar lo que su programa está haciendo, la diferencia que hay entre eso y lo que usted *desea* que haga, y modificar el programa para que pueda hacer lo que usted quiere.

Partir de un mensaje de error es la forma *más sencilla* de depuración. Al tener una indicación de Python sobre el error, tenemos una idea (un número de línea) de la posible ubicación del error. Esto nos indica en dónde buscar para corregir el problema y hacer que el error desaparezca.

La depuración se vuelve un poco más compleja cuando el programa funciona pero no hace lo que queremos. Ahora hay que averiguar lo que hace el programa y lo que queremos que haga en realidad.

El primer paso es *siempre* **averiguar qué está haciendo el programa**. Sin importar que haya mensajes de error implícitos o no, esto es siempre lo primero que debe hacer. Si recibe un error, la pregunta importante es por qué el programa funcionó hasta ese punto y qué valores de las variables estuvieron presentes en ese punto de tal forma que ocurriera el error.

**Idea de ciencias computacionales: ¡aprenda a rastrear código!**

Lo más importante que puede hacer al depurar sus programas es ser capaz de rastrear su código. Piense en su programa de la manera en que lo hace la computadora. Recorra cada línea y averigüe lo que hace.

■

Para empezar la depuración hay que recorrer el código, por lo menos las líneas alrededor de donde está ocurriendo el error. ¿Qué nos dice el error? ¿Qué podría estar provocándolo? ¿Cuáles son los valores de las variables antes y después de ese punto? La pregunta interesante es por qué ocurrió el error *en ese momento*. ¿Por qué no ocurrió en un punto anterior en el programa?

Si es posible, ejecute el programa. Siempre es más fácil hacer que la computadora nos *indique* lo que está ocurriendo en vez de tener que averiguarlo por nuestra cuenta. Ahora bien, el simple hecho de ejecutar sus funciones no le dará la respuesta. Agregue instrucciones `print` en su código para que le muestren el valor de las variables.

**Tip de depuración: ¡las instrucciones `print` son sus amigas!**

Use la instrucción `print` para imprimir lo que ocurre en sus programas. Haga esto cuando no pueda averiguar qué ocurre mediante el rastreo del programa. Imprima los valores de las ecuaciones que sean demasiado complejas. Imprima instrucciones simples como "¡Estoy en esta función ahora!" para saber en qué momento llega a las funciones que cree que está llamando. Deje que la computadora le *indique* lo que está haciendo.

■

Algunas veces, sobre todo en un ciclo, es conveniente usar `printNow`. Como esta instrucción imprime la cadena de entrada en el área de comandos *tan pronto como ocurre*, es más útil que `print` para la depuración. Nos conviene poder ver qué está ocurriendo *en el momento exacto* en que ocurre.

**Tip de depuración: no tenga miedo de modificar el programa**

Guarde una nueva copia de su programa y después elimine todas las partes que lo confundan. ¿Puede hacer que el resto del código se ejecute? Ahora empiece a agregar piezas de vuelta (copiar-pegar) de la copia original de su programa. Cambiar el programa de modo que sólo ejecute parte de éste en un momento dado es una excelente manera de averiguar qué está ocurriendo.

■

### 9.4.1 Averiguar cuál es la instrucción que nos debe preocupar

Con frecuencia, los errores más difíciles de descubrir son los que *aparentan* estar bien. Los errores de espaciado y los paréntesis que no coinciden entran en esta categoría. Estos errores son bastante difíciles de encontrar en los programas grandes, en donde el error sólo indica

que hay un problema "en alguna parte" alrededor de una línea dada, pero Python no sabe en dónde con exactitud.

Una estrategia consagrada para averiguar qué está mal es deshacerse de todas las instrucciones de las que estemos seguros. Sólo hay que colocar un "#" en frente de las instrucciones que usted crea que están bien. Si comenta una instrucción `if` o `for`, asegúrese de comentar también el bloque después de la instrucción. Ahora intente de nuevo.

Si el error desaparece, entonces se equivocó: en realidad comentó las instrucciones en donde está el problema. Quite el comentario a unas cuantas líneas e intente de nuevo. Cuando el mensaje de error vuelva a aparecer, significa que el error está en una de las líneas a las que acaba de quitarles el comentario.

Si el error sigue ahí, ahora sólo tiene que revisar unas cuantas instrucciones: las que siguen sin comentar. En un momento dado, el error desaparecerá o quedará una sola línea sin comentar. De cualquier forma, ahora puede averiguar en dónde está el error.

### 9.4.2 Ver las variables

Además de imprimir, existen otras herramientas integradas a JES para ayudarle a averiguar qué están haciendo sus programas. La función `showVars` le mostrará todas las variables y sus valores en el punto en el que se ejecute (figura 9.3). Le mostrará las variables en el contexto *actual* y las variables en el contexto *global* (accesibles incluso desde el área de comandos). Puede usar `showVars()` en el área de comandos para ver las variables que creó ahí; tal vez olvidó sus nombres o quería ver cuáles eran los valores en varias variables al mismo tiempo.

La otra herramienta poderosa en JES es el *Vigilante* (*Watcher*). El Vigilante le permite ver qué líneas se están ejecutando *a medida* que lo van haciendo. La figura 9.4 muestra al Vigilante ejecutando el código que se muestra a continuación: la función `crearAtardecer()` del programa 13 (página 63). Sólo tenemos que abrir el Vigilante (desde el menú DEBUG [Depurar] o desde el botón WATCHER [Vigilante]) y luego usar el área de comandos de la manera usual. Cada vez que ejecutemos nuestras propias funciones, se ejecutará el Vigilante.

```
def crearAtardecer(imagen):
  reducirAzul(imagen)
  reducirVerde(imagen)

def reducirAzul(imagen):
  for p in getPixels(imagen):
    valor=getBlue(p)
    setBlue(p,valor*0.7)

def reducirVerde(imagen):
  for p in getPixels(imagen):
    valor=getGreen(p)
    setGreen(p,valor*0.7)
```

Podemos detener la ejecución mediante PAUSE y luego ir paso a paso desde ahí con STEP. También podemos detener la ejecución con STOP, avanzar a velocidad completa mediante FULL SPEED o incluso establecer una velocidad desde lento hasta rápido. Cuando tenga el Vigilante abierto, el programa se *ejecutará* con menos rapidez. Entre más rápido se ejecute el programa, se mostrará menos información (es decir, no aparecerán todas las líneas ejecutadas en el Vigilante).

Además de recorrer paso a paso la ejecución y ver qué instrucciones se ejecutan en un momento dado, también podemos observar variables específicas. Después de hacer clic en ADD VARIABLE (Agregar variable), el Vigilante le pedirá el nombre de la variable. Después,

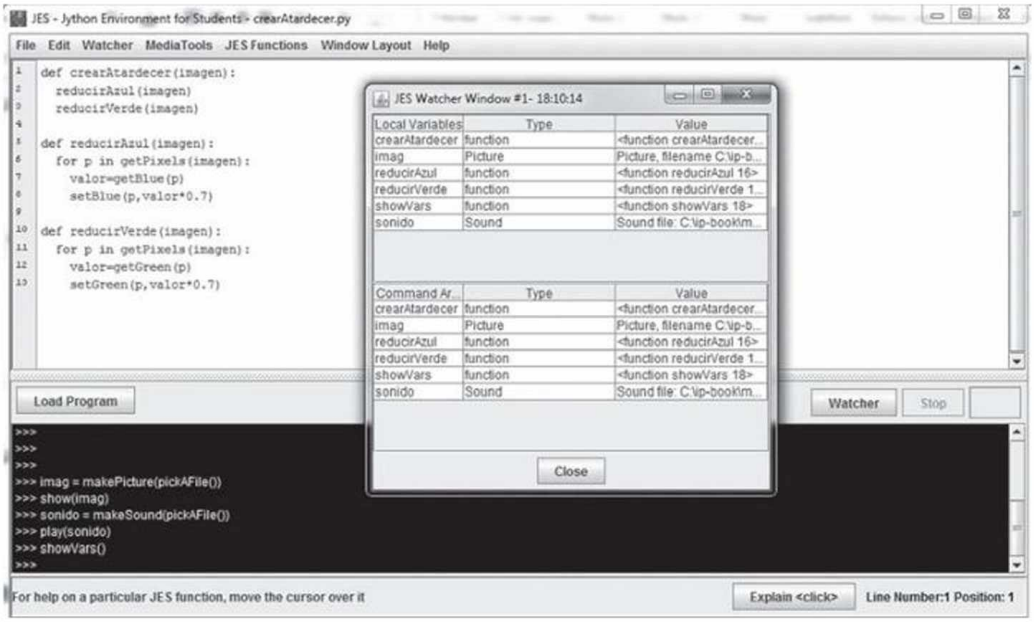



**FIGURA 9.3**
Visualización de variables en JES.

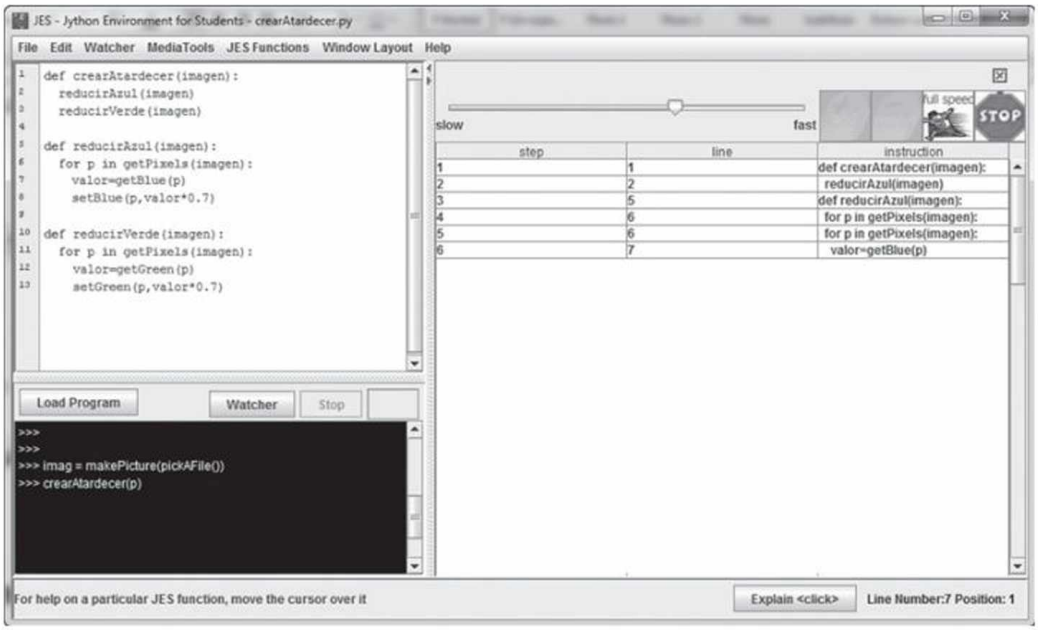



**FIGURA 9.4**
Recorrido paso a paso por la función `crearAtardecer()` con el Vigilante.

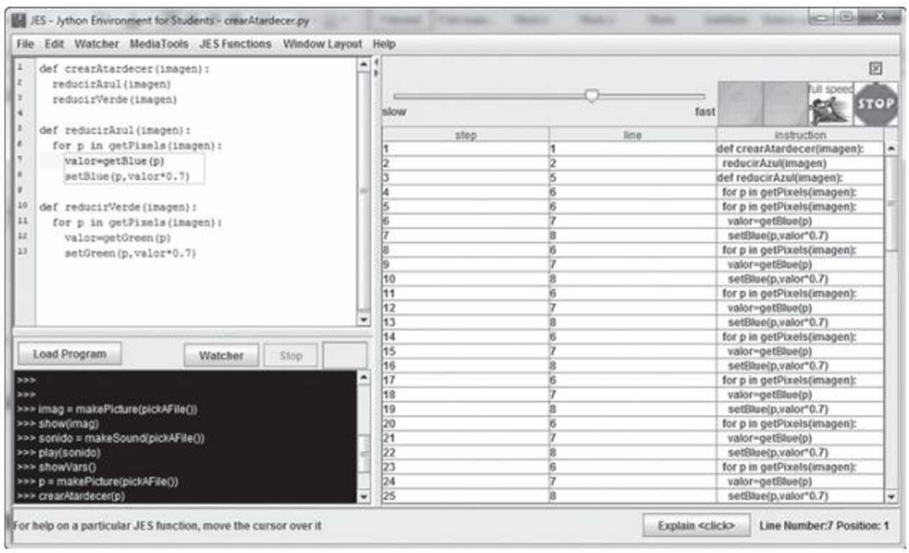



**FIGURA 9.5**
Visualización de la variable `valor` en la función `crearAtardecer()` con el Vigilante.

cuando el Vigilante se ejecute, aparecerá el valor de la variable junto con la línea. También podrá ver cuando la variable todavía no tenga un valor (figura 9.5).

### 9.4.3 Depuración del juego de aventuras

Vamos a usar algunas de estas técnicas con el juego de aventuras. El problema que tenemos con el juego de aventuras es que ¡básicamente funciona! No se generan mensajes obvios de error. Al pasar por nuestro proceso de prueba, pudimos descubrir que no estábamos manejando bien las entradas inválidas. ¿Qué más podría ser un problema?

Veamos lo que ocurre al ejecutar el programa:

```
>>> jugarJuego()
!Bienvenido a la casa de aventura¡
En cada salón le indicaremos hacia dónde
    puede ir.
Puede avanzar hacia el norte, sur, este u oeste, escribiendo
    esa dirección.
Escriba ayuda para volver a mostrar esta introducción.
Escriba terminar o salir para terminar el programa.
Usted está en el porche de una escalofriante casa.
Las ventanas están rotas. Es una noche oscura y tempestuosa.
Puede ir hacia el norte y entrar en la casa. Si se atreve.
Usted está en el recibidor de la casa. Hay telarañas
    en el rincón.
De pronto tiene una sensación de temor.
Hay un pasaje hacia el norte y otro hacia
    el este.
El porche queda detrás de usted hacia el sur.
Usted se encuentra en una sala de estar. Hay sofás,
    sillas y mesas pequeñas.
Todo está cubierto con polvo y telarañas.
```

```
De repente, escucha un ruido estremecedor en otro salón.
Puede ir hacia el norte o hacia el oeste.
Usted está en el comedor.
Hay restos de una comida en la mesa. No puede
    identificar lo que es, y tal vez no quiera hacerlo.
¿Acaso se escuchó un golpe proveniente del oeste?
Puede ir hacia el sur o hacia el oeste
Usted está en la cocina. Todas las superficies están cubiertas
    con ollas, sartenes, pedazos de comida y charcos de sangre.
Cree escuchar algo proveniente de las escaleras que van hacia
    el lado oeste del cuarto.
Se escucha que algo raspa, como si se arrastrara
    por el piso.
Puede ir hacia el sur o el este.
¡Adiós!
```

¿Qué está mal en el código anterior? He aquí dos problemas que tenemos con esa impresión:

1. Es bastante difícil distinguir un salón del otro. Las descripciones de los salones se confunden unas con otras.
2. No podemos regresar y ver qué escribimos en cada paso anterior. Si fuera un gran mapa con muchos salones, sería conveniente poder regresar y ver qué escribimos antes para ir a distintos salones.

Vamos a solucionar el primer problema. Necesitamos que aparezca algún tipo de espacio o separador entre las descripciones de cada salón. ¿En dónde colocamos esa instrucción? Sin duda, tiene que ser *dentro* del ciclo principal de nivel superior. Podría ir dentro de la función `mostrarSalon`, tal vez como la primera línea. O podría ir en la función de nivel superior, justo antes o después de mostrar el salón. Tomando en cuenta que es mejor realizar modificaciones de los detalles en las subfunciones, vamos a cambiar `mostrarSalon`.

**Programa 88: función `mostrarSalon` mejorada para el juego de aventuras**

```
def mostrarSalon(salon):
  printNow("==========")
  if salon == "Porche":
    mostrarPorche()
  if salon == "Recibidor":
    mostrarRecibidor()
  if salon == "Cocina":
    mostrarCocina()
  if salon == "SalaDeEstar":
    mostrarSE()
  if salon == "Comedor":
    mostrarComedor()
```

■

Ahora vamos a lidiar con el segundo problema. Debemos imprimir lo que escribió el jugador. Tenemos que hacer esto *después* de la llamada a `requestString`. De nuevo, este cambio podría estar en el ciclo de nivel superior o al principio de `elegirSalon`. Ambas opciones podrían funcionar. Esta vez vamos a tomar la otra opción. La función `elegirSalon` ya es de por sí bastante complicada. Vamos a responder justo después de que el jugador haga su elección.

**Programa 89: función jugarJuego mejorada para el juego de aventuras**

```
def jugarJuego():
  ubicacion = "Porche"
  mostrarIntroduccion()
  while not (ubicacion == "Salir") :
    mostrarSalon(ubicacion)
    direccion = requestString("¿Cuál dirección?")
    printNow("Usted escribió: "+direccion)
    ubicacion = elegirSalon(direccion, ubicacion)
```

■

Ahora podemos probar nuestro programa depurado.

```
>>> jugarJuego()
!Bienvenido a la casa de aventura¡
En cada salón le indicaremos hacia dónde
  puede ir.
Puede avanzar hacia el norte, sur, este u oeste, escribiendo
  esa dirección.
Escriba ayuda para volver a mostrar esta introducción.
Escriba terminar o salir para terminar el programa.
==========
Usted está en el porche de una escalofriante casa.
Las ventanas están rotas. Es una noche oscura y tempestuosa.
Puede ir hacia el norte y entrar en la casa. Si se atreve.
Usted escribió: norte
==========
Usted está en el recibidor de la casa. Hay telarañas
  en el rincón.
De pronto tiene una sensación de temor.
Hay un pasaje hacia el norte y otro hacia
  el este.
El porche queda detrás de usted hacia el sur.
Usted escribió: este
==========
Usted se encuentra en una sala de estar. Hay sofás,
  sillas y mesas pequeñas.
Todo está cubierto con polvo y telarañas.
De repente, escucha un ruido estremecedor en otro salón.
Puede ir hacia el norte o hacia el oeste.
Usted escribió: norte
==========
Usted está en el comedor.
Hay restos de una comida en la mesa. No puede
  identificar lo que es, y tal vez no quiera hacerlo.
¿Acaso se escuchó un golpe proveniente del oeste?
Puede ir hacia el sur o hacia el oeste
Usted escribió: oeste
==========
Usted está en la cocina. Todas las superficies están cubiertas
  con ollas, sartenes, pedazos de comida y charcos de sangre.
```

```
Cree escuchar algo proveniente de las escaleras que van hacia
  el lado oeste del cuarto.
Se escucha que algo raspa, como si se arrastrara
  por el piso.
Puede ir hacia el sur o el este.
Usted escribió: sur
==========
Usted está en el recibidor de la casa. Hay
  telarañas en el rincón.
De pronto tiene una sensación de temor.
Hay un pasaje hacia el norte y otro hacia
  el este.
El porche queda detrás de usted hacia el sur.
Usted escribió: salir
¡Adiós!
```

## 9.5 ALGORITMOS Y DISEÑO

Los algoritmos son descripciones generales de procesos que pueden implementarse en cualquier lenguaje de programación específico. El conocimiento de los algoritmos es una de las opciones en las cajas de opciones de los programadores profesionales. Hemos visto varios algoritmos hasta ahora:

- El *algoritmo de muestreo* es un proceso que puede usarse para desplazar la frecuencia de un sonido hacia arriba o hacia abajo, o para aumentar o reducir la escala de una imagen. No tenemos que hablar sobre los ciclos ni sobre incrementar índices de origen o de destino para describir el algoritmo de muestreo. Éste funciona modificando la forma en que copiamos muestras o píxeles de un origen a un destino; en vez de tomar cada muestra o píxel, tomamos una de cada dos muestras/pixeles o cada muestra/pixel dos veces, o usamos algún otro patrón de muestreo.
- También vimos cómo copiar píxeles o muestras de un origen a un destino. Sólo tenemos que saber en dónde nos encontramos tanto en el origen como en el destino.
- Vimos además que mezclar es en esencia lo mismo para píxeles que para muestras. Aplicamos una ponderación a cada uno de los píxeles o muestras a sumar y después sumamos los valores ponderados para crear un sonido mezclado o una imagen mezclada.

El rol de los algoritmos en el diseño es permitirnos *abstraer* una descripción del programa que vamos a diseñar, a un nivel por encima del código básico del programa. Los programadores profesionales conocen muchos algoritmos; esto les permite pensar en los problemas de diseño de los programas en un nivel más alto. Podemos hablar sobre negación de imágenes y cómo reflejar las imágenes negadas *sin tener que* hablar sobre ciclos ni índices de origen o de destino. Podemos enfocarnos en nombres más abstractos como "reflejar" sin enfocarnos en el código.

Los programadores también saben mucho *sobre* los algoritmos que conocen. Saben cómo hacer los algoritmos eficientes, cuándo no son útiles y cuáles podrían ser las dificultades de éstos. Por ejemplo, sabemos que al aplicar una escala a un sonido, debemos tener cuidado de no traspasar los límites de éste. Hay algoritmos mejores y peores, en términos de la rapidez con la que se ejecutan y qué tanta memoria requieren. En el capítulo 13 hablaremos más sobre la velocidad de los algoritmos.

En este programa tomamos decisiones que podrían haber sido diferentes. Por ejemplo, describimos los salones y la forma en que se conectan entre sí en el código del programa.

Podríamos haber guardado las descripciones como cadenas en variables. Incluso podríamos haber usado algunas otras *estructuras de datos*, como arreglos y secuencias, para describir la forma en que se conectan los salones entre sí. El programa tendría entonces una apariencia muy distinta. Hubiera procesado los *datos* en las variables más que describir los salones en sí. Al proceso de realizar elecciones entre distintas formas de escribir un programa se le conoce como *diseñar el programa*. Podemos pensar en el diseño del programa (esto es, las posibles decisiones además de sus fortalezas y debilidades) de una manera totalmente separada del código del programa en sí.

## 9.6 EJECUCIÓN DE PROGRAMAS FUERA DE JES

Los programas en Python pueden ejecutarse de muchas formas. Si crea programas más grandes y complejos, será conveniente poder ejecutarlos fuera de JES. Lo que usted aprenda en este libro podrá usarlo *directamente* en Python (o Jython). Los comandos como `for` y `print` funcionan en Python y en Jython. Las bibliotecas del sistema como `ftplib` y `urllib` son idénticas en Python y Jython.

Sin embargo, las herramientas de medios que usamos no están integradas en Python ni en Jython. Es posible usar las bibliotecas de medios en Python. Hay implementaciones de Python como Myro (`http://myro.roboteducation.org`) que incluyen las mismas funciones de imágenes que en JES. También está la *Biblioteca de imágenes de Python (PIL)* que proporciona una funcionalidad similar, con funciones que tienen nombres diferentes.

Usted puede usar en Jython las bibliotecas que proporcionamos en este libro. Jython está disponible en `http://www.jython.org` para la mayoría de los sistemas de computadora. Nuestras bibliotecas de medios funcionan en Jython con sólo unos cuantos comandos adicionales. La figura 9.6 muestra el uso de la biblioteca de medios en Jython en Linux.

He aquí cómo hacemos que las funciones de medios funcionen en Jython tradicional.

- Para buscar los módulos a importar mediante `import`, Python usa una variable llamada `sys.path` (de la biblioteca de sistema integrada `sys`) que contiene una lista de todos los directorios en donde debe buscar los módulos. Si usted quiere usar la biblioteca de medios de JES en Jython, necesita poner la ubicación de esos archivos de módulos en su variable `sys.path` (esto es lo que `setLibPath` hace por usted en JES).

  Necesita la instrucción `import sys` para tener acceso a la variable `sys.path`. Para manipular esa variable usamos el método `insert`, el cual coloca el directorio **Sources** de JES en su ruta (vea la figura 9.6). En equipos Macintosh hay que hacer referencia al código de Java y Jython dentro de la aplicación de JES, escribiendo algo como `sys.path.insert (0,"/users/guzdial/JES.app/Contents/Resources/Java")`.
- Después necesitamos la instrucción `from media import *` para que las funciones como `pickAFile` y `makePicture` estén disponibles en Jython.

  Cabe mencionar que la instrucción `from media import *` se inserta (lo cual es invisible para usted, el programador estudiante) en su área de programa cada vez que presiona el botón LOAD (Cargar). Así es como las funciones especiales de medios de JES se ponen a disposición de sus programas.

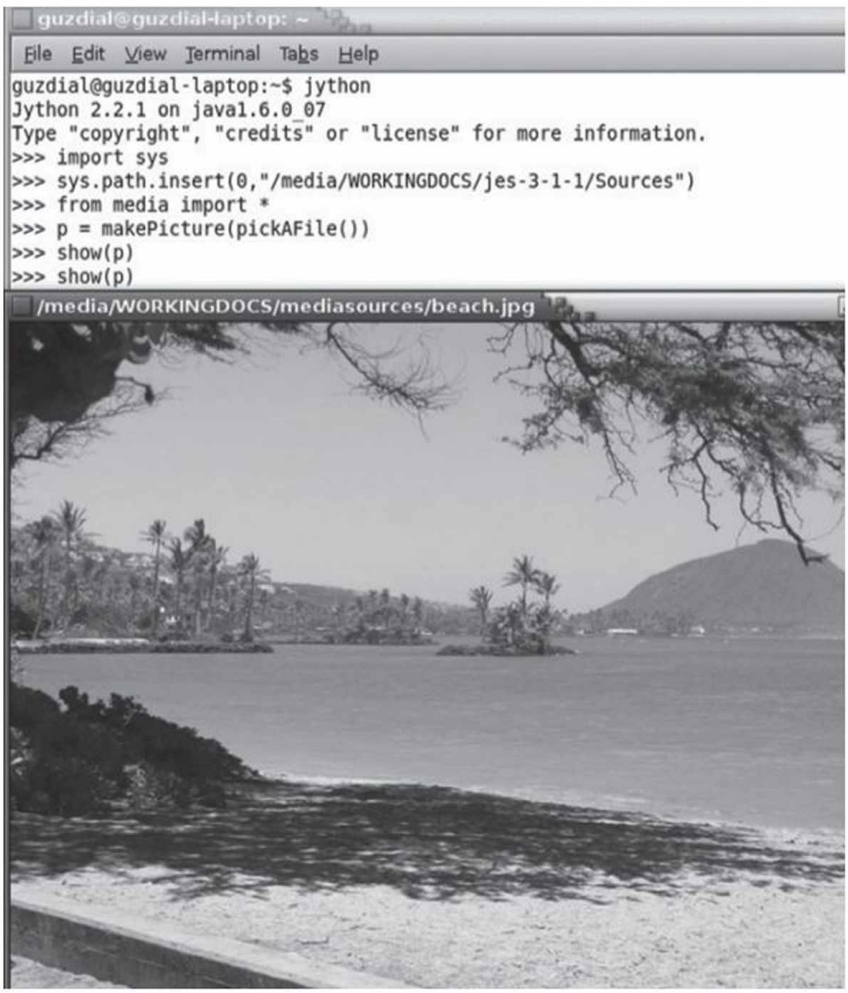



**FIGURA 9.6**
Uso de Jython para ejecutar funciones de medios fuera de JES en Linux.

A continuación le mostramos cómo se genera la imagen de la figura 9.6 desde Linux:

```
guzdial@guzdial - laptop:~\$ jython
Jython 2.2.1 on java1.6.0_07
Type "copyright", "credits" or "license" for more
  information.
>>> import sys
>>> sys.path.insert(0,"/media/MyUSB/jes-4-0/Sources")
>>> from media import *
>>> p = makePicture(pickAFile())
>>> show(p)
```

El Dr. Manuel A. Pérez-Quiñones en Virginia Tech y sus estudiantes descubrieron cómo usar las bibliotecas de medios en Mac OS X con Jython. Iniciaron Jython usando un archivo de *shell de UNIX* parecido a éste:

```
# Ejecución de jython dentro de JES (asumimos que jes-4-3.app
# está en la carpeta /Applications y que jython está dentro de la
# aplicación JES, lo cual es la configuración predeterminada)

# Establecer algunas variables
# (las siguientes dos son de mi propia creación, sin significado especial
# aparte de este uso)
# Esta es la ruta en la Mac en donde se guardan los archivos de
# java/jython JESHOME="/Applications/jes-4-3.app/
# Contents/Resources/Java"

# Estos son todos los archivos jar que usa JES, todos están
# en el directorio sobre JESJARS=$JESHOME/AVIDemo.jar:$
# JESHOME/customizer.jar:$JESHOME/jl1.0.jar:$JESHOME/
# jmf.jar:$JESHOME/junit.jar:$JESHOME/jython.jar:$JESHOME/
# mediaplayer.jar:$JESHOME/multiplayer.jar

# ejecutar jython
# algunas de las opciones extra son de la documentación de jython
# necesita configurar varias rutas: CLASSPATH para que java encuentre
# archivos jar, y jython.home para que jython encuentre algunos
# archivos de python (por ahora, uno más a continuación)
java -Xmx512m -Xss1024k -Dfile.encoding=UTF-8 -classpath
$JESJARS:$CLASSPATH -Dpython.home=$JESHOME/jython-2.2.1
-Dpython.executable=./jython org.python.util.jython
```

Después, dentro de Jython hay que escribir lo siguiente:

```
>>> import sys
>>> sys.path.insert(0, "/Applications/jes-4-3.app/
    Contents/Resources/Java/")
>>> from media import *   # importarlas
>>> #Ahora todo debe funcionar
>>> show(makePicture(pickAFile()))
```

## RESUMEN DE PROGRAMACIÓN

| | |
|---|---|
| `printNow` | Imprime la entrada de la función de inmediato mientras que el programa siga ejecutándose. En cambio, `print` no imprime la entrada hasta que el programa termine de ejecutarse. |
| `requestString` | Muestra un cuadro de diálogo con el indicador de entrada, acepta una cadena del usuario y devuelve esa cadena. También existen en JES las funciones análogas `requestNumber`, `requestInteger` e incluso `requestIntegerInRange` (que restringe el entero introducido para que esté entre dos valores de entrada). |
| `showVars` | Muestra todas las variables existentes y sus valores. |
| `while` | Itera por las instrucciones en el bloque de instrucciones que sigue después de la instrucción `while`, siempre y cuando la prueba especificada en la instrucción `while` sea verdadera. |

## PROBLEMAS

9.1 Por lo general, no se *optimiza* un programa (hacer que se ejecute más rápido o con menos uso de memoria) sino hasta después de ejecutarlo y de que esté bien depurado y probado (desde luego que tendremos que volver a probarlo después de cada modificación de optimización). He aquí una optimización que podríamos realizar en el juego de aventuras. Hasta ahora, `mostrarSalon` compara la variable `salon` con cada posible salón; incluso aunque se asocie antes. Python nos proporciona una forma de probar sólo una vez, mediante el uso de una instrucción `elif` en vez de instrucciones `if` posteriores. La instrucción `elif` significa "else if". Sólo se evalúa la instrucción `elif` si la instrucción `if` anterior es falsa. Puede tener tantas instrucciones `elif` como desee después de una instrucción `if`. Podría usarla de esta forma:

```
if (salon == "Porche"):
  mostrarPorche()
elif (salon == "Cocina"):
  mostrarCocina()
```

Vuelva a escribir el método `mostrarSalon` de una manera más óptima mediante el uso de `elif`.

9.2 Una función como `elegirSalon` es difícil de leer con todas las instrucciones `if` anidadas. Puede volverse más legible mediante el uso apropiado de comentarios para explicar lo que hace cada sección del código: comprobar el salón y después comprobar las posibles direcciones en el salón. Agregue comentarios a `elegirSalon` para facilitar su lectura.

9.3 Agregue comentarios a *todos* los métodos para que sea más fácil para alguien más leer la función.

9.4 Agregue otra variable para el jugador, llamada `mano`. Haga que la mano esté vacía en un principio. Cambie la descripción de modo que haya una "llave" en la sala de estar. Si el jugador escribe el comando `llave` mientras se encuentra en la sala de estar, la llave se colocará en su mano. Ahora, si el jugador tiene la llave mientras entra a la cocina, tendrá acceso a las escaleras y podrá ir hacia el "oeste" y subir por las escaleras. Tendrá que agregar algunos salones al juego para que esto pueda funcionar.

9.5 Cree la habilidad de `bajar` desde el Porche para explorar un mundo subterráneo secreto.

9.6 Agregue elementos secretos adicionales que el jugador pueda tener en su mano para permitir el acceso a distintos salones, como una linterna para que pueda tener acceso a un túnel debajo del porche.

9.7 Agregue una "bomba" en el comedor. Si el jugador escribe `bomba` estando en el comedor, ésta se colocará en su mano. Si el jugador escribe `bomba` al subir por las escaleras en donde se encuentra el Ogro, dejará caer la bomba y el Ogro explotará en pedazos.

9.8 Agregue la habilidad de que el jugador *pierda* en el juego (tal vez que muera). Cuando el jugador pierda, el juego deberá imprimir lo que ocurrió y después terminará. Tal vez si encuentra el ogro *sin* la bomba en su mano, el jugador pierda el juego.

9.9 Agregue la habilidad de que el jugador *gane* el juego. Cuando éste gane, el juego deberá imprimir lo que ocurrió y terminar. Tal vez al encontrar el salón del tesoro secreto bajo el Porche el jugador gane el juego.

9.10 Las descripciones de los salones no tienen que ser sólo verbales. Reproduzca un sonido relevante cuando el jugador entre a un salón específico. Use `play` de modo que pueda continuar el juego mientras se reproduce el sonido.

9.11 Las descripciones de los salones pueden ser visuales, así como textuales y auditivas. Muestre una imagen relevante al salón al momento de entrar en él. Para obtener puntos extra, muestre una imagen sólo una vez mediante `show` y, si el jugador vuelve a entrar al salón, use `repaint` para que vuelva a aparecer la imagen de nuevo.

9.12 Una posible fuente de errores en este juego de aventuras es que los nombres de los salones aparecen en varios lugares. Si Sala de estar se escribe como "SalaDeEstar" en un lugar y como "SalaEstar" (sin la palabra "De" en medio) en otro lugar, el juego no funcionará de manera correcta. Entre más salones agregue y entre más lugares haya en donde escriba los nombres de los salones, aumentarán las probabilidades de cometer un error.

Hay algunas formas de hacer que este error sea menos probable de ocurrir:

- No asigne nombres a los salones con cadenas de caracteres. Use mejor números. Es más fácil escribir y verificar "4" que "SalaDeEstar".
- Use una *variable* para `SalaDeEstar` y use esa variable para verificar la ubicación. Así no importará si usa números o cadenas (y las cadenas son más fáciles de leer y comprender).

Use una de estas técnicas para volver a escribir el juego de aventuras con menos errores potenciales.

9.13 La función `printNow` no es la única forma de presentar información al usuario durante la ejecución de un programa. También podríamos usar la función `showInformation` que recibe una cadena como entrada y luego la muestra en un cuadro de diálogo. En este momento, nuestras subfunciones de `mostrarSalon` suponen que vamos a mostrar la información del salón por medio de `printNow`. Si funciones como `mostrarPorche` *devolvieran una cadena* con la descripción, entonces la función `mostrarSalon` podría usar `printNow` para mostrar la descripción del salón, o `showInformation`.

Vuelva a escribir las funciones para mostrar los salones de modo que devuelvan una cadena; después modifique `mostrarSalon` para poder cambiar con facilidad entre imprimir la información de los salones y mostrarla en un cuadro de diálogo.

9.14 Considere el siguiente programa:

```
def pruebaMe(p,q,r):
  if q > 50:
    print r
  valor = 10
  for i in range(1,p):
    print "Hola"
    valor = valor - 1
  print valor
  print r
```

Si ejecutamos `pruebaMe(5,51,"¡Hola también a ti!")`, ¿qué imprimirá?

9.15
```
def nuevaFuncion(a, b, c):
    print a
    lista1 = range(1,5)
    valor = 0
    for x in lista1:
      print b
      valor = valor +1
    print c
    print valor
```

Si llama a la función anterior escribiendo nuevaFuncion `("Yo", "tú", "morsa")`, ¿qué imprimirá la computadora?

## PARA PROFUNDIZAR

En la actualidad los juegos de aventuras se conocen comúnmente como *ficción interactiva*. Hay sitios Web y archivos en donde podemos descargar y jugar ficción interactiva. Aún mejor, hay lenguajes de programación diseñados de forma especial para crear videojuegos, como *Inform 7*.

Quizás el mejor libro que se haya escrito sobre ingeniería de software sea *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2da edición)*, por Frederick P. Brooks (Addison-Wesley, 1995). Brooks señala que muchos de los problemas del desarrollo de software son cuestiones de organización. Recomendamos este libro ampliamente.

PARTE 3

# TEXTO, ARCHIVOS, REDES, BASES DE DATOS Y UNIMEDIA

**Capítulo 10** Creación y modificación de texto

**Capítulo 11** Técnicas avanzadas de texto: Web e información

**Capítulo 12** Creación de texto para Web

CAPÍTULO 10

# Creación y modificación de texto

10.1 TEXTO COMO UNIMEDIA
10.2 CADENAS: CREACIÓN Y MANIPULACIÓN DE CADENAS
10.3 MANIPULACIÓN DE PARTES DE CADENAS
10.4 ARCHIVOS: LUGARES EN DONDE PUEDE COLOCAR SUS CADENAS Y OTRAS COSAS
10.5 LA BIBLIOTECA ESTÁNDAR DE PYTHON

### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Generar texto en un estilo de carta modelo.
- Manipular texto estructurado, como listados telefónicos y de direcciones.
- Generar texto con estructuras aleatorias.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Acceder a los componentes de objetos mediante la notación de punto.
- Manipular cadenas.
- Leer y escribir en archivos.
- Comprender las estructuras de los archivos, como los árboles.
- Escribir programas que manipulen a otros programas, lo cual conduce a ideas poderosas como los intérpretes y compiladores.
- Usar los módulos en la Biblioteca estándar de Python, como las herramientas `random` y `os`.
- Tener una idea básica de la funcionalidad disponible en la Biblioteca estándar de Python.
- Ampliar la comprensión de la capacidad de importar elementos mediante `import`.

## 10.1 TEXTO COMO UNIMEDIA

Nicholas Negroponte, fundador de MIT Media Lab, dijo que la multimedia basada en computadora es posible gracias al hecho de que la computadora es en realidad **unimedia**, pues sólo comprende una cosa: ceros y unos. Podemos usar la computadora para multimedia debido a que es posible codificar cualquier medio como unos y ceros.

Él podría haber hablado también sobre el *texto* como unimedia. Podemos codificar cualquier medio en texto. Esto es mejor que usar ceros y unos, puesto que podemos *leer* el texto. Más adelante en este libro asociaremos sonidos al texto y después lo volveremos a asociar a los sonidos, y haremos lo mismo con las imágenes. Pero una vez que tengamos nuestros

medios en texto, no habrá necesidad de regresar al medio original: podemos asociar sonidos al texto y después a imágenes, con lo cual creamos *visualizaciones* de los sonidos.

La mayor parte de World Wide Web consta de texto. Visite cualquier página Web, después vaya al menú de su navegador Web y seleccione "VER EL CÓDIGO FUENTE". Lo que aparecerá a continuación es texto. Toda página Web consiste de texto. Este texto hace referencia a las imágenes, sonidos y animaciones que aparecen al ver la página, pero la página en sí está definida como texto. Las palabras en el texto son una notación conocida como *Lenguaje de marcación de hipertexto (HTML).*

Hasta ahora hemos podido arreglárnoslas con unas cuantas ideas de lenguajes de programación. JES se diseñó de modo que podamos realizar todo nuestro trabajo con sonidos e imágenes usando sólo instrucciones de asignación, `for`, `if`, `print`, `return` y funciones. Pero los lenguajes de programación tienen todavía más características y capacidades que éstas. En este capítulo empezaremos a mostrarle lo que hay dentro de JES para que pueda obtener una mayor capacidad en su programación.

## 10.2 CADENAS: CREACIÓN Y MANIPULACIÓN DE CADENAS

Por lo general el texto se manipula como **cadenas**. Una cadena es una secuencia de caracteres. Las cadenas se almacenan en memoria como un arreglo, justo igual que los sonidos. Las cadenas son una secuencia contigua de los buzones de correo de la memoria: los buzones que están uno al lado de otro. La cadena "Hola" se almacenaría en cuatro buzones, uno después del otro: un buzón contendría el código binario que representa a la "H", el siguiente contendría la "o", el otro contendría la "l", etcétera.

Las cadenas se definen con secuencias de caracteres encerrados entre comillas. Python es inusual en cuanto a que permite varios tipos distintos de comillas. Podemos usar comillas sencillas, dobles o incluso triples. Podemos *anidar* las comillas. Si empezamos una cadena con comillas dobles, entonces podemos usar comillas sencillas dentro de la cadena, ya que ésta no terminará sino hasta el siguiente conjunto de comillas dobles. Si empezamos a escribir una cadena con comillas sencillas, podemos colocar todas las comillas dobles que queramos dentro de la cadena, ya que Python espera la siguiente comilla sencilla para terminar.

```
>>> print 'Esta es una cadena entre comillas sencillas'
Esta es una cadena comillas sencillas
>>> print "Esta es una cadena entre comillas dobles"
Esta es una cadena entre comillas dobles
>>> print """Esta es una cadena entre comillas triples"""
Esta es una cadena entre comillas triples
```

¿Por qué las comillas triples? Porque nos permite incrustar nuevas líneas, espacios y tabuladores en nuestras cadenas. No podemos usarlas con facilidad desde el área de comandos, pero podemos hacerlo en el área del programa.

```
def cadenaLoca():
  print """Así se usan las comillas triples. ¿Por qué?
Observe las distintas líneas.

Porque podemos hacer esto."""
```

```
>>> cadenaLoca()
Así se usan las comillas triples. ¿Por qué?
Observe las distintas líneas.


Porque podemos hacer esto.
```

El valor de tener tantos tipos distintos de comillas es para facilitar la acción de colocar comillas *dentro* de las cadenas. Por ejemplo, HTML usa comillas dobles como parte de su notación. Si desea escribir una función de Python que cree páginas de HTML (un uso común para Python), entonces necesitará cadenas que contengan comillas. Como se puede usar cualquiera de estas comillas, podemos incrustar comillas dobles si usamos comillas sencillas para iniciar y terminar la cadena.

```
>>> print " " "
Invalid syntax
Your code contains at least one syntax error, meaning
it is not legal jython.
>>> print ' " '
"
```

Podemos considerar una cadena como un arreglo o secuencia de caracteres. En realidad es una secuencia: podemos usar una instrucción `for` para recorrer todos los caracteres.

```
>>> for i in "Hola":
...     print i
...
H
o
l
a
```

En memoria, una cadena es una serie de buzones consecutivos (para continuar con nuestra metáfora de memoria como un departamento de correspondencia), cada uno de los cuales contiene el código binario para el carácter correspondiente. La función `ord()` nos da la codificación ASCII (Código Estándar Estadounidense para el Intercambio de Información) para cada carácter. Así, la cadena "Hola" tiene cuatro buzones, en donde el primero contiene 72, el que sigue 111, después 108 y así, en lo sucesivo.

```
>>> cad = "Hola"
>>> for car in cad:
...       print ord(car)
...
72
111
108
97
```

En JES, ésta es una ligera simplificación. La versión de Python que usamos (Jython) está basada en Java, y en realidad *no* usa ASCII para codificar sus cadenas sino Unicode, una codificación para caracteres en donde se utilizan dos bytes por cada carácter. Dos bytes nos proporcionan 65 536 posibles combinaciones. Todos esos posibles códigos adicionales nos permi-

ten ir más allá de un simple alfabeto en latín, además de los signos de puntuación y números. Podemos representar Hiragana, Katakana y otros sistemas de *glifos*.

Lo que esto nos indica es que hay muchos más caracteres posibles de los que pueden escribirse en el teclado. No sólo hay símbolos especiales, sino también caracteres invisibles como los tabuladores y la pulsación de la tecla Intro. Para escribir estos caracteres en Python (y en muchos otros lenguajes, como Java y C) usamos *escapes de barra diagonal inversa*. Estos escapes consisten en la tecla de barra diagonal inversa \ seguida de un carácter.

- `\t` es lo mismo que presionar la tecla de tabulación.
- `\b` es lo mismo que presionar la tecla de retroceso (que no es un carácter muy útil de poner en una cadena, pero igual se puede hacer). Al imprimir `\b`, aparece como un cuadro en la mayoría de los sistemas; en realidad no puede imprimirse (ver a continuación).
- `\n` es lo mismo que presionar la tecla Intro.
- `\uXXXX` en donde XXXX es un código compuesto por dígitos del 0 al 9 y de la A a la F (conocido como número *hexadecimal*), representa el carácter Unicode con ese código. Puede buscar los códigos en `http://www.unicode.org/charts`.

No es fácil ver glifos de Unicode en JES,[1] pero podemos ver cómo afectan algunos de estos símbolos especiales en la impresión.

```
>>> print "hola\ta todos.\nMark"
hola    a todos.
Mark
```

Recuerde que anteriormente en este libro usamos una 'r' al principio de una cadena de nombre de archivo, por ejemplo:

```
r"C:\ip-book\mediasources\barbara.jpg"
```

Una 'r' indica a Python que debe leer la cadena en *modo puro*. Se ignoran todos los escapes de barra diagonal inversa. Esto es importante en rutas de Windows, ya que este sistema operativo usa las barras diagonales inversas como *delimitadores*. Por ejemplo, si su nombre de archivo empieza con 'b', Python vería esa `\b` como un carácter de retroceso y no como un delimitador -b.

Es fácil sumar cadenas mediante + (lo que se conoce también como **concatenación** de cadenas) y obtener las longitudes de las cadenas mediante la función `len()`.

```
>>> hola = "Hola"
>>> print len(hola)
4
>>> mark = ", Mark"
>>> print len(mark)
6
>>> print hola+mark
Hola, Mark
>>> print len(hola+mark)
10
```

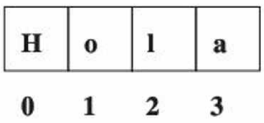
[1]Puede verlos en CPython, con un comando como `print u"\ufeed"`.

## 10.3 MANIPULACIÓN DE PARTES DE CADENAS

Usamos la notación de corchetes ([]) para hacer referencia a partes de las cadenas.

- `cadena[n]` devuelve el *n*-ésimo carácter en la cadena, en donde el primer carácter en la cadena es cero.
- `cadena[n:m]` devuelve una *porción* de la cadena que empieza en el *n*-ésimo carácter y llega hasta, *pero sin incluir al* *m*-ésimo carácter [similar a la forma en que trabaja la función `range()`]. De manera opcional puede omitir a *n* o a *m*. Si omite a *n*, se considera cero (el inicio de la cadena). Si omite a *m*, se considera como el final de la cadena. Podemos también usar números negativos en cualquier extremo para recortar ciertos caracteres de ese lado.

Podemos considerar que los caracteres de la cadena se encuentran en cajas, cada uno con su propio número de índice.




```
>>> hola = "Hola"
>>> print hola[1]
o
>>> print hola[0]
H
>>> print hola[2:4]
la
>>> print hola
Hola
>>> print hola[:2]
Ho
>>> print hola[2:]
la
>>> print hola[:]
Hola
>>> print hola[-1:]
A
>>> print hola[:-1]
Hol
```

### 10.3.1 Métodos de cadenas: introducción a los objetos y la notación punto

En realidad, todo en Python es más que un valor: es un *objeto*. Un objeto combina datos (como un número, una cadena o una lista) con los *métodos* que pueden actuar sobre ese objeto. Los métodos son como funciones, excepto que no son accesibles en forma global. No podemos ejecutar un método de la forma en que podemos ejecutar `pickAFile()` o `makeSound()`. Un método es una función a la que sólo se puede acceder *a través* de un objeto.

En Python, las cadenas son objetos. No son sólo secuencias de caracteres; también tienen métodos que no son accesibles a nivel global, sino que sólo las cadenas los conocen. Para ejecutar un método de una cadena, usamos la *notación punto*. Escribimos `objeto.método()`.

Un método de ejemplo que sólo las cadenas conocen es `capitalize()`. Este método pone en mayúscula la primera letra de la primera palabra de la cadena que lo invocó. No trabaja sobre una función ni sobre un número.

```
>>> prueba="esta es una prueba."
>>> print prueba.capitalize()
Esta es una prueba.
>>> print capitalize(prueba)
The error was:capitalize
Name not found globally.
A local or global name could not be found. You need
to define the function or variable before you try to
use it in any way.
>>> print 'esta es otra prueba'.capitalize()
Esta es otra prueba
>>> print 12.capitalize()
Invalid syntax
Your code contains at least one syntax error, meaning
it is not legal jython.
```

Hay *muchos* métodos útiles para las cadenas.

- `startswith(prefijo)` devuelve `true` si la cadena empieza con el prefijo indicado. Recuerde que `true` en Python puede ser desde 1 hasta cualquier número mayor, y `false` es cero.

```
>>> carta = "El Sr. Mark Guzdial solicita su agradable presencia..."
>>> print carta.startswith("El Sr.")
1
>>> print carta.startswith("La Sra.")
0
```

- `endswith(sufijo)` devuelve `true` si la cadena termina con el sufijo indicado. Esta función es en especial útil para revisar si un nombre de archivo es del tipo correcto para un programa.

```
>>> nombrearchivo="barbara.jpg"
>>> if nombrearchivo.endswith(".jpg"):
...         print "Es una imagen"
...
Es una imagen
```

- `find(cad)`, `find(cad,inicio)` y `find(cad,inicio,fin)` encuentran la `cad` en el objeto cadena y devuelven el número de índice en donde empieza. En las formas opcionales, puede indicarle desde qué número de índice empezar e incluso en dónde dejar de buscar.

Esto es muy importante: el método `find()` devuelve –1 si fracasa. ¿Por qué –1? Porque cualquier valor que sea de 0 a 1 menor que la longitud de la cadena *podría* ser un índice válido en donde podría encontrarse una cadena de búsqueda.

```
>>> print carta
El Sr. Mark Guzdial solicita su agradable presencia...
>>> print carta.find("Mark")
7
>>> print carta.find("Guzdial")
12
>>> print len("Guzdial")
7
>>> print carta[7:12+7]
Mark Guzdial
>>> print carta.find("fred")
-1
```

También existe `rfind(buscarcadena)` (y las mismas variaciones con parámetros opcionales), que busca desde el final de la cadena hacia el frente.

- `upper()` cambia la cadena a mayúsculas.
- `lower()` cambia la cadena a minúsculas.
- `swapcase()` convierte todas las mayúsculas en minúsculas y viceversa.
- `title()` convierte sólo los primeros caracteres en mayúsculas y el resto en minúsculas.

Estos métodos pueden colocarse *en cascada*: uno modifica el resultado de otro.

```
>>> cadena="Esta es una prueba de Algo."
>>> print cadena.swapcase()
eSTA ES UNA PRUEBA DE aLGO.
>>> print cadena.title().swapcase()
eSTA eS uNA pRUEBA dE aLGO.
```

- `isalpha()` devuelve `true` si la cadena no está vacía y contiene sólo letras: sin números ni signos de puntuación.
- `isdigit()` devuelve `true` si la cadena no está vacía y contiene sólo números. Puede usar esta función para verificar los resultados de una búsqueda. Suponga que necesita escribir un programa para buscar los precios de algunas acciones. Desea analizar un *precio* actual, no el nombre de una acción. Si no lo recibe en forma correcta, tal vez su programa podría comprar o vender acciones que usted no desee. Podría usar `isdigit()` para verificar su resultado de manera automática.
- `replace(búsqueda,reemplazo)` busca la cadena llamada `búsqueda` y la reemplaza con la cadena `reemplazo`. Devuelve el resultado pero no cambia la cadena original.

```
>>> print carta
El Sr. Mark Guzdial solicita su agradable presencia...
>>> carta.replace("a","!")
'El Sr. M!rk Guzdi!l solicit! su !gr!d!ble presenci!...'
>>> print carta
El Sr. Mark Guzdial solicita su agradable presencia...
```

### 10.3.2 Listas: texto poderoso y estructurado

Las listas son estructuras muy poderosas que podemos considerar como un tipo de *texto estructurado*. Las listas se definen con corchetes y comas entre sus elementos; pueden contener casi cualquier cosa, incluso sublistas. Al igual que las cadenas, podemos hacer referencia a las partes con notaciones de corchetes y podemos sumarlas con el símbolo +. Las listas son también secuencias, por lo que podemos usar un ciclo `for` con ellas para recorrer sus piezas.

```
>>> miLista = ["Este","es","un",12]
>>> print miLista
['Este', 'es', 'un', 12]
>>> print miLista[0]
Este
>>> for i in miLista:
...       print i
...
Este
es
un
12
>>> print miLista + ["De verdad"]
['Este', 'es', 'un', 12, 'De verdad']
>>> otraLista=["esta","tiene",["una",["sub","lista"]]]
>>> print otraLista
['esta', 'tiene', ['una', ['sub', 'lista']]]
>>> print otraLista[0]
esta
>>> print otraLista[2]
['una', ['sub', 'lista']]
>>> print otraLista[2][1]
['sub', 'lista']
>>> print otraLista[2][1][0]
sub
>>> print otraLista[2][1][0][2]
b
```

Las listas pueden usar el siguiente conjunto de métodos; las cadenas no pueden usarlos.

- `append(algo)` coloca `algo` al final de la lista.
- `remove(algo)` elimina `algo` de la lista si está ahí.
- `sort()` coloca la lista en orden alfabético.
- `reverse()` invierte la lista.
- `count(algo)` nos indica el número de veces que está `algo` en la lista.
- `max()` y `min()` son funciones que hemos visto antes; reciben una lista como entrada y nos proporcionan los valores máximo y mínimo en la lista.

```
>>> lista = ["oso","manzana","gato","elefante","perro","manzana"]
>>> lista.sort()
>>> print lista
['elefante', 'gato', 'manzana', 'manzana', 'oso', 'perro']
>>> lista.reverse()
```

```
>>> print lista
['perro', 'oso', 'manzana', 'manzana', 'gato', 'elefante']
>>> print lista.count('manzana')
2
```

Uno de los métodos más importantes para las cadenas, `split(delimitador)`, convierte una cadena en una lista de subcadenas, separándola con base en una cadena delimitadora que usted le proporcione. Esto nos permite convertir las cadenas en listas.

```
>>> print carta.split(" ")
['El', 'Sr.', 'Mark', 'Guzdial', 'solicita', 'su',
'agradable', 'presencia...']
```

Mediante el uso de `split()` podemos procesar **texto con formato**: texto en donde la separación entre las partes es un carácter bien definido, como el **texto delimitado por tabuladores** o el **texto delimitado por comas** de una hoja de cálculo. He aquí un ejemplo del uso de texto estructurado para guardar una agenda telefónica. Esta agenda tiene líneas separadas por nuevas líneas y partes separadas por signos de dos puntos. Podemos dividir con base en las nuevas líneas y después tomar en cuenta los signos de dos puntos para obtener una lista de sublistas. Las búsquedas por esta lista se pueden realizar con un ciclo `for` simple.

**Programa 90: una aplicación simple de agenda telefónica**

```
def agendatelefonica():
  return """
Mary:893-0234:Agente inmobiliario:
Pedro:897-2033:Operador de maquinaria:
Pablo:234-2342:Boleador profesional:"""

def telefonos():
  telefonos = agendatelefonica()
  listatelefonos = telefonos.split('\n')
  nuevalistatelefonos = []
  for lista in listatelefonos:
    nuevalistatelefonos = nuevalistatelefonos + [lista.split(":")]
  return nuevalistatelefonos

def buscarTelefono(persona):
  for gente in telefonos():
    if gente[0] == persona:
      print "El número telefónico de",persona,"es",gente[1]
```

■

## Cómo funciona

Hay tres funciones en este programa: una para proporcionar el texto de los teléfonos, la otra para crear la lista de teléfonos y la tercera para buscar un número telefónico.

- La primera función se llama `agendatelefonica`; es la que crea el texto estructurado y lo devuelve, usando comillas triples para poder dar formato a las líneas con caracteres de nueva línea. El formato es: nombre, dos puntos, número telefónico, dos puntos y empleo, luego dos puntos y el fin de la línea.

```
>>> print agendatelefonica()
Mary:893-0234:Agente inmobiliario:
Pedro:897-2033:Operador de maquinaria:
Pablo:234-2342:Boleador profesional:
```

- La segunda función se llama `telefonos`; devuelve una lista de todos los teléfonos. Accede a la lista telefónica mediante `agendatelefonica` y luego la divide en varias líneas. El resultado de `split` es una lista que consiste en una cadena con signos de dos puntos. A continuación, el ciclo en `telefonos` separa cada lista usando `split` con base en los signos de dos puntos. Lo que la función `telefonos` devuelve es una lista de listas.

```
>>> print telefonos()
[[''], ['Mary', '893-0234', 'Agente inmobiliario', ''],
['Pedro', '897-2033', 'Operador de maquinaria', ''],
['Pablo', '234-2342', 'Boleador profesional', '']]
```

Por último, la tercera función se llama `buscarTelefono`; recibe un nombre como entrada y busca el número telefónico correspondiente. Itera sobre todas las sublistas que devuelve `telefonos` y busca el teléfono en donde el primer elemento en la sublista (número de índice 0) sea el nombre introducido. Después imprime el resultado

```
>>> buscarTelefono('Pedro')
El número telefónico de Pedro es 897-2033
```

### 10.3.3 Las cadenas no tienen fuente

Las cadenas no tienen una **fuente** (apariencia característica de las letras) ni un **estilo** (por lo general negrita, cursiva, subrayado y otros efectos que se aplican a la cadena) asociados. La información de la fuente y el estilo se agregan a las cadenas mediante el uso de procesadores de palabras y otros programas. Por lo general, se codifican como **tiradas de estilo** (**style runs**).

Una tirada de estilo es una representación separada de la información de la fuente y el estilo, con índices en la cadena para mostrar en dónde deben realizarse los cambios. Por ejemplo, `El viejo zorro café corre` podría codificarse como [[negrita 0 7][cursiva 15 18]].

Piense en las cadenas con tiradas de estilo. ¿Cómo llamamos a esta combinación de información relacionada? Es evidente que no es un solo valor. ¿Podríamos codificar la cadena con las tiradas de estilo en una lista compleja? Claro; ¡podemos hacer casi *cualquier cosa* con las listas!

La mayor parte del software que maneja texto con formato codifica las cadenas con tiradas de estilo como un **objeto**. Los objetos tienen datos asociados, tal vez en varias partes (como las cadenas y las tiradas de estilo). Los objetos saben cómo actuar con sus datos, usando *métodos* que sólo los objetos de ese tipo conocen. Si varios objetos conocen el mismo nombre del método, es probable que haga lo mismo pero tal vez no de la misma forma.

Esto es sólo un indicio. Hablaremos con más detalle sobre los objetos en secciones posteriores.

## 10.4 ARCHIVOS: LUGARES EN DONDE PUEDE COLOCAR SUS CADENAS Y OTRAS COSAS

Los archivos son extensas colecciones de bytes en el disco duro, las cuales tienen un nombre asignado. Por lo general los archivos tienen un **nombre base** y un **sufijo de archivo**. El

archivo **barbara.jpg** tiene el nombre base "barbara" y un sufijo de archivo "jpg", el cual nos indica que el archivo es una imagen JPEG.

Los archivos se agrupan en **directorios** (también conocidos como **carpetas**). Los directorios pueden contener otros directorios además de archivos. Hay un directorio base en la computadora, el cual se conoce como **directorio raíz**. En una computadora que utiliza el sistema operativo Windows, el directorio base es similar a **C:**. A la descripción completa de los directorios que debemos visitar para llegar a un archivo específico partiendo desde el directorio base se le conoce como **ruta**.

```
>>> archivo=pickAFile()
>>> print archivo
C:\ip-book\mediasources\640x480.jpg
```

La ruta que se imprime nos indica cómo ir desde el directorio raíz hasta el archivo **640x480.jpg**. Empezamos en **C:**, seleccionamos el directorio **ip-book** y después el directorio **mediasources**.

A esta estructura la denominamos **árbol** (figura 10.1). A **C:** le llamamos la **raíz** del árbol. Este árbol tiene **ramas** en donde están los subdirectorios. Cualquier directorio puede contener más directorios (ramas) o archivos, los cuales se conocen como **hojas**. Exceptuando la raíz, cada **nodo** del árbol (rama u hoja) tiene un solo nodo rama **padre**, aunque un padre puede tener varios nodos ramas y hojas **hijos**.

Necesitamos saber acerca de los directorios y archivos si vamos a manipular archivos, en especial muchos de ellos. Si lidiamos con un sitio Web grande, vamos a estar trabajando con muchos archivos. Si vamos a usar video, tendremos cerca de 30 archivos (cuadros individuales) por cada segundo de video. En realidad no es conveniente escribir una línea de código para abrir cada cuadro de video. Es mejor escribir programas que recorran estructuras de directorios para procesar archivos Web o de video.

También podemos representar los árboles en listas. Como las listas pueden contener sublistas, así como los directorios pueden contener subdirectorios, la codificación es bastante fácil de realizar. El punto importante es que las listas nos permiten representar relaciones jerárquicas complejas, como los árboles (figura 10.2).

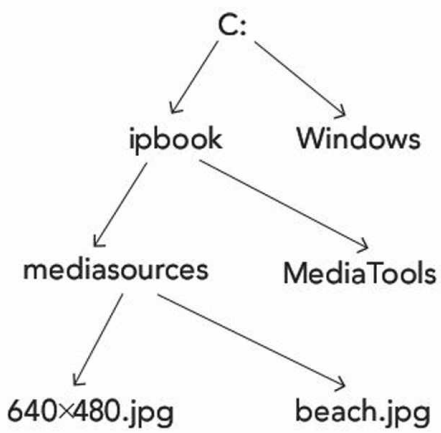



**FIGURA 10.1**
Diagrama de un árbol de directorios.

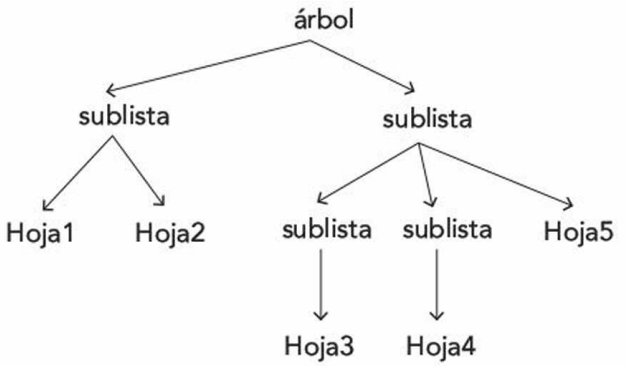



**FIGURA 10.2**
Diagrama para una lista.

```
>>> arbol = [["Hoja1","Hoja2"],[["Hoja3"],["Hoja4"],"Hoja5"]]
>>> print arbol
[['Hoja1', 'Hoja2'], [['Hoja3'], ['Hoja4'], 'Hoja5']]
>>> print arbol[0]
['Hoja1', 'Hoja2']
>>> print arbol[1]
[['Hoja3'], ['Hoja4'], 'Hoja5']
>>> print arbol[1][0]
['Hoja3']
>>> print arbol[1][1]
['Hoja4']
>>> print arbol[1][2]
Hoja5
```

### 10.4.1 Apertura y manipulación de archivos

Abrimos archivos para leer o escribir en ellos. Para ello usamos una función llamada `open(nombrearchivo, como)`. El nombre de archivo puede ser una ruta completa o sólo un nombre de archivo base y un sufijo. Si usted no proporciona una ruta, el archivo se abrirá en el directorio actual de JES.

La entrada `como` es una cadena que describe lo que deseamos hacer con el archivo.

- "rt" significa "leer el archivo como texto; traducir los bytes en caracteres para mí".
- "wt" significa "escribir en el archivo como texto".
- "rb" y "wb" significa "leer y escribir bytes" (respectivamente). Se utilizan estas opciones si vamos a manipular **archivos binarios** (como JPEG, WAV, Word o Excel).

La función `open()` devuelve un objeto archivo que podemos usar para manipular el archivo. El objeto archivo puede usar un conjunto de métodos.

- `archivo.read()` lee todo el archivo como una cadena gigante (si abrió el archivo para escribir, no trate de leer de él).
- `archivo.readlines()` lee todo el archivo y lo coloca en una lista, en donde cada elemento es una sola línea. Sólo podemos usar `read()` o `readlines()` una vez por cada apertura de archivo.

- `archivo.write(unacadena)` escribe `unacadena` en el archivo (si abrió el archivo para lectura, no trate de escribir en él).
- `archivo.close()` cierra el archivo. Si escribe datos en el archivo, al cerrarlo se asegurará de que todos los datos se escriban en el disco. Si lee datos del archivo, al cerrarlo se liberará la memoria que se usó para manipular el archivo. En cualquier caso es buena idea cerrar sus archivos cuando termine de usarlos. Una vez que cerramos un archivo, no es posible leer o escribir en él sino hasta que volvemos a abrirlo.

A continuación veremos ejemplos de cómo abrir un archivo de programa que escribimos antes y leerlo como una cadena, y también como una lista de cadenas.

```
>>> programa=pickAFile()
>>> print programa
C:\ip-book\programas\imagenChica.py
>>> archivo=open(programa,"rt")
>>> contenido=archivo.read()
>>> print contenido
def imagenchica():
  lienzo=makePicture(getMediaPath("640x480.jpg"))
  addText(lienzo,10,50,"Esto no es una imagen")
  addLine(lienzo,10,20,300,50)
  addRectFilled(lienzo,0,200,300,500,yellow)
  addRect(lienzo,10,210,290,490)
  return lienzo
>>> archivo.close()
>>> archivo=open(programa,"rt")
>>> lineas=archivo.readlines()
>>> print lineas
['def imagenchica():\n', '
lienzo=makePicture(getMediaPath("640x480.jpg"))\n', '
addText(lienzo,10,50,"Esto no es una imagen")\n', '
addLine(lienzo,10,20,300,50)\n', '
addRectFilled(lienzo,0,200,300,500,yellow)\n', '
addRect(lienzo,10,210,290,490)\n', '  return lienzo']
>>> file.close()
```

He aquí un ejemplo de cómo escribir un archivo gracioso. El carácter `\n` crea las nuevas líneas en el archivo.

```
>>> escribirArchivo = open("miarchivo.txt","wt")
>>> escribirArchivo.write("Esto es algo de texto.")
>>> escribirArchivo.write("Esto es un poco más.\n")
>>> escribirArchivo.write("Y ahora terminamos.\n\nFIN.")

>>> escribirArchivo.close()
>>> escribirArchivo=open("miarchivo.txt","rt")
>>> print escribirArchivo.read()
Esto es algo de texto.Esto es un poco más.
Y ahora terminamos.

FIN.
>>> escribirArchivo.close()
```

### 10.4.2 Generación de cartas modelo

No sólo podemos escribir programas para desglosar el texto estructurado, sino que también podemos escribir programas para *ensamblar* texto estructurado. Uno de los textos estructurados clásicos que todos conocemos son las cartas modelo, o "spam". Los verdaderos buenos escritores de spam (si acaso no es una contradicción de términos) llenan los detalles que de verdad se dirigen a *usted* en el mensaje. ¿Y cómo hacen esto? Es bastante sencillo: tienen una función que toma la entrada relevante y la inserta en los lugares correctos.

**Programa 91: un generador de cartas modelo**

```
def cartaModelo(sexo,apellido,ciudad,colorOjos):
  archivo = open("cartaModelo.txt","wt")
  archivo.write("Querid")
  if sexo=="F":
    archivo.write("a Sra. "+apellido+":\n")
  if sexo=="M":
    archivo.write("o Sr. "+apellido+":\n")
  archivo.write("Le escribo para recordarle la oferta ")
  archivo.write("que le enviamos la semana pasada. Todos en ")
  archivo.write(ciudad+" saben lo excepcional que es esta oferta ")
  archivo.write("(en especial los que tienen encantadores ojos color
  "+colorOjos+"). ")
  archivo.write("Esperamos escuchar noticias suyas pronto.\n")
  archivo.write("Sinceramente,\n")
  archivo.write("Lic. Tramposo,\nAbogado")
  archivo.close()
```

■

#### Cómo funciona

Esta función recibe el sexo, apellido (nombre de familia), ciudad y color de ojos como entrada. Abre un archivo llamado **cartaModelo.txt** y después escribe una apertura personalizada con el sexo del destinatario. Escribe varias líneas de texto, insertando la entrada en los lugares correctos. Después cierra el archivo.

Al ejecutar este programa con `cartaModelo("M","Guzdial","Decatur","café")`, genera lo siguiente:

```
Querido Sr. Guzdial:
Le escribo para recordarle la oferta que le enviamos la
semana pasada. Todos en Decatur saben lo excepcional que
es esta oferta (en especial los que tienen encantadores
ojos color café). Esperamos escuchar noticias suyas pronto.
Sinceramente,
Lic. Tramposo,
Abogado
```

### 10.4.3 Escritura de programas

Ahora vamos a empezar a *usar* archivos. Nuestro primer programa hará algo interesante: escribiremos un programa para *modificar* otro programa. Leeremos el archivo **imagenChica.py**

y modificaremos la cadena de texto que se inserta en la imagen. Usaremos `find()` para buscar la función `addText()` y luego buscaremos cada una de las comillas dobles. Después escribiremos un nuevo archivo con todo lo que contiene **imagenChica.py** hasta la primera comilla doble, insertaremos nuestra nueva cadena y colocaremos el resto del archivo desde la segunda comilla doble hasta el final.

**Programa 92: un programa para modificar el programa `imagenChica`**

```
def modificarChica(nombrearchivo,nuevaCadena):
  # Obtener el contenido del archivo original
  archivoPrograma="imagenChica.py"
  archivo = open(archivoPrograma,"rt")
  contenido = archivo.read()
  archivo.close()
  # Ahora, encontrar el lugar correcto para colocar nuestra nueva cadena
  posAgregar = contenido.find("addText")
  #Comilla doble después de addText
  primeraComilla = contenido.find('"',posAgregar)
  #Comilla doble después de primeraComilla
  ultimaComilla = contenido.find('"',primeraComilla+1)
  # Crear nuestro nuevo archivo
  nuevoArchivo = open(nombrearchivo,"wt")
  nuevoArchivo.write(contenido[:primeraComilla+1])  # incluye la comilla
  nuevoArchivo.write(nuevaCadena)
  nuevoArchivo.write(contenido[ultimaComilla:])
  nuevoArchivo.close()
```

■

## Cómo funciona

Este programa abre el archivo **imagenChica.py** (el nombre asume que está en el directorio JES, ya que no se proporciona ninguna ruta). Lee todo como una gran cadena y después cierra el archivo. Mediante el uso del método `find` encuentra la posición de `addText`, de la primera comilla doble y de la última comilla doble. Después abre un nuevo archivo (para escribir texto: "wt") y escribe todo lo del programa **imagenChica.py** hasta la primera comilla doble. Luego escribe la cadena de entrada y a continuación escribe el resto del programa, partiendo de la última comilla doble. De esta forma, reemplaza la cadena que se agrega. Por último, cierra el nuevo archivo.

Al ejecutar este programa con `modificarChica("muestra.py","Esta es una muestra de cómo modificar un programa")`, obtenemos **muestra.py**:

```
def imagenchica():
  lienzo=makePicture(getMediaPath("640x480.jpg"))
  addText(lienzo,10,50,"Esta es una muestra de cómo modificar un programa")
  addLine(lienzo,10,20,300,50)
  addRectFilled(lienzo,0,200,300,500,yellow)
  addRect(lienzo,10,210,290,490)
  return lienzo
```

Así es como funcionan los programas de dibujos basados en vectores. Al modificar una línea en AutoCAD, Flash o Illustrator, en realidad cambiamos la representación subyacente de

la imagen; en un sentido real, un pequeño programa cuya ejecución produce la imagen con la que trabajamos. Al modificar la línea, en realidad estamos modificando el programa, que después se vuelve a ejecutar para mostrarnos la imagen actualizada. ¿Es lento este proceso? Las computadoras son tan rápidas que ni siquiera lo notamos.

Es muy importante poder manipular texto para recopilar datos en Internet. La mayoría de Internet consiste tan sólo en texto. Vaya a su página Web favorita y use la opción VIEW SOURCE (VER CÓDIGO FUENTE) (o algo parecido) en el menú. Ése es el texto que define la página que vemos en el navegador. Más adelante aprenderemos a descargar páginas directamente desde Internet, pero por ahora supongamos que tenemos páginas o archivos de Internet guardados (*descargados*) en el disco duro y realizaremos búsquedas desde ahí.

Por ejemplo, hay lugares en Internet de donde podemos obtener secuencias de nucleótidos asociados con cosas como parásitos. Yo encontré un archivo de este tipo que tiene la siguiente apariencia:

```
>Schisto unique AA825099
gcttagatgtcagattgagcacgatgatcgattgaccgtgagatcgacga
gatgcgcagatcgagatctgcatacagatgatgaccatagtgtacg
>Schisto unique mancons0736
ttctcgctcacactagaagcaagacaatttacactattattattattatt
accattattattattattattactattattattattattactattattta
ctacgtcgctttttcactccctttattctcaaattgtgtatccttccttt
```

Suponga que tenemos una subsecuencia (como "ttgtgta") y queremos saber de qué parásito forma parte. Si leemos este archivo y lo colocamos en una cadena, podríamos buscar la subsecuencia. Si está ahí (es decir, si el resultado de `find` no es igual <> a –1), *retrocedemos* desde ahí para encontrar el ">" que comienza el nombre de cada parásito y luego *avanzamos* hacia el fin de línea (carácter de nueva línea) para obtener el nombre del archivo. Si no encontramos la subsecuencia (si `find()` devuelve un –1), entonces no se encuentra ahí.

**Programa 93: buscar una subsecuencia en secuencias de nucleótidos de parásitos**

```
def buscarSecuencia(sec):
  archivoSecuencias = getMediaPath("parasitos.txt")
  archivo = open(archivoSecuencias,"rt")
  secuencias = archivo.read()
  archivo.close()
  # Buscar la secuencia
  ubicaSec = secuencias.find(sec)
  #print "Se encontró en:",ubicaSec
  if ubicaSec <> -1:
    # Ahora, buscar el ">" con el nombre de la secuencia
    ubicaNombre = secuencias.rfind(">",0,ubicaSec)
    #print "Nombre en:",ubicaNombre
    finlinea = secuencias.find("\n",ubicaNombre)
    print "Se encontró en ",secuencias[ubicaNombre:finlinea]
  if ubicaSec == -1:
    print "No se encontró"
```

■

### Cómo funciona

La función `buscarSecuencia` recibe una parte de una secuencia como entrada. Abre el archivo **parasitos.txt** (en la carpeta de medios especificada mediante `setMediaPath`), lee todos los datos y los coloca en la cadena `secuencias`. Para buscar la secuencia en la cadena `secuencias` usamos `find`. Si se encuentra (es decir, que el resultado no sea –1), entonces buscamos *hacia atrás* partiendo desde donde encontramos la secuencia (`ubicaSec`) hacia el principio de la cadena (0) para encontrar el ">" que inicia la secuencia. Luego buscamos hacia delante partiendo desde el signo "mayor que" hasta el final de la línea ("\n"). Esto nos indica en qué parte de la cadena `secuencias` original podemos encontrar el nombre del parásito del que proviene nuestra subsecuencia de entrada.

Leemos todo el contenido del archivo y lo colocamos en una cadena grande, y después procesamos esa cadena. Pero, si tenemos archivos muy grandes tal vez sea más conveniente procesar el contenido una línea a la vez. Para ello podemos usar la función `readlines`, que devuelve una lista de cadenas con una cadena por cada línea del archivo. Por ejemplo, si desea modificar cada ocurrencia de una cadena en un archivo, podría usar la siguiente función.

**Programa 94: reemplazo de todas las ocurrencias de una palabra en un archivo**

```
def reemplazarPalabra(nombreArchivo,palabraOrig,palabraReemp):
  archivo = open(nombreArchivo,"rt")
  archivoSal = open ("sal-" + nombreArchivo,"wt")
  for linea in archivo.readlines():
    nuevaLinea = linea.replace(palabraOrig,palabraReemp);
    archivoSal.write(nuevaLinea)
  archivo.close()
  archivoSal.close()
```

■

Hay programas que vagan por Internet, recopilando información de páginas Web. Por ejemplo, la página de noticias de Google (`http://news.google.com`) no está escrita por reporteros. Google tiene programas que obtienen los encabezados de *otros* sitios de noticias. ¿Y cómo funcionan estos programas? Lo único que hacen es descargar páginas de Internet y extraer las piezas deseadas.

Por ejemplo, digamos que usted desea escribir una función para obtener la temperatura actual, leyéndola de una página del clima local. En Atlanta, una buena página del clima es `http://www.ajc.com/weather`: la página del clima de *Atlanta Journal-Constitution*. Al examinar el código fuente podemos descubrir en dónde aparece la temperatura actual en la página y cuáles son las características clave del texto a su alrededor para extraer sólo la temperatura. A continuación le mostramos la parte relevante de la página que Mark encontró un día:

```
<td ><img src="/shared-local/weather/images/ps.gif"
width="48" height="48" border="0"><font size=-2><br>
  </font>
<font size="-1"
face="Arial, Helvetica, sans-serif"><b>Currently
  </b><br>
Partly sunny<br> <font size="+2">54<b>&deg;</b><
  /font>
<font face="Arial, Helvetica, sans-serif" size="+1">
F</font></font></td> </tr>
```

Podemos ver la palabra `Currently` en el ejemplo anterior, y luego la temperatura justo antes de los caracteres `<b>°`. Es posible escribir un programa para extraer esas piezas y devolver la temperatura, dado que la página del clima está guardada en un archivo llamado **ajc-weather.html**. Ahora bien, este programa no *siempre* funcionará con la página del clima de AJC actual, ya que su formato podría variar y el texto clave que buscamos podría moverse o desaparecer. Pero mientras que el formato sea el mismo, esta receta funcionará.

**Programa 95: obtener la temperatura de una página Web**

```
def buscarTemperatura():
  archivoClima = getMediaPath("ajc-weather.html")
  archivo = open(archivoClima,"rt")
  clima = archivo.read()
  archivo.close()
  # Buscar la temperatura
  ubicActual = clima.find("Currently")
  if ubicActual <> -1:
    # Ahora, buscar los caracteres "<b>&deg;" que van después de la temperatura
    ubicTemp = clima.find("<b>&deg;",ubicActual)
    tempinicio = clima.rfind(">",0,ubicTemp)
    print "Temperatura actual:",clima[tempinicio+1:ubicTemp]
  if ubicActual == -1:
    print "Deben haber cambiado el formato de la página - no puedo¬
    encontrar la temperatura"
```

■

### Cómo funciona

Esta función asume que el archivo **ajc-weather.html** está guardado en la carpeta de medios especificada con `setMediaPath`. La función `buscarTemperatura` abre y lee el archivo como texto, y después lo cierra. Buscamos la palabra "Currently". Si la encontramos (el resultado no es –1), buscamos el marcador de grados después de "Currently" (almacenado en `ubicActual`). Después buscamos hacia atrás el final de la marca anterior, ">". La temperatura está entre estos puntos. Si `ubicActual` es –1, desistimos debido a que no pudimos encontrar la palabra "Currently".

## 10.5 LA BIBLIOTECA ESTÁNDAR DE PYTHON

En todo lenguaje de programación hay una forma de extender sus funciones básicas con nuevas funciones. En Python, a esta funcionalidad se le conoce como **importar un módulo**. Como vimos antes, un **módulo** es en sí un archivo de Python con nuevas capacidades definidas en él. Cuando importamos un módulo mediante `import`, es como si hubiéramos escrito ese archivo de Python en ese punto y todos los objetos, funciones y variables que contiene se definen al instante.

Python cuenta con una extensa biblioteca de módulos que podemos usar para realizar una gran variedad de cosas, como acceder a Internet, generar números aleatorios y acceder a los archivos en un directorio; esto último es útil cuando desarrollamos páginas Web o trabajamos con video.

Usemos esto como nuestro primer ejemplo. El módulo que usaremos es `os`. La función en el módulo `os` que sabe cómo listar los archivos en un directorio es `listdir()`. Para acceder a la pieza del módulo usamos la notación punto.

```
>>> import os
>>> print os.listdir("C:\ip-book\mediasources\pics")
['students1.jpg', 'students2.jpg', 'students5.jpg',
'students6.jpg', 'students7.jpg', 'students8.jpg']
```

Podemos usar `os.listdir()` para agregar títulos a las imágenes en un directorio o insertar una declaración, como los derechos de autor. Ahora bien, `listdir()` devuelve el nombre de archivo base y el sufijo. Eso es suficiente para asegurarnos de tener imágenes y no sonidos o alguna otra cosa. Pero esto no nos proporciona las rutas completas para `makePicture`. Para obtener una ruta completa, podemos combinar el directorio de entrada con el nombre de archivo base de `listdir()`; pero necesitamos un delimitador de ruta entre estas dos piezas. Python tiene un estándar: si un nombre de archivo contiene "//", éste se reemplazará con el delimitador de ruta correcto para el sistema operativo que estemos usando.

**Programa 96: agregar un título a un conjunto de imágenes en un directorio**

```
import os

def tituloDirectorio(dir):
  for archivo in os.listdir(dir):
    print "Procesando:",dir+"//"+archivo
    if archivo.endswith(".jpg"):
      imagen = makePicture(dir+"//"+archivo)
      addText(imagen,10,10,"Propiedad de CS1315 en Georgia Tech")
      writePictureTo(imagen,dir+"//"+"titulado-"+archivo)
```

■

### Cómo funciona

La función `tituloDirectorio` recibe un directorio (nombre de ruta, como una cadena) como entrada. Después recorre cada nombre de archivo `archivo` en el directorio. Si el nombre de archivo termina con ".jpg", es muy probable que sea una imagen. Por ende, creamos la imagen a partir del `archivo` en el directorio **dir** dado. Agregamos texto a la imagen y luego la escribimos en un archivo llamado "titulado-" más el nombre de archivo, en el directorio **dir** especificado.

## 10.5.1 Más sobre importaciones y sus propios módulos

En realidad hay varias formas de la instrucción `import`. La que usamos aquí, `import módulo`, nos permite usar todos los módulos disponibles por medio de la notación punto. A continuación le mostramos varias opciones más:

- Podemos importar sólo unas cuantas cosas de un módulo y después acceder a ellas sin usar la notación punto. Esta forma es `from módulo import nombre`.

```
>>> from os import listdir
>>> print listdir(r"C:\Documents and Settings")
['Default User', 'All Users', 'NetworkService',
'LocalService', 'Administrator', 'Driver',
'Mark Guzdial']
```

Podemos usar la forma `from módulo import *` para importar *todo* del módulo y usarlo sin emplear la notación punto.

- Podemos usar `import módulo as nuevonombre` si deseamos importar un módulo pero luego usar `nuevonombre` para hacer referencia al mismo. Esto facilita el acceso a las bibliotecas de Java desde Jython. Algunas veces las bibliotecas de Java tienen nombres extensos, como `java.awt.event`. Podemos usar esta sintaxis para crear una abreviación; por ejemplo,

```
import java.awt.event as event
```

Así podremos hacer referencia a los elementos en `java.awt.event` como `event`.

Un módulo es sólo un archivo de Python. Como vimos antes en este libro, podemos importar nuestro propio código como un módulo. Si tiene la función `buscarTemperatura` en el archivo **archivoBuscarTemperatura.py** en el directorio de JES, sólo tiene que ejecutar `import buscarTemperatura from archivoBuscarTemperatura` y luego usar `buscarTemperatura` como si la hubiera escrito en el área del programa.

### 10.5.2 Otro módulo divertido: random

Hay otro módulo divertido que algunas veces también es de utilidad: `random`. La función base `random.random()` genera números aleatorios (distribuidos de manera uniforme) entre 0 y 1.

```
>>> import random
>>> for i in range(1,10):
...         print random.random()
...
0.11018977740272162
0.5841616608865653
0.7152962423363127
0.41236073127668016
0.994984180858527
0.14234698452702643
0.1725024426417061
0.23523626475704162
0.033444918664808476
```

Los números aleatorios pueden ser divertidos cuando se aplican a tareas como elegir palabras aleatorias de una lista. La función `random.choice()` hace eso.

```
>>> for i in range(1,5):
...         print random.choice(["Esta", "es", "una",
"lista", "de", "palabras", "en", "orden", "aleatorio"])
...
orden
en
en
Esta
```

De aquí podemos generar enunciados aleatorios eligiendo sustantivos, verbos y frases de listas al azar.

**Programa 97: generación aleatoria de un lenguaje**

```
import random

def enunciado():
  sustantivos = ["Mark", "Adam", "Angela", "Larry", "Jose", "Matt", "Jim"]
  verbos = ["corre", "patina", "canta", "brinca", "salta", "trepa",¬
  "discute", "ríe"]
  frases = ["en un árbol", "sobre un tronco", "muy fuerte", "alrededor¬
  del arbusto", "mientras lee el periódico"]
  frases = frases + ["muy mal", "mientras patina", "en vez¬
  de calificar", "mientras escribe en Internet"]
  print random.choice(sustantivos), random.choice(verbos),
  random.choice(frases)
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un solo comando en Python no puede dividirse en varias líneas.

■

## Cómo funciona

Simplemente creamos listas de sustantivos, verbos y frases, teniendo cuidado de que todas las combinaciones tengan sentido en términos de número. La instrucción `print` define la estructura deseada: un sustantivo aleatorio, después un verbo aleatorio y luego una frase aleatoria.

```
>>> enunciado()
Mark discute mientras lee el periódico
>>> enunciado()
Angela discute en vez de calificar
>>> enunciado()
Jose trepa mientras escribe en Internet
>>> enunciado()
Larry patina sobre un tronco
>>> enunciado()
Jim salta en un árbol
>>> enunciado()
Matt corre en vez de calificar
>>> enunciado()
Mark trepa en vez de calificar
>>> enunciado()
Mark corre mientras patina
```

El proceso básico aquí es común en los programas de simulación. Lo que tenemos es una estructura definida en el programa: una definición de lo que cuenta como sustantivo, un verbo y una frase, además de una declaración de que lo que deseamos es un sustantivo, luego un verbo y después una frase. La estructura se rellena con elecciones al azar. La pregunta interesante es cuánto podemos simular con una estructura y la aleatoriedad. ¿Podríamos simular la inteligencia de esta manera? ¿Y cuál es la diferencia entre una *simulación* de inteligencia y una computadora que piensa de verdad?

Imagine un programa que lee la entrada del usuario y luego genera un enunciado al azar. Tal vez haya unas cuantas *reglas* en el programa que busquen palabras clave y cómo responder a ellas, como:

```
if entrada.find("Madre") <> -1:
      print "Dime más acerca de tu madre"
```

Joseph Weizenbaum escribió un programa como éste hace muchos años, conocido como *Doctor* (que más tarde se llamó *Eliza*). Su programa actuaba como un psicoterapeuta rogeriano, repitiendo todo lo que usted decía, con cierto rasgo aleatorio pero buscando palabras clave de modo que pareciera estar "escuchando" de verdad. El programa se diseñó como una broma y no como un esfuerzo real por crear una simulación de inteligencia. Para consternación de Weizenbaum, la gente lo tomó con seriedad. Empezaron a tratar el programa como si fuera un terapeuta. Weizenbaum cambió su rumbo de investigación de la *inteligencia artificial* para concentrarse en el uso ético de la tecnología y la facilidad con que es posible engañar a las personas mediante ella.

### 10.5.3 Una muestra de las bibliotecas estándar de Python

Hemos visto algunos de los módulos estándar de Python como `os`, `sys`, y `random` hasta ahora. Hay *muchos* módulos en la Biblioteca estándar de Python. Hay muchas razones para usar estos módulos.

- Están muy bien escritos: rápidos, bien documentados y probados. Puede confiar en ellos y ahorrarse el esfuerzo.
- Siempre es conveniente reutilizar el código del programa. Es una buena práctica hacerlo.
- En un proceso de diseño abajo-arriba, partir de módulos existentes es una excelente forma de empezar un nuevo proyecto.

Algunos de los módulos que podría ser conveniente explorar son:

- Los módulos `datetime` y `calendar` saben cómo manipular fechas, horas y calendarios. Por ejemplo, puede averiguar qué día de la semana era cuando se firmó la Declaración de la independencia de Estados Unidos en 1776.

```
>>> from datetime import *
>>> independencia = date(1776,7,4)
>>> independencia.weekday()
    3
>>> #0 es lunes, por lo que 3 es jueves
```

- El módulo `math` conoce muchas funciones matemáticas importantes, como `sin` (seno) y `sqrt` (raíz cuadrada).
- El módulo `zipfile` sabe cómo leer y escribir archivos "zip" comprimidos.
- El módulo `email` provee herramientas para escribir su propio programa y manipular correo electrónico (por ejemplo, como un filtro anti-spam).
- El módulo `SimpleHTTPServer` es en realidad un servidor Web por sí solo; puede programarse en Python para iniciar su ejecución.

## RESUMEN DE PROGRAMACIÓN

## PIEZAS GENERALES DE UN PROGRAMA

| | |
|---|---|
| `random` | Módulo para generar números aleatorios o realizar elecciones al azar. |
| `os` | Módulo para manipular el sistema operativo. |

## FUNCIONES DE CADENAS, FUNCIONES Y PIEZAS

| | |
|---|---|
| `cadena[n], cadena[n:m]` | Devuelve el carácter en la cadena en la posición $n$ ([n]) o la subcadena de $n$ a $m - 1$. Recuerde que las cadenas empiezan en el índice 0. |
| `startswith` | Devuelve verdadero si la cadena empieza con la cadena de entrada. |
| `endswith` | Devuelve verdadero si la cadena termina con la cadena de entrada. |
| `find` | Devuelve el índice si se encuentra la cadena de entrada en la cadena; –1 en caso contrario. |
| `upper,lower` | Devuelve una nueva cadena convertida a mayúsculas o minúsculas. |
| `isalpha, isdigit` | Devuelve verdadero si toda la cadena es alfabética o numérica (dígitos), respectivamente. |
| `replace` | Recibe dos subcadenas de entrada; reemplaza todas las instancias de la primera con la segunda en la cadena especificada. |
| `split` | Descompone una cadena en una lista de subcadenas, usado la cadena de entrada como el delimitador. |

## FUNCIONES DE LISTAS Y PIEZAS

| | |
|---|---|
| `append` | Adjunta la entrada al final de la lista. |
| `remove` | Elimina la entrada de la lista. |
| `sort` | Ordena la lista. |
| `reverse` | Invierte la lista. |
| `count` | Cuenta el número de veces que aparece la entrada en la lista. |
| `max, min` | Dada una lista de números como entrada, devuelve el valor máximo o mínimo (respectivamente) en la lista. |

## PROBLEMAS

10.1 Genere una variable de cadena que contenga el enunciado "No haga eso". Introduzca una variable de cadena que contenga comillas dobles. Introduzca una variable de cadena que contenga un tabulador y también una variable de cadena que contenga un nombre de archivo con barras diagonales inversas.

10.2 Escriba una función que imprima una de cada dos letras en una cadena.

10.3 Escriba una función que reciba una entrada e imprima las letras en la cadena en orden inverso.

10.4 Escriba una función que busque y elimine la segunda ocurrencia de una cadena especificada en una cadena que se pase como parámetro.

10.5 Escriba una función para convertir a mayúsculas una de cada dos palabras en un enunciado que se pase como parámetro. Por ejemplo, si tenemos "El perro corrió una gran distancia", debería imprimir "El PERRO corrió UNA gran DISTANCIA".

10.6 Escriba una función que reciba un enunciado y un índice como parámetros, y devuelva el enunciado con la palabra en el índice convertida a mayúsculas. Por ejemplo, si la función recibe el enunciado "Me encanta el color rojo" y el índice 4, deberá devolver "Me encanta el color ROJO".

10.7 Escriba una función que reciba un enunciado como entrada y lo devuelva con las palabras revueltas (el orden de las palabras todo mezclado). Por ejemplo, si recibe "Acaso algo rima con naranja", podría devolver "Naranja rima acaso con algo".

10.8 Escriba una función que pueda encontrar el código postal de una persona a partir de una cadena delimitada que represente una dirección. Por ejemplo, podría leer una cadena que contenga "nombre:línea1:línea2:ciudad:estado:códigoPostal" y devolver el código postal.

10.9 Modifique la función `modificarChica` para que use `readLines` en vez de `read`.

10.10 Cree una función que genere imágenes más pequeñas (miniaturas) de todas las imágenes en un directorio. El usuario deberá introducir la ruta del directorio y el factor de escala.

10.11 Cree un mensaje secreto, codificando cada carácter de una cadena en un número mediante el uso de `ord`.

10.12 Cree una función que invierta los elementos en una lista.

10.13 Cree una función que sustituya una cadena que reciba como parámetro en una lista de cadenas con una nueva cadena.

10.14 ¿De verdad los números que se devuelven de la función `random.random()` son aleatorios? ¿Cómo se generan?

10.15 Busque el programa Eliza de Joseph Weizenbaum. ¿Puede escribir algo similar pero restringido a preguntas de la escuela?

10.16 Escriba respuestas cortas tipo ensayo a estas preguntas.

(a) Proporcione un ejemplo de una tarea para la cual no escribiría un programa y proporcione otro ejemplo de una tarea para la cual escribiría un programa.

(b) ¿Cuál es la diferencia entre un arreglo, una matriz y un árbol? Proporcione un ejemplo en donde hayamos usado cada uno de estos elementos para representar algunos datos de interés para nosotros.

(c) ¿Qué es la notación punto y cuándo debe usarse?

(d) ¿Por qué es el rojo un color que no debe usarse para la técnica chromakey?

(e) ¿Cuál es la diferencia entre una función y un método?

(f) ¿Por qué es un árbol una mejor representación que un arreglo para los archivos en un disco? ¿Por qué tiene tantos directorios en su disco y no sólo uno gigantesco?

(g) ¿Cuáles son algunas ventajas que tienen los gráficos basados en vectores sobre las representaciones gráficas de mapas de bits (como JPEG, BMP, GIF)?

10.17 Ya vimos código para reflejar las imágenes en el programa 20 (página 79) y también vimos código para reflejar sonidos en el programa 69 (página 187). Por ende, debería ser muy sencillo usar el mismo algoritmo para reflejar *texto*. Escriba una función que reciba una cadena y luego devuelva esa cadena reflejada, con la mitad frontal copiada sobre la segunda mitad.

10.18 Extienda la receta de la carta modelo para que reciba una entrada representada por el nombre y el tipo de una mascota, y haga referencia a la mascota en la carta. `"A su mascota "+tipoMascota+","+nombreMascota+", le encantará nuestra oferta"`debería generar `"A su mascota poodle, Fifi, le encantará nuestra oferta"`.

10.19 Imagine que tiene una lista de los sexos (como caracteres individuales) de todos los estudiantes en su clase, ordenados por apellido. La lista se verá algo así como "MFFMMMFFMFMMFFFM", en donde M es masculino y F es femenino. Escriba una función (como se indica a continuación) llamada `porcentajeSexos(cadena)` para aceptar una cadena que represente los sexos. Deberá contar todas las M y F en la cadena e imprimir la relación (como un número decimal) de cada sexo. Por ejemplo, si la cadena de entrada fuera "MFFF", entonces la función debería imprimir algo como: "Hay 0.25 hombres, 0.75 mujeres" (*sugerencia:* multiplique mejor algo por 1.0 para asegurarse de obtener puntos flotantes y no enteros).

10.20 Usted trabajó hasta altas horas de la noche en una tarea y no se dio cuenta de que escribió una gran parte de su trabajo final con sus dedos en las teclas incorrectas.

Usted quiso escribir "Ésta es una multitud violenta", pero lo que en realidad escribió fue: "Ew5a ew7ha m7l5857d v89leh5a".

En esencia, cambió: W por S, 5 por T, 7 por U, H por N, 8 por I y 9 por O (esas fueron las únicas pulsaciones de tecla incorrectas; por suerte se dio cuenta antes de tener más errores). Además, nunca presionó la tecla de mayúsculas, por lo que sólo nos preocupan las letras minúsculas.

Con base en lo que conoce sobre Python, decide escribir un programa rápido para corregir su texto. Escriba una función llamada `corregirError` que reciba una cadena como entrada y devuelva una cadena con los caracteres que deberían haberse escrito.

*10.21 Escriba una función llamada `crearGraficos` que reciba una lista como entrada. Lo primero que hará esta función es crear un lienzo a partir del archivo **640×480.jpg** en la carpeta **mediasources**. Usted dibujará en el lienzo de acuerdo con los comandos en la lista de entrada. Cada elemento de la lista será una cadena. Habrá dos tipos de cadenas en la lista:

*Un problema más desafiante

- "b 200 120" significa que debe dibujar un punto negro en la posición *x* 200 y la posición *y* 120 (200,120). Desde luego que los números cambiarán, pero el comando siempre será una "b". Puede suponer que los números de entrada siempre tendrán tres dígitos.
- "l 000 010 100 200" significa que debe dibujar una línea desde la posición (0, 10) hasta la posición (100, 200).

Así, una lista de entrada podría ser la siguiente: `["b 100 200","b 101 200","b 102 200","l 102 200 102 300"]` (pero puede tener cualquier número de elementos).

## PARA PROFUNDIZAR

El libro que usa Mark para trabajar con los módulos de Python es *Python Standard Library* de Frederik Lundh [29]. Encontrará la lista de módulos de la biblioteca junto con su documentación en `http://docs.python.org/library/`.

CAPÍTULO 11

# Técnicas avanzadas de texto: Web e información


11.1 REDES: CÓMO OBTENER NUESTRO TEXTO DE LA WEB

11.2 USO DE TEXTO PARA CAMBIAR DE UN MEDIO A OTRO

11.3 CÓMO MOVER INFORMACIÓN ENTRE MEDIOS

11.4 USO DE LISTAS COMO TEXTO ESTRUCTURADO PARA REPRESENTACIONES DE MEDIOS

11.5 CÓMO OCULTAR INFORMACIÓN EN UNA IMAGEN


### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Escribir programas para acceder de manera directa a Internet y usar la información de texto.
- Traducir un sonido o una imagen a texto y traducir texto de vuelta a un sonido o a una imagen.
- Ocultar un mensaje en una imagen.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Escribir un programa para acceder a Internet y devolver información procesada.
- Demostrar que la información puede codificarse de muchas formas.

## 11.1 REDES: CÓMO OBTENER NUESTRO TEXTO DE LA WEB

Una *red* se forma siempre que haya computadoras que se comuniquen entre sí. Muy raras veces la comunicación se lleva a cabo con voltajes a través de cables, la forma en que una computadora codifica los ceros y unos en su interior. Es muy difícil mantener voltajes específicos a través de las distancias. En vez de ello, los ceros y unos se codifican de alguna otra forma. Por ejemplo, un **módem** (*modulador-demodulador* en sentido literal) es un aparato que asocia ceros y unos a distintas frecuencias de audio. Para los humanos, suena como un puñado de abejas zumbadoras, pero para los módems y las computadoras, es lenguaje binario puro.

Al igual que las cebollas y los ogros, las redes tienen capas. En el nivel inferior se encuentra el sustrato físico. ¿Cómo se pasan las señales? Los niveles superiores definen la forma en que se codifican los datos. ¿En qué consiste un cero? ¿Un uno? ¿Enviamos un bit a la vez? ¿Un **paquete** de bytes a la vez? Incluso hasta los niveles superiores definen el **protocolo** para la comunicación. ¿Cómo es que una computadora le dice a otra que desea hablar y qué es

lo que quiere decirle? ¿Cómo nos dirigimos a la computadora? Al pensar en estas distintas capas, y al mantenerlas diferenciadas, es muy sencillo intercambiar una parte sin tener que cambiar las otras. Por ejemplo, la mayoría de las personas con una conexión directa a una red usan una conexión alámbrica a una red **Ethernet**, pero en realidad Ethernet es un protocolo de nivel medio que también funciona a través de redes inalámbricas.

Considere cómo es posible acceder a las mismas páginas Web con un teléfono celular, una laptop mediante Wi-Fi o un equipo de escritorio mediante una conexión alámbrica directa a Internet. Las capas superiores del protocolo son muy distintas en cada uno de estos tres ejemplos. Un teléfono celular accede a Internet a través de la red celular. La laptop que usa Wi-Fi utiliza una capa superior de radiofrecuencia que se conecta al resto de Internet por medio de un *router*. No obstante, en las capas intermedia e inferior de la red los tres dispositivos acceden a la misma información de una forma idéntica. Los protocolos que indican la forma en que un navegador pide información a un servidor Web son todos iguales, sin importar el tipo de dispositivo ni el tipo de capa que se utilice para acceder a la red.

Los humanos también tenemos protocolos. Si Mark camina hacia usted, extiende su mano y dice: "Hola, mi nombre es Mark", lo más probable es que usted extienda su mano y diga algo como: "Mi nombre es Carolina" (suponiendo que se llame Carolina; si ése no fuera su nombre, sería bastante gracioso). En cada cultura hay un protocolo que indica cómo una persona debe saludar a otra. Los protocolos de computadora tratan sobre lo mismo, sólo que están escritos para comunicar el proceso con exactitud. Lo que se dice no es tan diferente. Una computadora puede enviar el mensaje 'HELO' a otra para iniciar una conversación (no sabemos por qué los que escribieron el protocolo no pusieron otra letra L para deletrear la palabra correcta en inglés), y una computadora puede enviar BYE para terminar la conversación (inclusive algunas veces al inicio de un protocolo de computadora se le conoce como *handshake*, o negociación). Todo esto se trata de establecer una conexión y asegurarse de que ambos lados comprendan lo que está ocurriendo.

**Internet** es una red de redes. Si usted tiene un dispositivo en su hogar para que sus computadoras puedan comunicarse entre sí (un **router**), entonces tiene una red. Con sólo eso, es probable que pueda copiar archivos entre las distintas computadoras e imprimirlos. Si conecta su red al extenso mundo de Internet (por medio de un **Proveedor de servicios de Internet**, o **ISP**), su red se vuelve parte de Internet.

Internet se basa en un conjunto de acuerdos acerca de varias cosas:

- *Cómo debemos dirigirnos a las computadoras:* en la actualidad, cada computadora en Internet tiene un número de 32 bits asociado con ella: valores de cuatro bytes que por lo general se escriben de la siguiente manera, separados por puntos: "101.132.64.15". A estos valores se les conoce como **direcciones IP** (o direcciones de protocolo Internet).

  Hay un sistema de **nombres de dominio** mediante el cual las personas pueden referirse a computadoras específicas sin necesidad de conocer sus direcciones IP. Por ejemplo, cuando usted accede a `http://www.cnn.com`, en realidad está accediendo a `http://157.166.226.26/`. Esto funcionó para nosotros la última vez que lo probamos, pero podría cambiar. Hay una red de *servidores de nombres de dominio* que llevan el registro de nombres como "www.cnn.com" y los asocian a direcciones como "157.166.226.26". El hecho de que los números pueden cambiar es una de las ventajas del servidor de nombres de dominio: el nombre sigue siendo el mismo y puede apuntar a cualquier dirección. Hay veces que uno está conectado a Internet sin poder acceder a los sitios Web favoritos si el servidor de nombres de dominio no funciona. Tal vez podamos acceder si escribimos la dirección IP en forma directa.

- *Cómo se comunican las computadoras:* los datos se colocan en *paquetes* que tienen una estructura bien definida, incluyendo la dirección IP del emisor, la dirección IP del receptor y una cantidad de bytes por paquete.
- *Cómo se encaminan los paquetes a través de Internet:* Internet se diseñó en tiempos de la *Guerra Fría*. Fue creada para resistir a un ataque nuclear. Si una sección de Internet se destruye (o tal vez se dañe o quede bloqueada como una forma de censura), el mecanismo de encaminamiento de paquetes de Internet encontrará una ruta para sortear el daño.

Pero las capas superiores de la red definen lo que *significan* los datos que se están transmitiendo. Una de las primeras aplicaciones que se colocaron sobre Internet fue el correo electrónico. Con el paso de los años, los protocolos de correo evolucionaron en estándares como **POP (protocolo de oficina postal)** y **SMTP (protocolo simple de transferencia de correo)**. Otro de los primeros protocolos importantes es **FTP (protocolo de transferencia de archivos)**, el cual nos permite transferir archivos entre computadoras.

Estos protocolos no son muy complicados. Al terminar la comunicación, es probable que una computadora diga BYE o QUIT a la otra. Cuando una computadora le dice a otra que acepte un archivo vía FTP, literalmente le dice "STO nombrearchivo" (de nuevo, los primeros desarrolladores de computadoras no querían ocupar dos bytes más para decir "STORE").

*World Wide Web* es otro conjunto más de acuerdos, desarrollados en su mayoría por Tim Berners-Lee. El servicio Web está basado sobre Internet y lo único que hace es agregar más protocolos sobre los que ya existen.

- *Cómo referirnos a las cosas en Internet*: en Web, hacemos referencia a los recursos mediante el uso de **URL (localizadores uniformes de recursos)**. Un URL especifica el protocolo que se debe usar para dirigirse al recurso, el nombre de dominio o el **servidor** que puede proporcionar el recurso, y la **ruta** hacia el recurso en ese servidor. Por ejemplo, un URL como `http://www.cc.gatech.edu/index.html` dice: "Usar el protocolo HTTP para hablar con la computadora en `www.cc.gatech.edu` y pedirle el recurso index.html".

  No es posible acceder a todos los archivos en todas las computadoras conectadas a Internet por medio de un URL. Hay ciertas precondiciones para poder acceder a un archivo mediante un URL. En primer lugar, una computadora a la que se pueda acceder por Internet debe estar ejecutando una pieza de software que comprenda un protocolo que los navegadores Web entiendan, por lo general **HTTP (protocolo de transferencia de hipertexto)** o FTP. A una computadora que ejecuta dicha pieza de software se le denomina *servidor*. Un navegador que accede a un servidor se llama **cliente**. En segundo lugar, es común que un servidor tenga un *directorio de servidor* accesible a través de ese servidor. Sólo estarán disponibles los archivos en ese directorio o en los subdirectorios dentro de ese directorio.
- *Cómo servir documentos:* el protocolo más común en Web es HTTP. Define cómo se sirven los recursos en Web. HTTP es bastante simple: su navegador literalmente dice a un servidor cosas como "GET index.html" (¡sólo esas letras!).
- *Cómo se dará formato a los documentos:* para aplicar formato a los documentos en Web se usa **HTML (Lenguaje de marcación de hipertexto)**.

Tal vez haya observado que el término **hipertexto** aparece con frecuencia en referencia a Web. El hipertexto es, en sentido literal, texto no lineal. Ted Nelson inventó el término *hipertexto* para describir el tipo de lectura que todos hacemos en Web pero que no existía antes de las computadoras: leer un poco en una página, después hacer clic en un vínculo y

leer un poco por allá, luego hacer clic en REGRESAR y continuar leyendo en donde nos quedamos. La idea básica del hipertexto se remonta a Vannevar Bush, uno de los consejeros de ciencia del presidente Franklin Roosevelt, pero no fue sino hasta que surgió la computadora que pudimos visualizar la implementación del modelo de Bush de un *Memex*: un dispositivo para capturar flujos de pensamiento. Tim Berners-Lee inventó el servicio Web y sus protocolos como una forma de dar soporte a la publicación rápida de hallazgos de investigación, con conexiones entre los documentos. En definitiva, Web no es el sistema más moderno de hipertexto. Los sistemas similares a los que Ted Nelson usó no permitían "vínculos muertos" (vínculos que ya no son accesibles). Pero con todo y sus defectos, Web *funciona.*

Un navegador (como Internet Explorer, Google Chrome, Opera) sabe mucho sobre Internet. Por lo general conoce varios protocolos, como HTTP, FTP, *gopher* (uno de los primeros protocolos de hipertexto) o *mailto* (SMTP). Conoce el HTML, cómo aplicarle formato y cómo obtener recursos referenciados dentro del HTML, como imágenes JPEG. También es posible acceder a Internet sin la mayor parte de toda esa sobrecarga. Los clientes de correo (Outlook y Apple Mail, por ejemplo) conocen algunos de estos protocolos sin necesidad de conocerlos todos. Incluso JES comprende un poco sobre SMPT y HTTP para dar soporte a la entrega de asignaturas.

Al igual que otros lenguajes modernos, Python cuenta con módulos que dan soporte al acceso a Internet sin toda la complejidad de un navegador. En esencia, es posible escribir programas pequeños que sean clientes. El módulo `urllib` de Python nos permite abrir direcciones URL y leerlas como si fueran archivos. A continuación le mostraremos cómo acceder al sitio Web de noticias `http://www.cnn.com` y mostrar los primeros 100 caracteres de la página Web que se recibe en respuesta.

```
>>> import urllib
>>> conexion = urllib.urlopen("http://www.cnn.com")
>>> noticias = conexion.read()
>>> conexion.close()
>>> noticias[1:100]
' <!DOCTYPE html>\n <html xmlns:fb="http://ogp.me/ns/fb#">\n
<head>\n <!--googleoff: all'
```

En un capítulo anterior creamos un programa lector de temperaturas para obtener información de una página del clima que se guardó directamente de Internet. Aún podemos hacer eso con sitios como `http://www.weather.com`. Pero también podemos obtener otro tipo de información en vivo de otros sitios Web.

A continuación veremos un programa que puede recuperar las "Actividades e intereses" de una página de Facebook, si acaso están disponibles en forma pública para un nombre de usuario o perfil especificado. Vamos a buscar piezas de HTML, lo que discutiremos en el capítulo 12. En realidad no necesita comprender sobre HTML para entender cómo funciona este ejemplo.

**Programa 98: obtener las actividades e intereses de una página de Facebook**

```
def fbActividadesIntereses(nombre):
  import urllib  # Podría ir arriba también
  intereses = ""
  # Obtener la página de Facebook de un nombre
  conexion = urllib.urlopen("http://www.facebook.com/"+nombre+"?ref=pb")
  fb = conexion.read()
  conexion.close()
  # Averiguemos si hay "Actividades"
```

```
  actInicio = fb.find("Actividades")
  ubicPagina = actInicio   # Lugar de inicio para buscar páginas
  if actInicio <> -1: # ¿Encontramos Actividades e intereses?
    # Ahora, un ciclo para buscar todas las actividades e intereses
    ubicPagina = fb.find("http://www.facebook.com/pages/",ubicPagina)
    while (ubicPagina <> -1):
      interesInicio = ubicPagina+30  # Para "http://www.facebook.com/
      pages/"
      interesFin = fb.find("/",interesInicio+1)
      interes = fb[interesInicio:interesFin]
      # Llegamos a la última página, que en realidad es un pie de página?
      if not ("footer" in interes):
        intereses = intereses+fb[interesInicio:interesFin]+","
        ubicPagina = fb.find("http//www.facebook.com/pages/",
        ubicPagina+1)
      else:
        ubicPagina = -1
  return intereses[:-1]
```

■

Ahora veamos algunos ejemplos de su uso:

```
>>> fbActividadesIntereses("mark.guzdial")
'Running,Running,Beer,Beer,Engineers-Bookstore,POLISH-
CLUB-OF-ATLANTA,Napoleons,
Sprig-Restaurant-and-Bar,The-Barrelhouse,National-Science-
Foundation,
Running-of-the-Ears,Cognitive-Science-Society,
Ludoliteracy,Seymour-Papert,Georgia-Techs-Computational-
Media-program.
ACM-Association-for-Computing-Machinery,Herbert-Simon,
Georgia-Computes,
Wait-Wait-Dont-Tell-Me,Sublime-Doughnuts,Terry-Pratchetts-
Discworld,
Dr-Horribles-Sing-Along-Blog,Firefly,Doug'
>>> fbActividadesIntereses("alison.clear")
'Grandchildren,Christmas,Travel,Scary-washing-machine,
Engage-Learning,Tea-Total,
ACM-SIGCSE,Jeremy-the-Sign-Language-Guy,
Bob-Parker-Deserves-a-Knighthood-for-all-the-hard-work-he-
has-done,
Leo-the-Neo,LightORama,Desparate-Housewives'
>>> fbActividadesIntereses("alfredtwo")
'Association-for-Computing-Machinery,Firearms,FIRST-
Robotics,Reading,Imagine-Cup-2010,
Microsoft-TeachTec,Ron-Charette,Christinas-Country-Cafe,
NH-TechFest,BookHampton,KinectEDucation,Microsoft-New-
England-Research-and-Development,
First-Robotics-for-Inspiration-and-Recognition-of-
Science-and-Technology,
Microsoft-Canada-Partners-in-Learning,ISTE-SIGCT,DreamSpark,
EduConnect-Microsoft,Bishop-Guertin-High-School,Microsoft-
Innovation-Center,Bytes-by-MSDN,
Bill-Miller-FRC-Team,Cool-Cat-Teacher,Brooklyn-Technical-
```

```
High-School,Bookey-Consulting,Inc,
Photosynth,Smooth-Fusion,Microsoft-Technology-Center-Boston,
Computer-Science-Teachers-Association,
HP-CodeWars,ISTE'
```

Algunas personas no tienen URL en Facebook, por lo que puede usar su número de perfil.

```
>>> fbActividadesIntereses("profile.php?id=554818875")
'Gardening,Reading-Books,Singing,Ride-Horses'
```

### Cómo funciona

La función `fbActividadesIntereses` está haciendo algo similar al programa para buscar la información del clima de la página Web de AJC. Tuvimos que realizar un extenso proceso de prueba y error para que esto funcionara bien. Usamos el módulo `urllib` para leer la página Web como una cadena y luego buscar información en la cadena de la misma forma que hicimos con el archivo en el capítulo anterior (desde luego que todo esto dejará de funcionar si Facebook cambia la forma en que muestra las actividades e intereses).

- Primero, cerramos sesión en Facebook (ya que nuestro programa no actuaría correctamente si tuviéramos la sesión abierta) y buscamos por los alrededores. Descubrimos que un URL que muestra "Actividades" tenía la forma siguiente: `http://www.facebook.com/mark.guzdial?ref=pb`.
- Al acceder a esa página y examinar la cadena devuelta, vimos que la sección de actividades empezaba con:

  ```
  <div id="pagelet_Actividades"
  data-referrer="pagelet_Actividades"
  data-referrer="pagelet_Actividades">
  ```

- Después cada actividad se listaba con una referencia de URL, como se indica a continuación:
  `<a href="http://www.facebook.com/pages/Gardening/ 105975426100728">`.
  El nombre del interés aparece después de `"/pages/"` y antes del siguiente `"/"`.
- La última referencia a un URL de "pages" en Facebook (`http://www.facebook.com/pages`) era a un pie de página ("footer"). Si llegamos hasta ahí, quiere decir que fuimos demasiado lejos.

Una vez que tenemos esta información, podemos escribir nuestro programa:

- Primero importamos `urllib`. También podríamos hacer esto *antes* de nuestra función, y así todas las funciones en el mismo archivo podrían acceder a urllib.
- Establecemos una cadena llamada `intereses`, para que contenga todos nuestros intereses.
- Usamos el `nombre` que recibe la función como entrada para construir un URL, `conexion = urllib.urlopen("http://www.facebook.com/"+nombre+"?ref=pb")`. Después descargamos la página como una cadena en la variable `fb` y cerramos la conexión.
- Ahora empezamos a buscar. Tratamos de encontrar la indicación del inicio de las actividades e intereses (`fb.find("Actividades")`). Si encontramos esto (el resultado no es −1), podemos empezar a buscar referencias de páginas de Facebook. Usaremos esa ubicación (`ubicPagina`) como el punto de inicio para nuestra búsqueda de actividades.

- Como no sabemos cuántos intereses habrá, iniciamos un ciclo. Buscamos una referencia de página con `ubicPagina = fb.find("http://www.facebook.com/pages/", ubicPagina)`.
- Si encontramos una, averiguamos el final del URL (`interesFin = fb.find("/", interesInicio+1)`)y luego extraemos el nombre del interés (`interes = fb [interesInicio:interesFin]`). Si ese interés contiene el nombre "footer", es muy probable que hayamos ido demasiado lejos; establezca la `ubicPagina` en −1 para indicar eso. Si no es así, agregamos el interés a nuestra cadena intereses con una coma para separar el siguiente interés.
- Al terminar, recortamos la coma final: `return intereses[:-1]`.

**Idea de ciencias computacionales: ¿es legal el "scraping"?**

Lo que hicimos en el ejemplo anterior se conoce como *scraping*. Escribimos un programa para obtener datos utilizados por la computadora a partir de un resultado diseñado para humanos. Si extraemos información de la pantalla, se conoce como *screen scraping*. Es una forma nada elegante e ineficiente de recopilar datos. Los datos que recopilamos están disponibles para el público; lo único que hacemos es automatizar ese proceso. No hay nada ilegal en ello. Ahora bien, el acuerdo de miembros de Facebook (por lo menos al momento de escribir este libro) dice, "Usted acepta no usar secuencias de comandos automatizadas para recolectar información del Servicio o del Sitio". Estamos usando de manera explícita una secuencia de comandos automatizada para recolectar información del sitio de Facebook. Sin embargo, nuestro programa no está accediendo al sitio como miembro de Facebook. Sólo accede a la información disponible en forma pública, sin iniciar sesión como miembro. Facebook cuenta con una interfaz para acceder a la información de manera automática. Ésa sería una forma más elegante y menos complicada de recopilar esta información. Este ejemplo nos muestra *cómo* podríamos usar la técnica de *scraping* para obtener información de Facebook. No es la mejor forma de recopilar esa información.

■

Podemos usar FTP a través de la biblioteca `ftplib` en Python.

```
>>> import ftplib
>>> conexion = ftplib.FTP("cleon.cc.gatech.edu")
>>> conexion.login("guzdial","micontraseña")
'230 User guzdial logged in.'
>>> conexion.storbinary("STOR barbara.jpg",open(getMediaPath("barbara.jpg")))
'226 Transfer complete.'
>>> conexion.storlines("STOR JESintro.txt",open("JESintro.txt"))
'226 Transfer complete.'
>>> conexion.close()
```

Para crear interacción en Web, necesitamos programas que generen HTML. Por ejemplo, al escribir una frase en un área de texto y después hacer clic en el botón BUSCAR, hacemos que un programa se ejecute en el servidor que realiza la búsqueda y después *genera* el HTML (página Web) que vemos como respuesta. Python es un lenguaje común para este tipo de programación. Sus módulos, su facilidad de uso de comillas y su facilidad de escritura lo convierten en una excelente opción para escribir programas Web interactivos.

## 11.2 USO DE TEXTO PARA CAMBIAR DE UN MEDIO A OTRO

Como dijimos al principio de este capítulo, podemos considerar el texto como *unimedia*. Podemos asociar de un sonido a un texto y viceversa, y podemos hacer lo mismo con imágenes. Incluso podemos hacer algo más interesante: podemos pasar del sonido al texto y luego a imágenes.

¿Por qué querríamos hacer algo como esto? ¿Por qué debería interesarnos transformar los medios de esta forma? Por las mismas razones que nos interesa la digitalización de medios en general. Los medios digitales transformados en texto pueden transmitirse con mayor facilidad de un lugar a otro, aparte de que es más sencillo comprobar los errores e incluso corregirlos. Los estándares internacionales para el correo electrónico de Internet[1] requieren que el envío de archivos binarios (como imágenes y sonidos) a través de un mensaje de correo electrónico se convierta primero a texto. Esto no es muy difícil de lograr, y ocurre de una manera casi transparente para el usuario del correo electrónico. *Es posible poner la información en muchas representaciones diferentes. Podemos elegir nuevas representaciones de la información para poder realizar nuevas cosas.*

Es fácil asociar sonido al texto. El sonido es sólo una serie de muestras (números). Podemos escribir fácilmente estas muestras a un archivo.

**Programa 99: escribir un sonido a un archivo en forma de números de texto**

```
def sonidoATexto(sonido,nombrearchivo):
  archivo = open(nombrearchivo,"wt")
  for s in getSamples(sonido):
    archivo.write(str(getSampleValue(s))+"\n")
  archivo.close()
```

■

### Cómo funciona

Recibimos un sonido y un nombre de archivo como entrada y luego abrimos el nombre de archivo para escribir texto. A continuación iteramos por cada muestra y la escribimos en el archivo. Usamos la función `str()` en este caso para convertir un número en su representación de cadena, para poder agregarle una nueva línea y escribirla en un archivo.

¿Qué hacemos con el sonido como texto? Lo manipulamos como una serie de números, como con Excel (figura 11.1). Es muy sencillo realizar modificaciones, como multiplicar cada muestra por 2.0 en este caso. Incluso podemos graficar los números y ver el mismo tipo de gráfico de sonido que vemos en MediaTools (figura 11.2) (no obstante, obtendrá un error: a Excel no le gusta graficar más de 32000 puntos y, a 22000 muestras por segundo, 32000 muestras no es un sonido largo).

¿Cómo convertimos una serie de números de vuelta en un sonido? Por ejemplo, suponga que realiza cierta modificación en los números y ahora desea escuchar el resultado. ¿Cómo puede lograrlo? La mecánica de Excel es sencilla: sólo hay que copiar la columna que desea en una nueva hoja de cálculo, guardarla como texto y luego obtener el nombre de la ruta del archivo de texto que va a usar en Python.

El programa en sí es un poco más complicado. Al pasar de sonido a texto, sabíamos que podíamos usar `getSamples` para escribir todas las muestras. Pero ¿cómo sabemos cuántas líneas hay en el archivo? No podemos usar `getLines`, ya que no existe. Debemos tener cuidado con dos problemas: (a) tener más líneas en el archivo de las que podamos ajustar en el sonido que vamos a usar para leer, y (b) quedarnos sin líneas antes de llegar al final del sonido.

[1]RFC 822, en caso de que le interese. En realidad, el correo electrónico de Internet se define como compuesto por texto solamente.

| | A |
|---|---|
| 58 | 6500 |
| 59 | 5906 |
| 60 | 4762 |
| 61 | 2974 |
| 62 | 617 |
| 63 | -2215 |
| 64 | -4982 |
| 65 | -7545 |
| 66 | -9391 |
| 67 | -10548 |
| 68 | -11155 |
| 69 | -10973 |
| 70 | -10163 |
| 71 | -8769 |
| 72 | -6781 |
| 73 | -4556 |
| 74 | -2245 |
| 75 | -363 |
| 76 | 1406 |
| 77 | 2700 |
| 78 | 3571 |
| 79 | 4093 |
| 80 | 4205 |
| 81 | 4015 |
| 82 | 3390 |
| 83 | 2760 |
| 84 | 2172 |
| 85 | 1503 |
| 86 | 1050 |
| 87 | 611 |

archivosonido

**FIGURA 11.1**
Archivo de sonido a texto leído en Excel. *Fuente:* Capturas de pantallas de Microsoft Excel 20010. Copyright © 2010 por Microsoft Corporation. Reimpreso con permiso.

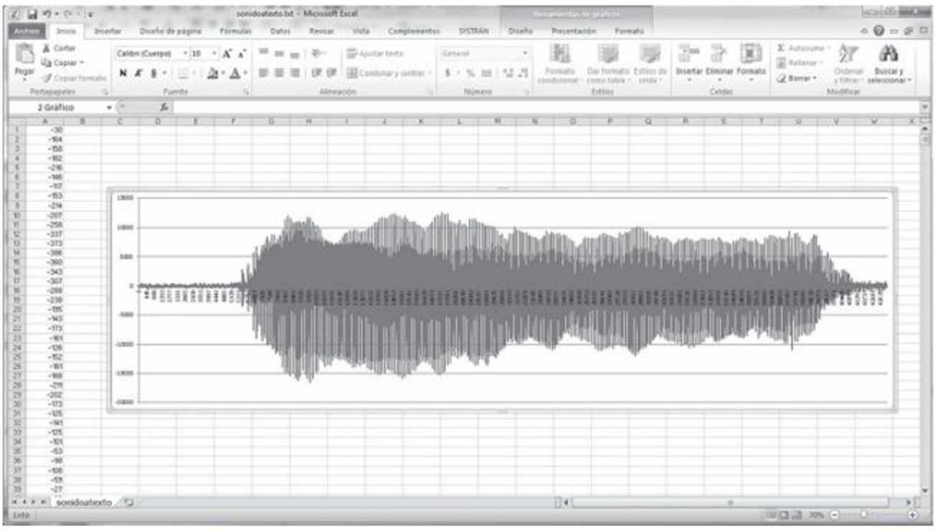


**FIGURA 11.2**
Archivo de sonido a texto graficado en Excel. *Fuente:* Capturas de pantallas de Microsoft Excel 20010. Copyright © 2010 por Microsoft Corporation. Reimpreso con permiso.

Para hacer esto usaremos un ciclo `while`, que vimos brevemente en un capítulo anterior. Al igual que una instrucción `if`, el ciclo `while` recibe una expresión y ejecuta su bloque si resulta ser verdadera. Difiere de un `if` en cuanto a que *después* de ejecutar el bloque, un ciclo `while` *vuelve a evaluar* la expresión. Si aún es verdadera, se ejecutará de nuevo todo el bloque. En un momento dado es de esperarse que la expresión se vuelva falsa; a continuación se ejecutará la línea que va después del bloque del `while`. Si no se vuelve falsa, tendremos lo que se conoce como *ciclo infinito*: el ciclo seguirá ejecutándose para siempre (en sentido hipotético).

```
while 1==1:
  print "Esto seguirá imprimiéndose hasta que apaguemos la computadora."
```

Para nuestro ejemplo de texto a sonido, queremos seguir leyendo muestras del archivo y almacenarlas en el sonido *siempre y cuando* haya números en el archivo y *siempre y cuando* quede espacio en el sonido. Usamos la función `float()` para convertir el número de cadena en un número real (también funcionaría con `int`, pero esto nos da la oportunidad de introducir a `float`).

```
>>> print 2*"123"
123123
>>> print 2 * float("123")
246.0
```

**Programa 100: convertir un archivo de números de texto en un sonido**

```
def textoASonido(nombrearchivo):
  # Preparar el sonido
  sonido = makeSound(getMediaPath("sec3silence.wav"))
  indiceSonido = 0
  # Preparar el archivo
  archivo = open(nombrearchivo,"rt")
  contenido=archivo.readlines()
  archivo.close()
  indiceArchivo = 0
  # Seguir hasta que se agote el espacio del sonido o el contenido del archivo
    while (indiceSonido < getLength(sonido)) and (indiceArchivo < len(contenido)):
    muestra=float(contenido[indiceArchivo])
    # Obtener la línea del archivo
    setSampleValueAt(sonido,indiceSonido,muestra)
    indiceArchivo = indiceArchivo + 1
    indiceSonido = indiceSonido + 1
  return sonido
```

■

## Cómo funciona

La función `textoASonido` recibe un nombre de archivo como entrada, el cual contiene las muestras en forma de números. Abrimos un sonido de silencio de tres segundos para contener el sonido. `indiceSonido` representa la siguiente muestra a leer e `indiceArchivo` representa el siguiente número a leer para el `contenido` de la lista del archivo. El ciclo `while` indica que debemos continuar *hasta que* el `indiceSonido` se pase de la longitud del sonido *o* el `indiceArchivo` se pase del final del archivo (en la lista `contenido`). En un ciclo convertimos la siguiente cadena de la lista en un valor `float`, establecemos el valor de la muestra en ese valor y luego incrementamos tanto `indiceArchivo` como `indiceSonido`. Al terminar (mediante cualquiera de las condiciones en el `while`), devolvemos el sonido de entrada mediante `return`.

## 11.3 CÓMO MOVER INFORMACIÓN ENTRE MEDIOS

No es *necesario* asociar los sonidos al texto y luego regresar de vuelta a los sonidos. Podríamos optar mejor por pasar a una imagen. El siguiente programa recibe un sonido y asocia cada una de las muestras a un pixel. Todo lo que tenemos que hacer es definir nuestra asociación, que especifica la forma en que deseamos representar las muestras. Elegimos una muy simple: si la muestra es mayor a 1000, el pixel será rojo; si es menor a −1000 será azul y todo lo demás será verde (figura 11.3).

Ahora tenemos que lidiar con el caso en el que se agoten las muestras antes de quedarnos sin píxeles. Para ello usaremos otra nueva construcción de programación: la instrucción `break`. Esta instrucción detiene nuestro ciclo actual y pasa a la línea que le sigue justo después. En este ejemplo, si nos quedamos sin muestras detendremos el ciclo `for` que procesa los píxeles.

**Programa 101: visualización del sonido**

```
def sonidoAImagen(sonido):
  imagen = makePicture(getMediaPath("640x480.jpg"))
  indiceSonido = 0
  for p in getPixels(imagen):
    if indiceSonido == getLength(sonido):
      break
    muestra = getSampleValueAt(sonido,indiceSonido)
    if muestra > 1000:
      setColor(p,red)
    if muestra < -1000:
      setColor(p,blue)
```

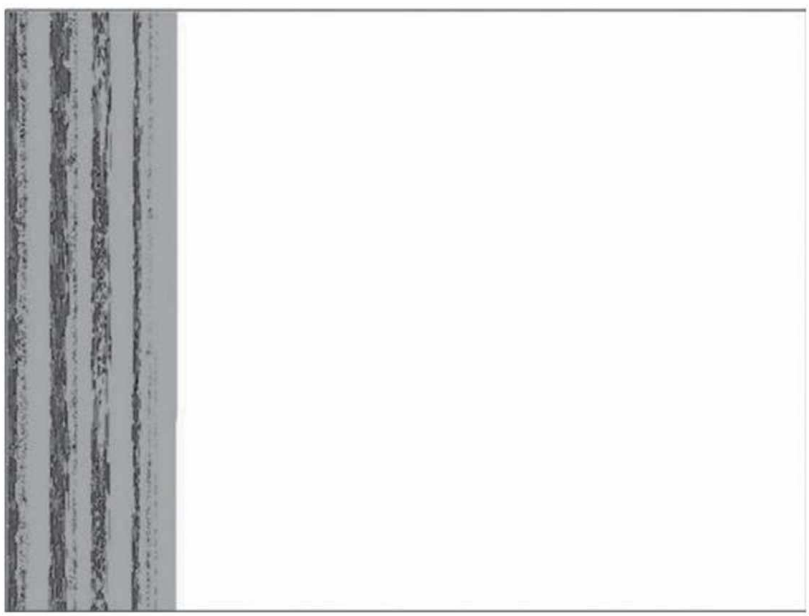


**FIGURA 11.3**
Una visualización del sonido "This is a test".

```
    if muestra <= 1000 and muestra >= -1000:
      setColor(p,green)
    indiceSonido = indiceSonido + 1
  return imagen
```

■

### Cómo funciona

En `sonidoAImagen` abrimos una imagen en blanco de 640 × 480 y recibimos un sonido como entrada. Para cada uno de los píxeles en la imagen, obtenemos un valor de muestra en `indiceSonido` e ideamos una asociación a un color, después establecemos el pixel `p` a ese color. Luego incrementamos el `indiceSonido`. Si alguna vez `indiceSonido` se pasa del final del sonido, simplemente ejecutamos la instrucción `break` y nos salimos del ciclo. Al final devolvemos la imagen creada mediante `return`.

- Considere la forma en que WinAmp genera sus visualizaciones, o cómo es que Excel o MediaTools realizan gráficos, o la manera en que este programa realiza su visualización. Cada uno de ellos tan sólo decide una manera distinta de asociación de las muestras a los colores y el espacio. Es sólo una asociación. Son sólo bits.

**Idea de ciencias computacionales: son sólo bits**

Los sonidos, las imágenes y el texto son sólo "bits". Tan sólo representan información. Podemos asociar de un medio a otro de cualquier forma deseada, e incluso convertir de vuelta al medio original. Lo único que tenemos que hacer es definir nuestra representación.

■

Ahora, ¿podemos regresar de nuevo? ¿Qué pasaría si tomamos una imagen como la de la figura 11.3 y la proporcionamos como entrada para una función que conozca la asociación? ¿Podríamos obtener un sonido de vuelta? Si el sonido fuera el de alguien hablando, ¿podríamos escucharlo de nuevo?

**Programa 102: escuchar una imagen**

```
def imagenASonido(imagen):
  sonido = makeEmptySoundBySeconds(10)
  indiceSonido = 0
  for p in getPixels(imagen):
    if indiceSonido == getLength(sonido):
      break
    if getRed(p) > 200:
      setSampleValueAt(sonido,indiceSonido,1000)
    elif getBlue(p) > 200:
      setSampleValueAt(sonido,indiceSonido,-1000)
    elif getGreen(p) > 200:
      setSampleValueAt(sonido,indiceSonido,0)
    indiceSonido = indiceSonido + 1
  return(sonido)
```

■

### Cómo funciona

En `imagenASonido` recibimos una imagen como entrada y devolvemos un sonido a partir de ella. Para empezar, creamos un sonido vacío de 10 segundos de longitud (`makeEmptySound-`

`BySeconds(10)`). La idea es tener un sonido grande para llenarlo, pero también es una limitación: no podemos procesar más de 10 segundos de sonido.

Por lo tanto, usamos una variable llamada `indiceSonido` para representar la siguiente muestra que llenaremos en nuestro `sonido` de destino. Usamos `p` para representar cada pixel. Si llegamos al final del sonido antes de llegar al final de los píxeles (`indiceSonido == getLength(sonido)`), ejecutamos `break` y nos detenemos.

Podríamos comprobar que el pixel `p` sea exactamente rojo (el componente rojo es 255 y los componentes verde y azul son 0), verde o azul, pero dado que usamos JPEG (un formato con pérdidas), es poco probable. Sin embargo, hay que tener en cuenta que colocamos los colores rojo, verde y azul, por lo que podemos suponer que un pixel con *mucho* rojo probablemente sea de ese color. Entonces comprobaremos si el componente rojo es mayor que 200; de ser así, asignaremos el valor 1 000 a la muestra correspondiente (en `indiceSonido`). Si tiene mucho azul, asignamos –1 000 a la muestra y si hay mucho verde le asignamos un 0. Usaremos `elif` en este ejemplo como abreviación para `else if`. Sin importar el valor que asignemos, el siguiente paso será avanzar a la siguiente muestra (`indiceSonido = indiceSonido + 1`) antes de pasar al siguiente pixel. Al final, devolvemos el sonido mediante `return(sonido)`.

Si codificamos las palabras dichas por alguien, de un sonido a una imagen, y luego convertimos el resultado de vuelta mediante `imagenASonido`, ¿cree usted que podríamos distinguir las palabras? En el capítulo 6 maximizamos el sonido y era posible distinguir las palabras, a pesar de que sólo teníamos un bit por muestra. En este caso tenemos tres valores o dos bits por muestra. ¡Mucha información! Y en definitiva, podemos distinguir las palabras con bastante claridad.

Ya vimos que podemos colocar la información del habla dentro de una imagen. ¿Cuántas imágenes en Internet tienen información del habla oculta en su interior, tal vez usando una asociación más sofisticada entre muestras y píxeles? La idea clave aquí es que la información del habla es sólo *información* que puede representarse en múltiples medios.

## 11.4 USO DE LISTAS COMO TEXTO ESTRUCTURADO PARA REPRESENTACIONES DE MEDIOS

Como dijimos antes, las listas son muy poderosas. Es muy fácil pasar de un sonido a las listas.

**Programa 103: asociar sonidos a listas**

```
def sonidoALista(sonido):
  lista = []
  for s in getSamples(sonido):
    lista = lista + [getSampleValue(s)]
  return lista

>>> lista = sonidoALista(sonido)
>>> print lista[0]
6757
>>> print lista[1]
6852
>>> print lista[0:100]
```

```
[6757, 6852, 6678, 6371, 6084, 5879, 6066, 6600,
7104, 7588, 7643, 7710, 7737, 7214, 7435, 7827,
7749, 6888, 5052, 2793, 406, -346, 80, 1356, 2347,
1609, 266, -1933, -3518, -4233, -5023, -5744,
-7394, -9255, -10421, -10605, -9692, -8786, -8198,
-8133, -8679, -9092, -9278, -9291, -9502, -9680,
-9348, -8394, -6552, -4137, -1878, -101, 866, 1540,
2459, 3340, 4343, 4821, 4676, 4211, 3731, 4359, 5653,
7176, 8411, 8569, 8131, 7167, 6150, 5204, 3951, 2482,
818, -394, -901, -784, -541, -764, -1342, -2491,
-3569, -4255, -4971, -5892, -7306, -8691, -9534,
-9429, -8289, -6811, -5386, -4454, -4079, -3841,
-3603, -3353, -3296, -3323, -3099, -2360]
```

■

Pasar de imágenes a listas es igual de sencillo; sólo hay que definir nuestra representación. ¿Qué hay sobre asociar cada pixel como sus posiciones *X* y *Y*, y después sus canales rojo, verde y azul? Tenemos que usar corchetes dobles debido a que queremos estos cinco valores como sublistas dentro de la lista grande.

**Programa 104: asociar imágenes a listas**

```
def imagenALista(imagen):
  lista = []
  for p in getPixels(imagen):
    lista = lista + [[getX(p),getY(p),getRed(p),getGreen(p),getBlue(p)]]
  return lista

>>> imagen = makePicture(pickAFile())
>>> imaglista = imagenALista(imagen)

>>> print imaglista[0:5]
[[0, 0, 168, 131, 105], [1, 0, 168, 131, 105], [2, 0, 169,
132, 106], [3, 0, 169, 132, 106], [4, 0, 170, 133, 107]]
```

■

También es fácil hacerlo al revés. Sólo tenemos que asegurarnos de que nuestras posiciones *X* y *Y* estén dentro de los límites de nuestro lienzo.

**Programa 105: asociar listas a imágenes**

```
def listaAImagen(lista):
  imagen = makePicture(getMediaPath("640x480.jpg"))
  for p in lista:
    if p[0] <= getWidth(imagen) and p[1]<=getHeight(imagen):
      setColor(getPixel(imagen,p[0],p[1]),makeColor(p[2],p[3],p[4]))
  return imagen
```

■

Podemos asegurar que esto funcionará debido a que es posible ver cómo se lleva a cabo la asociación de ambas formas y luego considerar sólo la asociación de listas a imágenes. Los números no *tienen* que provenir de una imagen. De igual forma podríamos haber asociado

los datos del clima, de un tablero de cotizaciones o de casi cualquier otra cosa con una lista de números para luego visualizarlos. Al fin de cuentas, son sólo bits...

Lo que en realidad estamos haciendo aquí es cambiar la codificación. No cambiamos la información base para nada. Las distintas codificaciones nos ofrecen diferentes capacidades.

Un matemático muy inteligente llamado Kurt Gödel usó la noción de las codificaciones para producir una de las pruebas más brillantes del siglo XX. Kurt probó el **teorema de incompletitud**, demostrando así que no cualquier sistema matemático poderoso puede probar todas las verdades matemáticas. Descubrió una asociación de enunciados matemáticos de verdad a números. Una vez que estos enunciados se convirtieron en números, él pudo demostrar que existen números que representan enunciados verdaderos que no podrían derivarse del sistema matemático. De esta forma, demostró que ningún sistema de lógica puede probar todos los enunciados verdaderos. Al cambiar su codificación obtuvo nuevas capacidades y, por ende, pudo probar algo que nadie sabía que existía.

Claude Shannon fue un ingeniero y matemático estadounidense que desarrolló la *teoría de la información*. Esta teoría describe cómo puede representarse la información en distintos medios. Al asociar una imagen con texto o sonidos, estamos aplicando la teoría de la información.

## 11.5 CÓMO OCULTAR INFORMACIÓN EN UNA IMAGEN

La *esteganografía* trata sobre ocultar información en formas que no puedan detectarse con facilidad. Si tenemos un mensaje de texto visualizado en color negro sobre una imagen blanca, podemos ocultarlo dentro de otra imagen. Para ello debemos primero cambiar todos los valores de color rojo en la imagen original a números pares. Después iteramos por los píxeles del texto que deseamos ocultar y, si el color del pixel está cerca del negro, entonces hacemos el valor de rojo impar en la imagen modificada.

**Programa 106: codificación del mensaje**

```
def codificar(imagMsj,original):
  # Suponer que imagMsj y original tienen las mismas medidas
  # Primero, hacemos todos los píxeles rojos pares
  for pxl in getPixels(original):
    # Usar el operador módulo para probar si es impar
    if (getRed(pxl) % 2) == 1:
      setRed(pxl, getRed(pxl) - 1)
  # Después en donde haya negro en imagMsj
  # hacer impar el rojo en el pixel original correspondiente
  for x in range(0,getWidth(original)):
    for y in range(0,getHeight(original)):
      pxlMsj = getPixel(imagMsj,x,y)
      pxlOrig = getPixel(original,x,y)
      if (distance(getColor(pxlMsj),black) < 100.0):
        # Es un pixel de mensaje. Hacer el valor rojo impar.
        setRed(pxlOrig, getRed(pxlOrig)+1)
```

■

Ahora podemos ocultar un mensaje en la imagen de la playa, haciendo lo siguiente:

```
>>> playa = makePicture(getMediaPath("beach.jpg"))
>>> explore(playa)
>>> msj = makePicture(getMediaPath("msg.jpg"))
>>> codificar(msj,playa)
>>> explore(playa)
>>> writePictureTo(playa,getMediaPath("ocultoplaya.png"))
```

Debemos guardar la imagen usando el formato png o bmp. No guarde la imagen usando el formato JPEG (jpg). Este formato tiene pérdidas, lo cual significa que no guarda la imagen exacta como se especificó de manera original. Descarta los detalles (como ciertos valores de color rojo específicos) que por lo general no detectaríamos, pero en este caso queremos guardar la imagen exactamente como está, para poder decodificarla después.

¿Puede detectar alguna diferencia entre la imagen original de la playa y la que contiene el mensaje oculto (figura 11.4)? Lo dudamos.

Ahora vamos a recuperar el mensaje oculto. He aquí la función para hacerlo.

**Programa 107: decodificación del mensaje**

```
def decodificar(imagCodificada):
  # Recibe una imagen codificada. Devuelve el mensaje original
  mensaje = makeEmptyPicture(getWidth(imagCodificada),getHeight(imagCodificada))
  for x in range(0,getWidth(imagCodificada)):
    for y in range(0,getHeight(imagCodificada)):
      pxlCod = getPixel(imagCodificada,x,y)
      pxlMsj = getPixel(mensaje,x,y)
      if (getRed(pxlCod) % 2) == 1:
        setColor(pxlMsj,black)
  return mensaje
```

■

**FIGURA 11.4**
La imagen original (*izquierda*) y la imagen con el mensaje oculto (*derecha*).

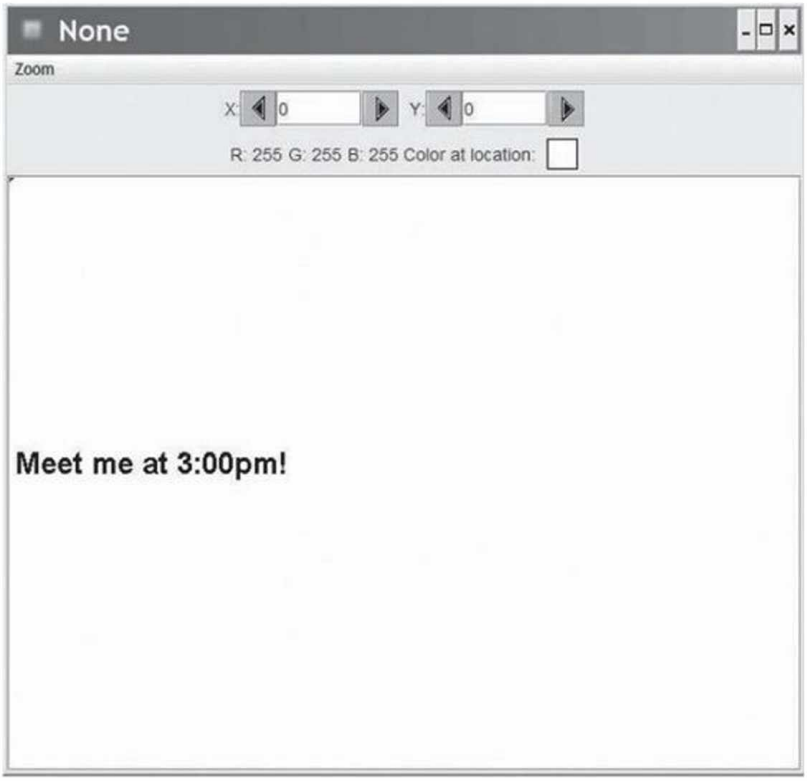



**FIGURA 11.5**
El mensaje decodificado.

Podemos decodificar el mensaje usando el siguiente código. Con él deberá poder leer el mensaje resultante (figura 11.5).

```
>>> msjOrig = decodificar(playa)
>>> explore(msjOrig)
```

## RESUMEN DE PROGRAMACIÓN
## PIEZAS GENERALES DE UN PROGRAMA

| | |
|---|---|
| `while` | Crea un ciclo. Ejecuta el cuerpo en forma iterativa mientras que la expresión lógica proporcionada sea verdadera (es decir, que no sea cero). |
| `break` | Interrumpe un ciclo de inmediato; salta hasta el final del `for` o `while`. |
| `urllib, ftplib` | Módulo para usar el acceso a URL o FTP. |
| `str` | Convierte los números (u otros objetos) a sus representaciones en cadena. |
| `float` | Convierte un número o una cadena a su representación equivalente en punto flotante. |

## PROBLEMAS

11.1 Vaya a una página que contenga mucho texto, como `http://www.cnn.com`, y use los menús de su navegador para GUARDAR el archivo con un nombre como **miPagina.html**. Edite el archivo mediante JES o con un editor como el Bloc de notas de Windows. Busque texto en la página que aparezca al momento de verla, como el texto de un encabezado o artículo. *¡Cambie el texto!* En vez de "manifestantes" provocando disturbios, cambie el texto por "estudiantes universitarios" o incluso "niños de preescolar". Ahora ABRA ese archivo en su navegador. ¡Acaba de reescribir las noticias!

11.2 Cree una función que extraiga datos de varios sitios Web y reúna todo en una página Web.

11.3 Cree una función que extraiga un sonido de un URL y cree un clip de sonido guardado en su equipo local.

11.4 Cree una función que extraiga una imagen de un URL y cree una miniatura guardada en su equipo local.

11.5 Relacione la letra de la definición con la frase apropiada como se indica a continuación (*así es, quedará una definición sin usar*).

`___ Servidor de nombre de dominio` `___ Servidor Web` `___HTTP` `___HTML`

`___ Cliente` `____ Dirección IP` `___ FTP` `___ URL`

(a) Una computadora que asocia nombres como *www.cnn.com* con sus direcciones en Internet.

(b) Un protocolo que se utiliza para mover archivos entre computadoras (por ejemplo, de su computadora personal a una computadora más grande que actúa como servidor Web).

(c) Una cadena que explica cómo (qué protocolo), en qué máquina (nombre de dominio) y en qué lugar de la máquina (ruta) es posible encontrar un archivo específico en Internet.

(d) Una computadora que ofrece archivos a través de HTTP.

(e) El protocolo sobre el cual se basa la mayor parte de Web, una forma muy simple orientada a la transmisión rápida de pequeñas piezas de información.

(f) Lo que es un navegador (como Internet Explorer) al contactar a un servidor como *google.com*.

(g) Las etiquetas que van en las páginas Web para identificar partes de la página y cómo debe aplicarse el formato.

(h) Un protocolo utilizado para transmitir correo electrónico entre computadoras.

(i) El identificador numérico de una computadora en Internet: cuatro números entre 0 y 255, como *120.32.189.12*.

11.6 Para cada uno de los siguientes enunciados, vea si puede determinar la representación en términos de bits y bytes.

(a) Las direcciones de Internet son de cuatro números, cada uno entre 0 y 255. ¿Cuántos bits hay en una dirección de Internet?

(b) En el lenguaje de programación Basic, las líneas pueden enumerarse, cada una entre 0 y 65535. ¿Cuántos bits se necesitan para representar un número de línea?

(c) El color de cada pixel tiene tres componentes: rojo, verde y azul, cada uno de los cuales puede tener un valor entre 0 y 255. ¿Cuántos bits se necesitan para representar el color de un pixel?

(d) En algunos sistemas, una cadena sólo puede tener hasta 1 024 caracteres. ¿Cuántos bits se necesitan para representar la longitud de una cadena?

11.7 ¿Qué es un servidor de nombres de dominio? ¿Qué hace?

11.8 ¿Qué son FTP, SMTP y HTTP? ¿Para qué se utiliza cada uno?

11.9 ¿Qué es hipertexto? ¿Quién lo inventó?

11.10 ¿Cuál es la diferencia entre un cliente y un servidor?

11.11 ¿En qué le ayuda saber cómo manipular texto para recopilar y crear información en Internet?

11.12 ¿Qué es Internet?

11.13 ¿Qué es un ISP? ¿Puede dar un ejemplo de uno?

11.14 Escriba una función para asociar una imagen con un sonido.

11.15 ¿Es posible ocultar una imagen de color en otra imagen? ¿Por qué sí o por qué no?

11.16 Escriba una función para traducir texto en una imagen.

11.17 Escriba una función para traducir texto en un sonido.

11.18 Escriba una función para cifrar texto, reemplazando cada letra con la letra siguiente en el alfabeto. Escriba una función para descifrar dicho mensaje también.

11.19 Escriba una función para cifrar texto, cambiando el orden de los caracteres en el mensaje. Escriba una función para descifrar dicho mensaje también.

11.20 Escriba una función para cifrar texto mediante el uso de las posiciones de las letras en algún otro documento. Escriba una función para descifrar dicho mensaje también.

## PARA PROFUNDIZAR

El libro que usa Mark para trabajar con los módulos de Python es *Python Standard Library* de Frederik Lundh [29]. Encontrará la lista de módulos de la biblioteca junto con su documentación en `http://docs.python.org/library/`.

CAPÍTULO 12

# Creación de texto para la Web

12.1 HTML: LA NOTACIÓN DE LA WEB

12.2 ESCRITURA DE PROGRAMAS PARA GENERAR HTML

12.3 BASES DE DATOS: UN LUGAR PARA ALMACENAR NUESTRO TEXTO

### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Obtener ciertas habilidades básicas con HTML.
- Generar HTML de manera automática para introducir datos, como una página de índice para un directorio de imágenes.
- Usar bases de datos para generar contenido Web.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Usar otra base de números (hexadecimal) para especificar los colores RGB.
- Establecer la diferencia entre XML y HTML.
- Explicar qué es SQL y qué tiene que ver con las bases de datos relacionales.
- Crear y usar subfunciones (funciones utilitarias).
- Demostrar un uso de las tablas hash (diccionarios).

## 12.1 HTML: LA NOTACIÓN DE LA WEB

En su mayoría, World Wide Web consiste en texto y una gran parte de éste se escribe en el lenguaje de especificación **HTML** (**lenguaje de marcado de hipertexto**). HTML se basa en **SGML** (**lenguaje de marcado estándar generalizado**), lo cual es una forma de agregar texto adicional al texto propio para identificar partes lógicas del documento: "Aquí está el título", "Éste es un encabezado" y "Éste es un simple párrafo". En un principio se suponía que HTML (al igual que SGML) sólo identificaría las partes lógicas de un documento; su *apariencia* era responsabilidad del navegador. Se *esperaba* que los documentos se vieran distintos de un navegador a otro. Pero a medida que Web evolucionó, se desarrollaron dos metas por separado: poder especificar *muchas* partes lógicas (precios, números de piezas, códigos de cotización en bolsa, temperaturas) y ser capaz de controlar el formato con mucho cuidado.

Para la primera meta surgió el **XML** (**lenguaje de marcación extensible**). Este lenguaje nos permite definir nuevas etiquetas, como `<numeropieza>7834JK</numeropieza>`. Para la segunda meta se desarrollaron cosas como las **hojas de estilo en cascada**. También se desarrolló otro lenguaje de marcación conocido como **XHTML**, que viene siendo HTML en términos del XML.

En la mayor parte de este capítulo introduciremos el XHTML, pero no vamos a diferenciarlo del HTML original. Sólo hablaremos de él como si fuera HTML.

Aquí no vamos a mostrar un tutorial completo para HTML. Hay muchos de éstos disponibles, tanto en documentos impresos como en Web, y muchos de ellos son de alta calidad. Escriba "tutorial de HTML" en su motor de búsqueda favorito y elija una de las coincidencias. Nosotros hablaremos mejor sobre algunas nociones generales de HTML y mencionaremos las etiquetas más esenciales que debe conocer.

En un lenguaje de marcación, se inserta texto en el texto original para identificar las partes. En HTML, el texto insertado (conocido como **etiquetas**) se delimita con los signos "<" y ">". Por ejemplo, `<p>` inicia un párrafo y `</p>` lo termina.

Las páginas Web tienen varias partes y éstas se anidan entre sí. La primera es doctype en la parte superior de la página y anuncia el tipo de página (es decir, si el navegador debería tratar de interpretarla como HTML, XHTML, CSS u otra cosa). Después del tipo de documento vienen un encabezado (`<head>...</head>`) y un cuerpo (`<body>...</body>`). El encabezado puede contener información como el título anidado en su interior: el título se acaba antes de que termine el encabezado. El cuerpo tiene muchas piezas anidadas en su interior, como encabezados (h1 empieza y termina antes de que termine el cuerpo) y párrafos. Todo el cuerpo y el encabezado se anidan dentro de un conjunto de etiquetas `<html>...</html>`. La figura 12.1 muestra el código fuente de una página Web simple y la figura 12.2 muestra cómo aparece la página en Internet Explorer. Pruebe esto por su cuenta. Escriba el código en JES, guárdelo con un sufijo de archivo .html y luego ábralo en un navegador Web. La única diferencia entre este archivo y cualquier página Web es que el primero se encuentra en su disco. Si estuviera en un servidor Web, sería una página Web.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transition//EN" "http://www.3org/TR/html4/loose.dtd">
<html>
</head>
<title>La página Web más simple posible</title>
</head>
<body>
<h1>Un encabezado simple</h1>
<p>Este es un párrafo en la página
Web más simple posible.</p>
</body>
</html>
```

**Sí, todo eso es el DOCTYPE**

**No importa en dónde coloca los retornos de carro ni los espacios adicionales**

**FIGURA 12.1**
Código fuente de una página simple en HTML.

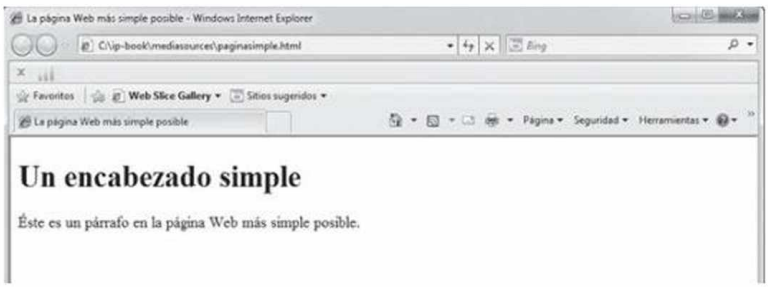



**FIGURA 12.2**
Página simple de HTML abierta en Internet Explorer. *Fuente*: captura de página de Internet Explorer 8. Copyright © 2009 por Microsoft Corporation.

**Error común: los navegadores son indulgentes pero por lo general se equivocan**

Los navegadores pueden ser muy indulgentes. Si olvida el DOCTYPE o comete errores en el HTML, el navegador literalmente adivinará lo que usted quiso decir y luego tratará de mostrarlo. No obstante, la ley de Murphy nos dice que estará *equivocado*. Si desea que su página Web se vea justo como lo desea, asegúrese de tener bien su HTML.

■

He aquí algunas de las etiquetas que debe conocer bien:

- La etiqueta `<body>` puede recibir parámetros para configurar el fondo, texto y los colores de los vínculos. Estos colores pueden ser simples, como "rojo" o "verde", o pueden ser colores RGB específicos.

  Podemos especificar colores en **hexadecimal**. Éste es un sistema numérico con base 16, mientras que el sistema numérico decimal es base 10. Los números decimales del 1 al 20 se traducen al hexadecimal en 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13 y 14. Piense en el "14" hexadecimal como uno por 16 más 4 por uno nos da un resultado de 20.

  La ventaja del sistema hexadecimal es que cada dígito corresponde a cuatro bits. Dos dígitos hexadecimales corresponden a un byte. Por ende, los tres bytes de los colores RGB son seis dígitos hexadecimales, en orden RGB. El FF0000 hexadecimal es el rojo: 255 (FF) para rojo, 0 para verde y 0 para azul. El 0000FF es azul, el 000000 es negro y el FFFFFF es blanco.
- Los encabezados se especifican mediante el uso de las etiquetas `<h1>...</h1>` a `<h6>...</h6>`. Los números menores son más prominentes.
- Hay muchas etiquetas para distintos tipos de estilos: énfasis `<em>...</em>`, cursiva `<i>...</i>`, negrita `<b>...</b>`, tipos de letra más grande `<big>...</big>` y más pequeño `<small>...</small>`, tipo de letra de máquina de escribir `<tt>...</tt>`, texto preformateado `<pre>...</pre>`, citas en bloque `<blockquote>...</blockquote>`, subíndices `<sub>...</sub>` y superíndices `<sup>...</sup>` (figura 12.3). También podemos controlar cosas como los tipos de letra y el color mediante el uso de las etiquetas `<font>...</font>`.
- Podemos insertar saltos de línea sin nuevos párrafos usando `<br />`.

**FIGURA 12.3**
"Pantalla del texto anterior" *Fuente:* Capturas de pantalla de Microsoft Internet Explorer 8.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transition//EN"
    "http://www.w3org/TR/html4/loose.dtd">
<html>

  <head>
    <title>La página Web más simple posible</title>
  </head>

  <body>

<h1>Un encabezado simple</h1>

<p>Este es un párrafo en la página Web <br/>
                          más simple posible.</p>

<img src='mediasources/flower1.jpg"/>

  </body>

</html>
```

**FIGURA 12.4**
Inserción de una imagen en una página de HTML. *Fuente:* Capturas de pantalla de Microsoft Internet Explorer 8. Copyright © 2009 por Microsoft Corporation. Reimpreso con permiso.

- Usamos la etiqueta `<image src="imagen.jpg"/>` para insertar imágenes (figura 12.4). La etiqueta `image` recibe una imagen como un parámetro `src=`. Lo que viene después es la especificación de una imagen, en una de varias formas.
  - Si es sólo un nombre de archivo (como **flower1.jpg**), entonces se asume que será una imagen en el mismo directorio que el archivo HTML que la está referenciando.
  - Si es una ruta, se asume que será una ruta que vaya *del mismo directorio que la página de HTML* hasta esa imagen. Entonces, si una página de HTML en MIS DOCUMENTOS hiciera referencia a una imagen en el directorio **mediasources**, podríamos tener una referencia a **mediasources/flower1.jpg**. Aquí podemos usar convenciones de UNIX (por ejemplo, `".."` hace referencia al directorio padre, por lo que **../imagenes/flower1.jpg** iría al directorio padre y luego bajaría a **imagenes** para obtener la imagen **flower1.jpg**).
  - También puede ser un URL completo: es posible hacer referencia a imágenes que se encuentren en otros servidores totalmente distintos.

  También es posible manipular la anchura y altura de las imágenes con opciones en la etiqueta de la imagen (por ejemplo, `<image height="100" src="flower1.jpg">` para limitar la altura a 100 píxeles) y ajustar la anchura, de modo que la imagen mantenga su relación altura:anchura. Si usa la opción `alt` puede especificar que aparezca texto si no es posible mostrar la imagen (por ejemplo, para navegadores de audio o Braille).
- Podemos usar la *etiqueta de ancla* `<a href="unlugar.html">texto de ancla</a>` para crear vínculos del texto de ancla hacia algún otro lado. En este ejemplo, `algunlado.html` es el *destino* del ancla: es hacia dónde vamos al hacer clic en ésta. El *ancla* es en donde hacemos clic. Puede ser texto, como `texto de ancla`, o puede ser una imagen. Como vemos en la figura 12.5, el destino también puede ser un URL completo.

```
<body>
<h1>Un encabezado simple</h1>
<p>Este es un párrafo en la página Web
<br/>
más simple posible.</p>
<img src="mediasources/flower1.jpg"
   alt="Una flor"/>
<p>Aquí está un vínculo a
<a href=
   "http://www.cc.gatech.edu/~mark.guz
dial/">Mark Guzdial</a>
</p>

</body>
```

**FIGURA 12.5**
Una página de HTML con un vínculo en ella. Las capturas de pantalla son de Microsoft Internet Explorer 8. Copyright © 2009 por Microsoft Corporation. Reimpreso con permiso.

Observe también en la figura 12.5 que los saltos de línea en el archivo no aparecen en el navegador. Podemos incluso tener saltos de línea en medio de una etiqueta de ancla y no afectarán la vista. Los saltos de línea que *importan* (es decir, que aparecen en la vista del navegador) se generan mediante etiquetas como `<br />` y `<p>`.

- Podemos crear listas con viñetas (*listas desordenadas*) y listas numeradas (*listas ordenadas*) si usamos las etiquetas `<ul>...</ul>` y `<ol>...</ol>`, respectivamente. Los elementos individuales se especifican usando las etiquetas `<li>...</li>`.
- Las tablas se crean mediante las etiquetas `<table>...</table>`. Las tablas se construyen a partir de filas mediante las etiquetas `<tr>...</tr>` y cada fila consiste en varios elementos de datos, los cuales se identifican con etiquetas `<td>...</td>` (figura 12.6). Las tablas contienen filas y éstas contienen elementos de datos.

Hay *mucho* más sobre el HTML, como los marcos (tener subventanas dentro de la ventana de la página de HTML), divisiones (`<div />`), reglas horizontales (`<hr />`), applets y JavaScript. Los elementos antes mencionados son los más imprescindibles para comprender el resto de este capítulo.

```
<table border="5">
<tr><td>Columna
  1</td><td>Columna
  2</td></tr>
<tr><td>Elemento en columna
  1</td><td>Elemento en
  columna 2</td></tr>
</table>
```

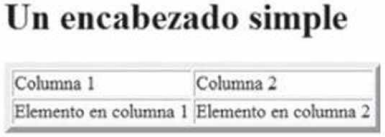



**FIGURA 12.6**
Inserción de una tabla en una página HTML.

## 12.2 ESCRITURA DE PROGRAMAS PARA GENERAR HTML

En sí, HTML no es un lenguaje de programación ya que no puede especificar ciclos, condicionales, variables, tipos de datos ni nada de lo que hemos aprendido sobre cómo especificar el proceso. HTML se usa para describir la estructura y no el proceso.

Habiendo dicho esto, es muy fácil escribir programas para generar HTML. Las múltiples formas que tiene Python para colocar las cadenas entre comillas son *bastante* útiles para este propósito.

**Programa 108: generación de una página simple de HTML**

```
def crearPagina():
  archivo=open("generado.html","wt")
  archivo.write("""<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transition//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head> <title>La página Web más simple posible</title>
</head>
<body>
<h1>Un encabezado simple</h1>
<p>Algo de texto simple.</p>
</body>
</html>""")
  archivo.close()
```

■

Esto funciona, pero es en verdad aburrido. ¿Para qué escribir un programa si puede hacerlo con un editor de texto? Escribimos programas para tener facilidad de reproducir un proceso, de comunicarlo y de personalizarlo. Ahora vamos a diseñar un creador de páginas de inicio.

**Programa 109: nuestro primer creador de páginas de inicio**

```
def crearPaginaInicio(nombre, interes):
  archivo=open("paginainicio.html","wt")
  archivo.write("""<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transition//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Página de inicio de """+nombre+"""</title>
</head>
<body>
<h1>Bienvenido a la página de inicio de """+nombre+"""</h1>
<p>¡Hola! Soy """+nombre+""". Esta es mi página de inicio.
Este es mi interés: """+interes+"""</p>
</body>
</html>""")
  archivo.close()
```

■

Así, al ejecutar crear `PaginaInicio("Barb", "caballos")` se creará la siguiente página:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/
  loose.dtd">
<html>
<head>
<title>Página de inicio de Barb</title>
</head>
<body>
<h1>Bienvenido a la página de inicio de Barb</h1>
<p>¡Hola! Soy Barb. Esta es mi página de inicio.
Este es mi interés: caballos</p>
</body>
</html>
```

**Tip de depuración: escribir primero el HTML**

Los programas para generar HTML pueden ser confusos. Antes de que empiece a tratar de escribir uno, escriba el HTML. Cree una muestra de cómo desea que se vea el HTML Asegúrese de que funcione en su navegador. *Después* escriba la función de Python que genere ese tipo de HTML.

■

No obstante, la modificación de este programa es un proceso doloroso. Hay tanto detalle en el HTML y es difícil lograr escribir correctamente todas las comillas. Estamos mejor si usamos *subfunciones* para dividir el programa en piezas que sean más fáciles de manipular. De nuevo, esto es un ejemplo del uso de la **abstracción procedural**. He aquí una versión del programa en donde confinamos en el área superior las partes que cambiaremos con más frecuencia.

**Programa 110: creador de páginas de inicio mejorado**

```
def crearPaginaInicio(nombre, interes):
  archivo=open("paginainicio.html","wt")
  archivo.write(doctype())
  archivo.write(title("Página de inicio de"+nombre))
  archivo.write(body("""
<h1>Bienvenido a la página de inicio de """+nombre+"""</h1> <p>¡Hola! Soy
"""+nombre+""". Esta es mi página de inicio. Este es mi interés:
"""+interes+"""</p>"""))
  archivo.close()

def doctype():
  return '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">'

def title(cadenatitulo):
  return "<html><head><title>"+cadenatitulo+"</title></head>"

def body(cadenacuerpo):
  return "<body>"+cadenacuerpo+"</body></html>"
```

■

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
   HTML 4.01 Transition//EN"
   "http://www.w3.org/TR/html4/loose.dtd"
   ><html><head><title>Muestras de
   C:/ip-book/mediasources/imagenes</title>
   </head><body><h1>Muestras de
   c:/ip-book/mediasources/imagenes
   </h1>
<p>Nombre del archivo: anthony.jpg<image
   src="anthony.jpg"height="100"/></p>
<p>Nombre del archivo: arch.jpg<image
   src="arch.jpg"height="100"/></p>
<p>Nombre del archivo: arthurs-seat.jpg<image
   src="arthurs-seat.jpg"height="100"/></p>
<p>Nombre del archivo: barbara.jpg<image
   src="barbara.jpg"height="100"/></p>
<p>Nombre del archivo: beach.jpg<image
   src="beach.jpg"height="100"/></p>
<p>Nombre del archivo: bigben.jpg<image
   src="bigben.jpg"height="100"/></p>
</body></html>
```

**FIGURA 12.7**
Creación de una página de miniaturas. *Fuente:* Captura de pantalla de Internet Explorer 8. Copyright © 2009 por Microsoft Corporation. Reimpreso con permiso.

Podemos obtener contenido para nuestras páginas Web de cualquier lugar que nos guste. A continuación veremos un programa que puede extraer información de un directorio proporcionado como entrada, para luego generar una página índice de esas imágenes (figura 12.7). No vamos a listar a `doctype()` ni a las demás *funciones utilitarias* en este ejemplo; sólo nos enfocaremos en la parte que nos importa. Y así es como deberíamos considerarlo: sólo la parte que nos importa; escribimos `doctype()` una vez y luego nos olvidamos de ello.

**Programa 111: generación de una página de miniaturas**

```
import os

def crearPaginaMuestra(directorio):
  archivoMuestras.open(directorio+"//muestras.html","wt")
  archivoMuestras.write(doctype())
  archivoMuestras.write(title("Muestras de "+directorio))
  # Ahora, vamos a crear la cadena que formará el cuerpo.
  muestras="<h1>Muestras de "+directorio+" </h1>"
  for archivo in os.listdir(directorio):
    if archivo.endswith(".jpg"):
      muestras=muestras+"<p>Nombre del archivo: "+archivo
      muestras=muestras+'<image src="'+archivo+'"height="100"/></p>'
    archivoMuestras.write(body(muestras))
    archivoMuestras.close()
```

■

Podemos extraer información de la Web para crear nuevo contenido Web. Así es como funcionan los sitios como *Google News*.[1] He aquí una versión de nuestro creador de páginas de inicio que incluye nuestros intereses de Facebook. Podría usarlo de la siguiente manera: `crearPaginaInicio("Mark Guzdial", "mark.guzdial")`. (También necesitará las funciones `doctype()`, `title()` y `body()` que vimos antes).

[1] `http://news.google.com`

**Programa 112: generación de una página de inicio con información de intereses de Facebook**

```
import urllib

def crearPaginaInicio(nombre, fbNombre):
  archivo=open("C:/Users/MarkGuzdial/Documents/paginainicio.html","wt")
  archivo.write(doctype())
  archivo.write(title("Página de inicio de"+nombre))
  archivo.write(body("""
<h1>Bienvenido a la página de inicio de """+nombre+"""</h1> <p>¡Hola! Soy
"""+nombre+""". Esta es mi página de inicio. Estos son mis intereses:
"""+fbActividadesIntereses(fbNombre)+"""</p>"""))
  archivo.close()

def fbActividadesIntereses(nombre):
   import urllib  # Podría ir arriba también
   intereses = ""
   conexion = urllib.urlopen("http://www.facebook.com/
   "+nombre+"?ref=pb")
   fb = conexion.read()
   conexion.close()
   actInicio = fb.find("activities_and_interests")
   ubicPagina = actInicio   # Lugar de inicio para buscar páginas
  if actInicio <> -1: # ¿Encontramos Actividades e intereses?
    ubicPagina = fb.find("http://www.facebook.com/pages/",ubicPag)
    while (ubicPagina <> -1):
      interesInicio = ubicPagina+30  # Para "http://www.facebook.com/
      pages/"
       interesFin = fb.find("/",interesInicio+1)
       interes = fb[interesInicio:interesFin]
       if not ("footer" in interes):
          intereses = intereses+fb[interesInicio:interesFin]+","
          ubicPagina = fb.find("http//www.facebook.com/pages/",
          ubicPagina+1)
       else:
          ubicPagina = -1
   return intereses[:-1]
```

■

¿Recuerda el generador de enunciados aleatorios? También podemos agregarlo.

**Programa 113: generador de páginas de inicio con enunciados aleatorios**

```
import urllib
import random

def crearPaginaInicio(nombre, fbNombre):
  archivo=open("C:/Users/MarkGuzdial/Documents/paginainicio.html","wt")
  archivo.write(doctype())
  archivo.write(title("Página de inicio de"+nombre))
  archivo.write(body("""
```

```
<h1>Bienvenido a la página de inicio de """+nombre+"""</h1> <p>¡Hola! Soy
"""+nombre+""". Esta es mi página de inicio. Estos son mis intereses:
"""+fbActividadesIntereses(fbNombre)+"""</p><p>Reflexión aleatoria
del día:
"""+enunciado()+"</p>"))
  archivo.close()

def enunciado():
  sustantivos = ["Mark", "Alicia", "Maria", "Latrice", "Jose",
  "Corey", "Teshaun"]
  verbos = ["corre", "patina", "canta", "brinca", "salta",¬
  "trepa","discute", "ríe"]
  frases = ["en un árbol", "sobre un tronco", "muy fuerte",¬
  "alrededor del arbusto", "mientras lee el periódico"]
  frases = frases + ["muy mal", "mientras patina",¬
  "en vez de calificar", "mientras escribe en Internet"]
  return random.choice(sustantivos)+ " "+random.choice(verbos)+ ¬
  " "+random.choice(frases)+"."

def fbActividadesIntereses(nombre):
  import urllib  # Podría ir arriba también
  intereses = ""
  conexion = urllib.urlopen("http://www.facebook.com/
  "+nombre+"?ref=pb")
  fb = conexion.read()
  conexion.close()
  actInicio = fb.find("activities_and_interests")
  ubicPagina = actInicio   # Lugar de inicio para buscar páginas
  if actInicio <> -1: # ¿Encontramos Actividades e intereses?
    ubicPagina = fb.find("http://www.facebook.com/pages/",ubicPag)
    while (ubicPagina <> -1):
      interesInicio = ubicPagina+30  # Para "http://www.facebook.com/
      pages/"
      interesFin = fb.find("/",interesInicio+1)
      interes = fb[interesInicio:interesFin]
      if not ("footer" in interes):
        intereses = intereses+fb[interesInicio:interesFin]+","
        ubicPagina = fb.find("http//www.facebook.com/pages/",
        ubicPagina+1)
      else:
        ubicPagina = -1
  return intereses[:-1]
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un solo comando en Python no puede dividirse en varias líneas.

■

## Cómo funciona

Daremos un recorrido por este extenso ejemplo.

- Vamos a necesitar a `urllib` y a `random` en esta función, por lo que importamos estas bibliotecas mediante `import` en la parte superior de nuestra área del programa.

- Nuestra función principal es `crearPaginaInicio`. La invocamos con un nombre y un nombre de Facebook para buscar los intereses.
- En la parte superior de `crearPaginaInicio` abrimos el archivo de HTML que vamos a escribir; después escribimos el `doctype` usando la función utilitaria (no se muestra aquí, pero tendría que estar en el área del programa). Escribimos el título (mediante la función `title`) con el `nombre` de entrada insertado y luego el cuerpo (mediante la función `body`).
- La entrada para la función `body` es una cadena larga; es en su construcción en donde invocamos a `fbActividadesIntereses()` y a `enunciado()`. Usamos comillas triples para poder insertar retornos en donde se necesiten. Escribimos el `nombre` y los valores de retorno de las funciones, concatenando todo en medio de la cadena de HTML.
- Invocamos a `fbActividadesIntereses()` a la mitad del proceso por medio de la cadena del cuerpo. Pasamos el nombre de Facebook (`fbNombre`) de la entrada a `crearPaginaInicio`.
- Invocamos a `enunciado()` como nuestra `"Reflexión aleatoria del día"`. Esta versión de enunciado es un poco distinta, en cuanto a que usa `return` en vez de `print` para poder usar la salida en nuestra cadena del cuerpo de HTML.
- Por último, de vuelta en `crearPaginaInicio` cerramos el archivo de HTML y terminamos.

¿De dónde cree que los grandes sitios Web obtienen toda su información? Hay *tantas* páginas en esos sitios Web. ¿De dónde lo obtienen todo? ¿En dónde lo almacenan todo?

## 12.3 BASES DE DATOS: UN LUGAR PARA ALMACENAR NUESTRO TEXTO

Los sitios Web extensos usan **bases de datos** para almacenar su texto y demás información. Los sitios como EBay.com, Amazon.com y CNN.com tienen grandes bases de datos con mucha información en ellas. Las páginas en estos sitios no las genera alguien que escribe la información. En vez de ello hay programas que recorren la base de datos, recopilan toda la información y generan páginas de HTML. Pueden hacer esto en forma programada, para seguir actualizando la página (consulte la información de "última hora" en CNN.com o en Google News).

¿Por qué usar bases de datos en vez de simples archivos de texto? Hay cuatro razones:

- Las bases de datos son rápidas. Almacenan *índices* que llevan el registro de la ubicación de la información clave (como apellido o número de ID) en un archivo, para que podamos encontrar "Guzdial" de inmediato. Los archivos se indexan con base en el nombre de archivo, pero no con el contenido *dentro* del mismo.
- Las bases de datos son estandarizadas. Puede acceder a bases de datos de Microsoft Access, Informix, Oracle, Sybase y MySQL desde cualquier variedad de herramientas o lenguajes.
- Las bases de datos pueden ser **distribuidas**. Una gran cantidad de usuarios en distintas computadoras conectadas a través de una red pueden colocar información en una base de datos y extraer información de ella.
- Las bases de datos almacenan **relaciones**. Cuando usamos listas para representar píxeles, tuvimos que mantener en nuestra mente el significado de cada número. Las bases de datos almacenan nombres para *campos* de datos. Cuando una base de datos sabe qué campos son importantes (por ejemplo, en cuáles se van a realizar búsquedas con más frecuencia), puede indexarse con base en esos campos.

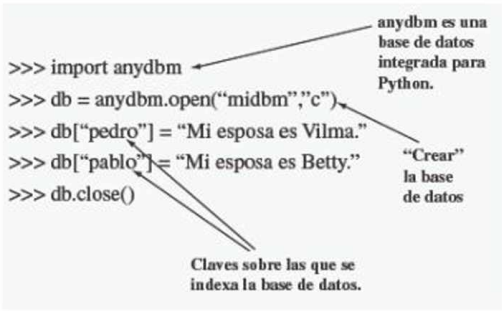



**FIGURA 12.8**
Uso de la base de datos simple.

Python cuenta con soporte integrado para varias bases de datos distintas y provee un soporte general para cualquier tipo de base de datos, conocido como `anydbm` (figura 12.8). Las **claves** van entre corchetes y son los campos por medio de los cuales se obtiene el acceso más rápido. Tanto las claves como los valores sólo pueden ser cadenas si se usa `anydbm`. Ahora veamos un ejemplo de cómo recuperar información de `anydbm`.

```
>>> db = anydbm.open("midbm","r")
>>> print db.keys()
['pablo', 'pedro']
>>> print db['pablo']
Mi esposa es Betty.
>>> for k in db.keys():
...         print db[k]
...
Mi esposa es Betty.
Mi esposa es Vilma.
>>> db.close()
```

Hay otra base de datos estándar en Python llamada `shelve`, la cual nos permite colocar cadenas, listas, números o casi cualquier otra cosa en el valor. La base de datos `shelve` es útil debido a que es un estándar de Python: siempre está disponible. Sin embargo, *no* es una base de datos relacional.

```
>>> import shelve
>>> db=shelve.open("mishelf","c")
>>> db["uno"]=["Esta es",["una","lista"]]
>>> db["dos"]=12
>>> db.close()
>>> db=shelve.open("mishelf","r")
>>> print db.keys()
['uno', 'dos']
>>> print db['uno']
['Esta es', ['una', 'lista']]
>>> print db['dos']
12
```

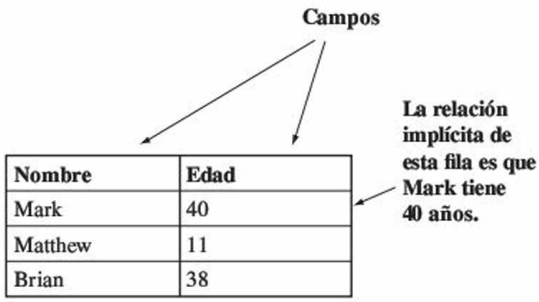
### 12.3.1 Bases de datos relacionales

La mayoría de las bases de datos modernas son **bases de datos relacionales**. En las bases de datos relacionales, la información se almacena en tablas (figura 12.9). Las columnas en una tabla relacional tienen nombres y se asume que las filas de datos están relacionadas.

Las relaciones complejas se almacenan a través de varias tablas. Digamos que tiene un grupo de imágenes de estudiantes y desea llevar la cuenta de los estudiantes y en cuáles imágenes aparecen: en donde haya más de un estudiante en una imagen específica. Podría registrar una estructura así en una colección de tablas para registrar a los estudiantes y sus números de identificación, las imágenes y los ID de imagen, y después la asociación entre ID de estudiante e ID de imagen, como en la figura 12.10.

¿Cómo usaríamos tablas como la de la figura 12.10 para averiguar en qué imagen aparece Brittany? Empezamos por buscar el ID de Brittany en la tabla de estudiantes, luego buscamos el ID de la imagen en la tabla imagen-estudiante, luego buscamos el nombre de la imagen en la tabla de imágenes, lo que produce el resultado **Class1.jpg**. ¿Qué hay sobre averiguar quién está en esa imagen? Buscamos el ID en la tabla de imágenes, luego buscamos cuáles ID de los estudiantes están relacionadas con ese ID de imagen y después buscamos los nombres de los estudiantes.

Este uso de múltiples tablas para responder a una **consulta** (solicitar información a una base de datos) se conoce como **unión**. Las uniones de bases de datos funcionan mejor si las tablas se mantienen simples, con sólo una relación por fila.




| Nombre | Edad |
|---|---|
| Mark | 40 |
| Matthew | 11 |
| Brian | 38 |

**FIGURA 12.9**
Ejemplo de una tabla relacional.

| Imagen | IDImagen |
|---|---|
| Class1.jpg | P1 |
| Class2.jpg | P2 |

| NombreEstudiante | IDEstudiante |
|---|---|
| Katie | S1 |
| Brittany | S2 |
| Carrie | S3 |

| IDImagen | IDEstudiante |
|---|---|
| P1 | S1 |
| P1 | S2 |
| P2 | S3 |

**FIGURA 12.10**
Representación de relaciones más complejas entre varias tablas.

### 12.3.2 Ejemplo de una base de datos relacional mediante el uso de tablas hash

Para explicar las ideas de las bases de datos relacionales, en esta sección construiremos una base de datos relacional mediante el uso de una estructura más simple en Python. De esta forma podemos describir cómo funcionan algunas ideas de bases de datos, como las uniones. Esta sección es opcional.

Podemos usar una estructura conocida como **tabla hash** o **diccionario** (en otros lenguajes se conocen como *arreglos asociativos*) para crear filas para las tablas relacionales de las bases de datos mediante el uso de `shelve`. Las tablas hash nos permiten relacionar claves y valores como las bases de datos, pero sólo en la memoria.

```
>>> fila={'NombreEstudiante':'Katie','IDEstudiante':'S1'}
>>> print fila
{'NombreEstudiante': 'Katie', 'IDEstudiante': 'S1'}
>>> print fila['IDEstudiante']
S1
>>> print fila['NombreEstudiante']
Katie
```

Además de definir toda la tabla hash de una sola vez, podemos llenarla por fracciones.

```
>>> imagenFila = {}
>>> imagenFila['Imagen']='Clase1.jpg'
>>> imagenFila['IDImagen']='P1'
>>> print imagenFila
{'Imagen': 'Clase1.jpg', 'IDImagen':'P1' }
>>> print imagenFila['Imagen']
Clase1.jpg
```

Ahora podemos crear una base de datos relacional, almacenando cada tabla en una base de datos `shelve` separada y representando cada fila como una tabla hash.

**Programa 114: creación de una base de datos relacional mediante el uso de `shelve`**

```
import shelve
def crearBasesDatos():
  # Crea la base de datos de estudiantes
  estudiantes=shelve.open("estudiantes.db","c")
  fila = {'NombreEstudiante':'Katie','IDEstudiante':'S1'}
  estudiantes['S1']=fila
  fila = {'NombreEstudiante':'Brittany','IDEstudiante':'S2'}
  estudiantes['S2']=fila
  fila = {'NombreEstudiante':'Carrie','IDEstudiante':'S3'}
  estudiantes['S3']=fila
  estudiantes.close()
  # Crear base de datos de imágenes
  imagenes=shelve.open("imagenes.db","c")
  fila = {'Imagen':'Clase1.jpg','IDImagen':'P1'}
  imagenes['P1']=fila
  fila = {'Imagen':'Clase2.jpg','IDImagen':'P2'}
  imagenes['P2']=fila
  imagenes.close()
  #Crear base de datos imagen-estudiante
  imagenes=shelve.open("imagen-estudiantes.db","c")
```

```
fila = {'IDImagen':'P1','IDEstudiante':'S1'}
imagenes['P1S1']=fila
fila = {'IDImagen':'P1','IDEstudiante':'S2'}
imagenes['P1S2']=fila
fila = {'IDImagen':'P2','IDEstudiante':'S3'}
imagenes['P2S3']=fila
imagenes.close()
```

■

### Cómo funciona

En sí, la función `crearBasesDatos` crea tres tablas de bases de datos distintas.

- La primera es **estudiantes.db**. Creamos diccionarios (tablas hash) que representan a Katie, quien es "S1"; luego almacenamos eso en la base de datos `estudiantes` y usamos el ID de estudiante "S1" como índice. Hacemos lo mismo para Brittany y Carrie: es correcto usar la misma variable `fila` para cada una de ellas, ya que sólo creamos la tabla hash y luego la almacenamos en la base de datos.
- A continuación creamos la base de datos **imagenes.db**. Establecemos la relación entre la "imagen" **Clase1.jpg** y su ID "P1". Luego almacenamos eso en la base de datos `imagenes`. Repetimos el proceso para la imagen **Clase2.jpg**. ¿Acaso podríamos haber usado una variable `basedatos` para cada base de datos? Como sólo abrimos una a la vez y luego la cerramos antes de la siguiente, sí podríamos hacerlo. Sólo que hubiera sido menos legible.
- Por último, abrimos y llenamos la base de datos **imagen-estudiantes.db**. Creamos filas para relacionar los ID de las imágenes con los ID de los estudiantes, las almacenamos en la base de datos y luego la cerramos.

Si usamos las bases de datos que acabamos de crear podemos realizar una *unión*. Obviamente la idea es realizar esta búsqueda un tiempo después de crear la base de datos, tal vez después de agregarle muchas entradas más (si sólo tuviéramos dos imágenes y tres estudiantes en la base de datos, sería un ejercicio bastante tonto en programación). Tenemos que iterar a través de los datos para encontrar los valores que necesitamos hacer coincidir entre las bases de datos.

**Programa 115: realización de una unión con nuestra base de datos `shelve`**

```
def quienEstaEnClase1():
  # Obtener el IDImagen
  imagenes=shelve.open("imagenes.db","r")
  for clave in imagenes.keys():
    fila = imagenes[clave]
    if fila['Imagen'] == 'Clase1.jpg':
      id = fila['IDImagen']
  imagenes.close()
  # Obtener los ID de los estudiantes
  listaestudiantes=[]
  imagenes=shelve.open("imagen-estudiantes.db","c")
```

```
for clave in imagenes.keys():
  fila = imagenes[clave]
  if fila['IDImagen']==id:
    listaestudiantes.append(fila['IDEstudiante'])
imagenes.close()
print "Buscamos a:",listaestudiantes
# Obtener los nombres de los estudiantes
estudiantes = shelve.open("estudiantes.db","r")
for clave in estudiantes.keys():
  fila = estudiantes[clave]
  if fila['IDEstudiante'] in listaestudiantes:
    print fila['NombreEstudiante'],"está en la imagen"
estudiantes.close()
```

■

### Cómo funciona

Cada una de las partes de la unión requiere una iteración diferente por nuestra base de datos.

- Primero abrimos **imagenes.db** y llamamos a esta variable `imagenes`. Iteramos por todas las claves, obtenemos cada `fila` (tabla hash) y verificamos si la entrada `'Imagen'` es **Clase1.jpg**. Cuando la encontramos, almacenamos el ID de la imagen en `id`. A continuación cerramos la base de datos debido a que terminamos de usarla.
- Luego abrimos **imagen-estudiantes.db** en `imagenes` (no hay problema si volvemos a usar el nombre). Sabemos que puede haber más de un estudiante en una imagen, por lo que creamos una lista para almacenar todos los ID de estudiantes que encontremos: `listaestudiantes`. Para cada clave en `imagenes` buscamos las entradas `'IDImagen'` en la tabla hash que coincidan con nuestra `id`. Si encontramos una, anexamos la parte `'IDEstudiante'` de la tabla hash a nuestra `listaestudiantes`.
- Por último, abrimos la base de datos **estudiantes.db** en `estudiantes`. Iteramos por todas las claves y obtenemos la `fila` de la tabla hash. Aquí usamos una característica ingeniosa de las listas que no hemos visto antes: preguntamos si una cadena está en (`in`) la lista `listaestudiantes`. Si el `'IDEstudiante'` dado es uno de los que estamos buscando (en `listaestudiantes`), entonces imprimimos el `'NombreEstudiante'` correspondiente de la tabla hash fila. Por último cerramos la base de datos `estudiantes` para limpiar todo.

Al ejecutar este código obtenemos lo siguiente:

```
>>> quienEstaEnClase1()
Buscamos a: ['S2', 'S1']
Brittany está en la imagen
Katie está en la imagen
```

## 12.3.3 Trabajar con SQL

Con las bases de datos reales no tenemos que realizar iteraciones para las uniones. Lo común es usar **SQL** (**Lenguaje de consulta estructurado**) para manipular y consultar las bases de datos. En realidad hay varios lenguajes en la familia SQL, pero aquí no vamos a marcar distinciones. SQL es un lenguaje de programación extenso y complejo, por lo que ni siquiera vamos a intentar cubrirlo todo. Pero vamos a ver lo suficiente de este lenguaje como para que usted tenga una buena idea de lo que es y cómo se usa.

Podemos usar SQL con muchas bases de datos diferentes, incluyendo Microsoft Access. Python puede comunicarse con casi cualquiera de ellas en la forma que usaremos aquí. También existen bases de datos disponibles sin costo, como **MySQL**, que usan SQL y pueden controlarse desde Python. Si desea jugar con los ejemplos que le mostraremos aquí, puede instalar *MySQL* de `http://www.mysql.com` en JES. Necesita instalar MySQL y descargar el *archivo JAR* que permite a Java acceder a MySQL; luego debe colocar ese archivo en su carpeta **JythonLib**.

Para manipular una base de datos de MySQL desde JES, hay que crear una *conexión* que nos proporcione un *cursor* para manipularlo mediante SQL. Al igual que un cursor en pantalla registra la posición en donde escribimos o copiamos (en qué línea y en qué caracteres), el cursor de la base de datos lleva el registro de dónde nos encontramos dentro de esta. Usamos MySQL desde JES con una función como la que indicamos a continuación, para poder ejecutar `con = obtenerConexion()`.

**Programa 116: obtener una conexión a MySQL desde Jython**

Todos estos detalles pueden ser difíciles de recordar, por lo que los ocultamos en una función y sólo tenemos que usar `con = obtenerConexion()` con el siguiente código en el área del programa.

```
from com.ziclix.python.sql import zxJDBC

def obtenerConexion():
  db = zxJDBC.connect("jdbc:mysql://localhost/test", "root",None, "com.mysql.
jdbc.Driver")
  con = db.cursor()
  return con
```

■

Para ejecutar comandos de SQL desde aquí, usamos un método **execute** en la conexión. El método `execute` recibe una cadena como entrada, que consta de comandos de SQL. He aquí un ejemplo:

```
con.execute("create table Persona (name VARCHAR(50), edad INT)")
```

Ahora veamos una pequeña muestra de SQL:

- Para crear una tabla en SQL, usamos `create table nombretabla (nombrecolumna tipodatos,...)`. Así, en el ejemplo anterior creamos una tabla llamada *Persona* con dos columnas: un nombre con un número variable de hasta 50 caracteres y una edad representada por un entero. Los otros tipos de datos son: numéricos, punto flotante, fecha, hora, año y texto.
- Para insertar datos en SQL usamos `insert into nombretabla values (valorcolumna1, valorcolumna2, ...)`. Aquí es donde nuestros tipos múltiples de comillas resultan ser de utilidad.

```
con.execute('insert into Persona values
  ("Mark",40)')
```

- Piense que, al igual que puede seleccionar de una manera literal las filas que desea en una tabla, en un procesador de palabras, en una hoja de cálculo, puede elegir los datos de una base de datos. Algunos ejemplos de comandos de selección en SQL son:

```
Select * from Persona
Select nombre, edad from Persona
Select * from Persona where edad>40
Select nombre, edad from Persona where edad>40
```

Podemos hacer todo esto desde Python. Nuestra conexión tiene una **variable de instancia** (como un método, pero que sólo los objetos de ese tipo conocen) llamada `rowcount`, la cual nos indica el número de filas seleccionadas. El método `fetchone()` nos proporciona la siguiente fila seleccionada, como una **tupla** (considere este término como un tipo especial de lista: podemos usarla justo como una lista para la mayoría de los propósitos).

```
>>> con.execute("select nombre,edad from Persona")
>>> print con.rowcount
3
>>> print con.fetchone()
('Mark',40)
>>> print con.fetchone()
('Barb',41)
>>> print con.fetchone()
('Brian',36)
```

También podemos seleccionar mediante una condición.

**Programa 117: seleccionar y mostrar datos mediante una selección condicional**

```
def mostrarUnasPersonas(con, condicion):
  con.execute("select nombre, edad from Persona "+condicion)
  for i in range(0,con.rowcount):
    resultados=con.fetchone()
    print resultados[0]+" tiene "+str(resultados[1])+" años"
```

■

## Cómo funciona

La función `mostrarUnasPersonas` recibe la entrada de una conexión de base de datos llamada `con` junto con una condición, lo cual tiene como objetivo representar una cadena que contiene una cláusula `where` de SQL. Después pedimos a la conexión que ejecute (`execute`) un comando `select` de SQL. Concatenamos la `condicion` al final de nuestro comando `select`. Después tenemos un ciclo para cada una de las filas devueltas de la selección (usamos `con.rowcount` para obtener ese número de filas). Usamos `fetchone` para obtener cada respuesta e imprimirla. Observe el uso del indexado de lista normal en la tupla devuelta.

He aquí un ejemplo del uso de `mostrarUnasPersonas`:

```
>>> mostrarUnasPersonas(con,"where edad >= 40")
Mark tiene 40 años
Barb tiene 41 años
```

Ahora podemos pensar cómo realizar una de nuestras uniones mediante el uso de una selección condicional. El siguiente fragmento de código dice: "Devuelve una imagen y el nombre de un estudiante de estas tres tablas, en donde el nombre del estudiante sea 'Brittany', en donde el ID del estudiante sea igual que el ID del estudiante en la tabla de ID de estudiantes-imágenes y el ID de imagen de la tabla de ID sea igual que el ID en la tabla de imágenes".

```
Select
   p.imagen,
   s.nombreEstudiante
from
   Estudiantes as s,
   ID as i,
   Imagenes as p
where
   (s.nombreEstudiante="Brittany") and
   (s.IDestudiante=i.IDestudiante) and
   (i.IDimagen=p.IDimagen)
```

### 12.3.4 Uso de una base de datos para crear páginas Web

Ahora regresemos a nuestro creador de página Web. Podemos almacenar información en nuestra base de datos, recuperarla y luego colocarla en nuestra página Web; justo igual que Amazon, CNN y eBay.

```
>>> import anydbm
>>> db=anydbm.open("C:/Users/MarkGuzdial/Documents/news","c")
>>> db["encabezado"]="Katie cumple 8"
>>> db["historia"]="""Nuestra hija Katie cumplió 8 años
    ayer. Tuvo una gran fiesta de cumpleaños. Sus abuelos
    asistieron. El fin de semana anterior la visitaron tres
    de sus amigas para una pijamada y luego una corrida
    matutina a Dave And Buster's."""
>>> db.close()
```

Al ejecutar el programa siguiente con `crearPaginaInicio("Mark Guzdial", "mark.guzdial")`, obtenemos una página Web como la de la figura 12.11.

**Programa 118: creación de una página Web con contenido de una base de datos**

```
def crearPaginaInicio(nombre, fbNombre):
  archivo=open("C:/Users/MarkGuzdial/Documents/paginainicio.html","wt")
  archivo.write(doctype())
  archivo.write(title("Página de inicio de"+nombre))
  # Importar el contenido de la base de datos
  db=anydbm.open("C:/Users/MarkGuzdial/Documents/news","r")
  archivo.write(body("""
<h1>Bienvenido a la página de inicio de """+nombre+"""</h1> <p>¡Hola! Soy
"""+nombre+""". Esta es mi página de inicio. Estos son mis intereses: """+fbActi
vidadesIntereses(fbNombre)+"""</p>
<p>Reflexión aleatoria del día:
"""+enunciado()+"""</p>
<h2>Noticias recientes:
"""+db["encabezado"]+"""</h2>
<p>"""+db["historia"]+"</p>"))
  archivo.close()
  db.close()

# El resto, como enunciado(), de los ejemplos anteriores
```

■

**FIGURA 12.11**
Salida de ejemplo del generador de páginas de inicio, extrayendo información de la Web y de una base de datos.

Ahora podemos deducir cómo funciona un sitio Web grande como CNN.com. Los reporteros introducen sus historias en una base de datos que los distribuye por todo el mundo. Los editores (también distribuidos o todos en un solo lugar) recuperan las historias de la base de datos, las actualizan y luego las vuelven a guardar. El programa de generación de páginas Web se ejecuta con regularidad (tal vez con más frecuencia cuando surge una historia muy popular), recopila las historias y genera el HTML. "¡Listo! ¡Ya tenemos un sitio Web extenso!". Las bases de datos son realmente imprescindibles para la ejecución de los sitios Web grandes.

## PROBLEMAS

12.1 ¿Cuáles son las diferencias entre SGML, HTML, XML y XHTML?

12.2 Convierta los siguientes números hexadecimales a decimal: 2A3, 321, 16, 24, F3.

12.3 Convierta los siguientes números decimales a hexadecimal: 113, 64, 129, 72, 3.

12.4 Convierta los siguientes colores a hexadecimal: gris, amarillo, rosa, naranja y magenta. Puede obtener los valores rojo, verde y azul para cada color mediante la instrucción `print color`.

12.5 Escriba una función para crear una página de inicio simple con su nombre, su imagen y una tabla con los títulos de sus cursos y nombres de los maestros.

12.6 Escriba una función para leer y devolver el encabezado actual de `http://www.cnn.com`. Modifique la función que crea una página de inicio para usar esta función.

12.7 Escriba una función para leer y devolver el estado de la página de Facebook de uno de sus amigos. Modifique la función que crea una página de inicio para usar esta función y mostrar los mensajes de estado de las páginas de inicio de Facebook de cinco de sus amigos.

12.8 Use una tabla hash para almacenar abreviaciones de mensajes de texto y sus definiciones. Por ejemplo, "tmb" significa "también". Use esta tabla hash para decodificar un mensaje de texto.

12.9 Si tenemos una base de datos relacional con una tabla de personas que contenga una identificación, nombre y edad, ¿qué devuelven cada una de las siguientes instrucciones?

- Select * from persona.
- Select edad from persona.
- Select id from persona.
- Select nombre, edad from persona.
- Select * from persona where edad > 20.
- Select nombre from persona where edad > 20.

12.10 ¿Por qué debe usar una base de datos relacional? ¿Hay otros tipos de bases de datos? ¿Quién inventó las bases de datos relacionales?

12.11 Preguntas sobre bases de datos relacionales:

- ¿Qué es una tabla de base de datos?
- ¿Qué es una unión?
- ¿Qué es una consulta?
- ¿Qué es una conexión?

12.12 Preguntas sobre bases de datos relacionales:

- ¿Qué es SQL?
- ¿Cómo creamos una tabla en SQL?
- ¿Cómo insertamos una fila en una tabla en SQL?
- ¿Cómo obtenemos datos de una tabla en SQL?

12.13 Su padre lo llama por teléfono: "Mi personal de soporte técnico dice que el sitio Web de la empresa está caído porque el programa de base de datos está averiado. ¿Qué tiene que ver la base de datos con el sitio Web de nuestra empresa?". Usted le explica cómo es que las bases de datos pueden ser fundamentales para operar sitios Web extensos. Explique (a) cómo puede crearse el sitio Web por medio de la base de datos y (b) cómo se crea el HTML en realidad.

12.14 Usted tiene una nueva computadora que parece conectarse a Internet, pero cuando trata de visitar `http://www.cnn.com` obtiene el error "No se encontró el servidor". Después de llamar a soporte técnico, le dicen que trate de ir a `http://64.236.24.20`. Eso funciona. Ahora tanto usted como el técnico saben lo que está mal con la configuración de su computadora. ¿Qué es lo que no funciona bien, ya que puede visitar un sitio por Internet pero no puede lograr que se reconozca el nombre de dominio `www.cnn.com`?

12.15 Dada una carpeta que contiene ciertas imágenes, cree una página índice de HTML con vínculos a cada una de las imágenes. Escriba una función que reciba una cadena que represente la ruta hacia un directorio. Debe crear una página llamada **index.html** en la carpeta, la cual será una página de HTML que contenga un vínculo a cada archivo JPEG en el directorio.

También debe generar una copia en miniatura (la mitad del tamaño original) de cada imagen. Use `makeEmptyPicture` para crear una imagen en blanco del tamaño correcto; después reduzca la escala de la imagen original y colóquela en la imagen en blanco. A la nueva imagen debe asignarle el nombre "mitad-" + el nombre de archivo original (por ejemplo, si el nombre de archivo original es **pedro.jpg**, guarde la nueva

imagen como **mitad-pedro.jpg**). El ancla en el vínculo a cada imagen de tamaño completo deberá ser la imagen de la mitad del tamaño.

12.16 Agregue algo a su generador de páginas de inicio de HTML para generar comentarios aleatorios relevantes sobre el clima, dependiendo de la temperatura.

- Si va a ser menor de 32°F (0°C), debe insertar "¡Cuidado con el hielo!" o "¿Acaso va a nevar?"
- Si va a estar entre 32°F (0°C) y 50°F(10°C), debe insertar "¡No puedo esperar a que acabe el invierno!" o "¡Vamos ya, primavera!"
- Si va a ser mayor de 50°F (10°C) pero menor de 80°F (26.7°C), debe insertar "¡Está empezando a hacer calor!" o "Clima para usar chaqueta ligera"
- Si va a ser mayor o igual de 80°F (26.7°C), debe insertar "¡POR FIN! ¡Verano!" u "¡Hora de ir a nadar!"

Escriba una función llamada `comentarioclima` que reciba una temperatura como entrada y devuelva una de las frases anteriores al azar. En otras palabras, debe usar la temperatura para decidir cuáles son las frases relevantes y después elegir una de ellas al azar.

12.17 Escriba una función que lea cadenas delimitadas desde un archivo con nombres y números telefónicos, que después utilice una base de datos para almacenar los nombres como las claves y los números telefónicos como los valores. Debe recibir como entrada el nombre de archivo y el nombre de la persona cuyo número telefónico estamos buscando. ¿Cómo buscaría el número telefónico para encontrar el nombre al que pertenece?

12.18 Cree una base de datos relacional que tenga una tabla de personas, una tabla de imágenes y una tabla de personas-imágenes. En la tabla de personas, almacene un número de identificación, el nombre de la persona y su edad. En la tabla de imágenes, almacene un número de identificación de imagen y el nombre de archivo. En la tabla personas-imágenes lleve la cuenta de las personas en cada imagen. Escriba una función que le permita encontrar todas las imágenes para personas mayores de cierta edad.

12.19 Cree una base de datos relacional que tenga una tabla de productos, una tabla de clientes, una de pedidos y una de artículos de pedidos. En la tabla de productos almacene un número de identificación, nombre, imagen, descripción y precio. En la de clientes, almacene el número de identificación, nombre y dirección. En la de pedidos almacene el número de identificación del pedido y el número de identificación del cliente. En la tabla de artículos de pedidos almacene el número de identificación del pedido, el número de identificación del producto y la cantidad. Escriba una función que le permita buscar todos los pedidos de un cliente específico. Escriba una función que le permita buscar todos los pedidos con un costo total mayor a cierto valor especificado.

12.20 Vaya a `www.half.com` y busque una película popular que vendan como DVD. ¿Qué tablas de bases de datos y campos cree usted que se necesitan para representar los datos para un DVD?

## PARA PROFUNDIZAR

Hay varios buenos libros sobre el uso de Python y Jython para desarrollo y programación Web. En especial le recomendamos los libros de Gupta [18] y Hightower [24] por sus discusiones sobre el uso de Java en contextos de bases de datos. Encontrará excelentes recursos en Web sobre el uso de bases de datos desde Jython, como:

`http://wiki.python.org/jython/UserGuide#getting-a-connection.`

# PARTE 4 PELÍCULAS

Capítulo 13 Creación y modificación de películas

CAPÍTULO 13

# Creación y modificación de películas

13.1 GENERACIÓN DE ANIMACIONES
13.2 TRABAJAR CON UNA FUENTE DE VIDEO
13.3 CREACIÓN DE UN EFECTO DE VIDEO DESDE ABAJO HACIA ARRIBA

### Objetivos de aprendizaje del capítulo

**Los objetivos de aprendizaje de medios para este capítulo son:**

- Comprender cómo una serie de imágenes fijas pueden percibirse como movimiento.
- Crear animaciones con distintos movimientos y efectos.
- Usar fuentes de video para animaciones y procesamiento.
- Averiguar cómo se implementa un efecto digital.

**Los objetivos de ciencias computacionales para este capítulo son:**

- Comprender otro ejemplo de uso de múltiples funciones para facilitar la codificación.
- Comprender un ejemplo detallado del diseño e implementación abajo-arriba.
- Usar argumentos opcionales y con nombre para las funciones de Python.

En realidad, las películas (video) son muy simples de manipular. Constan de arreglos de imágenes (**cuadros**). Hay que tener en cuenta la **velocidad de cuadro** (el número de cuadros por segundo), pero en general ya hemos tratado antes la mayoría de los conceptos. Usaremos el término *películas* para referirnos en forma genérica a las **animaciones** (movimiento generado en su totalidad por dibujos gráficos) y **video** (movimiento generado por algún tipo de proceso fotográfico).

Las películas funcionan gracias a una característica de nuestro sistema visual, conocida como **persistencia de la visión**. No podemos ver todos los cambios que ocurren en el mundo. Por ejemplo, por lo general no podemos ver que nuestros ojos parpadean, aun cuando lo hacen con bastante frecuencia (generalmente, 20 veces por minuto). Pero cada vez que parpadeamos, no nos entra el pánico y decimos: "¿A dónde se fue el mundo?". En vez de ello nuestros ojos retienen una imagen por un corto tiempo y siguen diciendo al cerebro que ahí sigue la misma imagen.

Si vemos una imagen *relacionada* después de otra con la suficiente rapidez, nuestro ojo retiene la imagen y nuestro cerebro ve un movimiento continuo. Una velocidad "bastante rápida" se define como alrededor de 16 cuadros por segundo: si vemos las 16 imágenes relacionadas en menos de un segundo, pensamos en movimiento continuo. Si las imágenes no

están relacionadas nuestro cerebro reporta un *montaje*: una colección de imágenes dispares (aunque tal vez conectadas en forma temática). Nos referimos a estos 16 *cuadros por segundo* (*fps*) como el límite inferior para la sensación de movimiento.

Las primeras películas mudas eran de 16 fps. Las películas de cine se estandarizaron en 24 fps para hacer el sonido más consistente; 16 fps no proporcionaban suficiente espacio físico en el filme para codificar suficientes datos de sonido (¿alguna vez se preguntó por qué las películas mudas se ven por lo común rápidas y entrecortadas? Piense en lo que ocurre al aumentar la escala de una imagen o sonido: eso es exactamente lo que ocurre si reproducimos una película de 16 fps a 24 fps). El video digital (como, las cámaras de video) captura a 30 fps. ¿Hasta qué nivel sigue siendo útil? Hay algunos experimentos de la Fuerza Aérea de EUA que sugieren que los pilotos pueden reconocer un destello de luz en la forma de una aeronave (y averiguar de qué tipo es) en 1/200 de un segundo. Los jugadores de videojuegos dicen que pueden distinguir una diferencia entre el video de 30 fps y el de 60 fps.

Es un desafío trabajar con películas debido a la cantidad y velocidad de los datos involucrados. El **procesamiento en tiempo real** de video (por ejemplo, realizar una modificación en cada cuadro a medida que entra o sale) es difícil debido a que cualquier tipo de procesamiento que realicemos debe ajustarse en 1/30 de un segundo. Vamos a realizar los cálculos para ver cuántos bytes se requieren para grabar video:

- Un segundo de imágenes con un tamaño de cuadro de 640 × 480 a 30 fps significa 30 (*cuadros*) * 640 * 480 (*píxeles*) = 9 216 000 píxeles.
- Si usamos color de 24 bits (un byte por cada uno de los valores R, G y B) tenemos 27 648 000 bytes, o 27 megabytes *por segundo*.
- Para un largometraje de 90 minutos, son 90 * 60 * 27 648 000 = 149 299 200 000 bytes, o 149 gigabytes.

Las películas digitales casi siempre se almacenan en formato comprimido. Un DVD sólo almacena 6.47 gigabytes, por lo que incluso en un DVD la película está comprimida. Los estándares de formatos de películas como *MPEG*, *QuickTime* y *AVI* son todos formatos de película comprimidos. No graban todos los cuadros: sólo graban *cuadros clave* y luego registran las diferencias entre un cuadro y el siguiente. El formato *JMV* es un poco distinto: es un archivo de imágenes JPEG, por lo que todos los cuadros están ahí, pero cada uno de ellos está comprimido.

Las películas pueden usar algunas técnicas de compresión diferentes a las imágenes o los sonidos. Considere el caso en que vemos a alguien caminar a través de un cuadro de una cámara. Entre dos cuadros sucesivos sólo hay un cambio muy pequeño: el lugar en el que se encontraba la persona y en dónde se encuentra ahora. Si sólo registráramos esas diferencias y no todos los píxeles para todo el cuadro, entonces ahorraríamos mucho espacio.

En realidad, una película MPEG es sólo una secuencia de imágenes MPEG mezcladas con un archivo de audio MPEG (como MP3). Vamos a seguir este ejemplo y *no* lidiaremos aquí con el sonido. Las herramientas que describiremos en la siguiente sección *pueden* crear películas con sonido, pero el verdadero truco de procesar las películas es manejar todas las imágenes. Entonces eso es en lo que nos enfocaremos aquí.

## 13.1 GENERACIÓN DE ANIMACIONES

Para hacer películas vamos a crear una serie de cuadros JPEG y luego los volveremos a ensamblar. Colocaremos todos nuestros cuadros en un solo directorio y los enumeraremos de modo que las herramientas sepan cómo volver a ensamblarlos en una película en el orden

correcto. Llamaremos literalmente a nuestros archivos **cuadro01.jpg**, **cuadro02.jpg** y así en lo sucesivo. Es importante incluir los ceros a la izquierda para que los archivos se listen en orden numérico cuando se coloquen por orden alfabético.

A continuación le mostramos nuestro primer programa generador de películas, que lo único que hace es mover un rectángulo rojo de la parte superior izquierda a la esquina inferior derecha (figura 13.1).

**FIGURA 13.1**
Unos cuantos cuadros de la primera película: mover un rectángulo hacia abajo y a la derecha.

**Programa 119: creación de una película con un rectángulo en movimiento**

```
def crearPeliculaRect(directorio):
  for num in range(1,30):  #29 cuadros (1 a 29)
    lienzo = makeEmptyPicture(300,200)
    addRectFilled(lienzo,num * 10, num * 5, 50,50, red)
    # convertir el número en una cadena
    numStr = str(num)
    if num < 10:
      writePictureTo(lienzo,directorio+"\\cuadro0"+numStr+".jpg")
    if num >= 10:
      writePictureTo(lienzo,directorio+"\\cuadro"+numStr+".jpg")
  pelicula = makeMovieFromInitialFile(directorio+"\\cuadro00.jpg");
  return pelicula
```

■

Para probar el código, cree un directorio llamado **rect** en el directorio **Temp** y luego ejecute lo siguiente.

```
>>> rectM = crearPeliculaRect("c:\\Temp\rect")
>>> playMovie(rectM)
```

Al ejecutar `playMovie(película)`, mostrará todos los cuadros de la película en un reproductor. Una vez que ésta haya terminado de reproducirse, puede usar el botón PREV (ANTERIOR) para ver el cuadro anterior y el botón NEXT (SIGUIENTE) para ver el siguiente cuadro en la película. El botón PLAY MOVIE (REPRODUCIR PELÍCULA) reproducirá toda la película de nuevo. El botón DELETE ALL PREVIOUS (ELIMINAR TODO ANTES DE) eliminará todos los cuadros anteriores al cuadro mostrado en el directorio. El botón DELETE ALL AFTER (ELIMINAR TODO DESPUÉS DE) eliminará todos los cuadros después del cuadro mostrado en el directorio. El botón WRITE QUICKTIME (ESCRIBIR QUICKTIME) escribirá una película QuickTime de todos los cuadros en el directorio. El botón WRITE AVI (ESCRIBIR AVI) escribirá una película AVI de todos los cuadros en el directorio. Las películas estarán en el directorio con los cuadros y tendrán el mismo nombre que el directorio.

## Cómo funciona

La parte clave de esta receta son las líneas justo después de `makeEmptyPicture`. Tenemos que calcular una posición diferente para el rectángulo, dependiendo del número de cuadro actual. Las ecuaciones en las funciones `addRectFilled()` calculan distintas posiciones para distintos cuadros de la película (figura 13.2).

**FIGURA 13.2**
El reproductor de películas mostrando la película del rectángulo.

Aunque `setPixel()` se altera si usted intenta establecer un pixel fuera de los límites de la imagen, las funciones de gráficos como `addText()` y `addRect()` no generan errores por salirse de los bordes. Sólo *recortan* la imagen (muestran sólo lo que pueden) para que podamos crear código simple para realizar animaciones sin preocuparnos por salirnos de los límites. Esto facilita de manera considerable el proceso de creación de una película de teletipo (figura 13.3).

**FIGURA 13.3**
Una película de teletipo.

**Programa 120: generación de una película de teletipo**

```
def teletipo(directorio,cadena):
  for num in range(1,100): #99 cuadros
    lienzo = makeEmptyPicture(300,100)
    # Empieza por la derecha y avanza hacia la izquierda
    addText(lienzo,300-(num*10),50,cadena)
    # Ahora, escribir el cuadro
    # Hay que lidiar con los números de cuadro de un solo dígito vs. los de¬
    doble dígito de manera distinta
    numStr=str(num)
    if num < 10:
      writePictureTo(lienzo,directorio+"//cuadro0"+numStr+".jpg")
```

¬Estas líneas del programa deben continuar con las siguientes líneas. Un comando individual en Python no puede dividirse entre varias líneas.

```
    if num>=10:
      writePictureTo(lienzo,directorio+"//cuadro"+numStr+".jpg")
```

■

### Cómo funciona

La función `teletipo` recibe un directorio en el que se guardarán los cuadros de la película, además de una cadena a escribir. Para cada uno de los 99 cuadros (del 1 al 100, pero sin incluir el 100), creamos una imagen vacía con un tamaño de (300, 100) y colocamos la cadena en esa imagen como texto. La posición y siempre está en 50 (la misma posición en sentido vertical) y la posición x (en sentido horizontal) está en `300-(num*10)`. A medida que aumenta el número de cuadro (`num`), esta ecuación se hace más pequeña. Por ende, en cada cuadro la cadena se acerca más al lado izquierdo (valores de x más pequeños) del cuadro. Convertimos el número de cuadro `num` en una cadena llamada `numStr` y la usamos para crear un nombre de archivo con el número correcto de ceros a la izquierda.

¿Podemos mover más de una cosa a la vez? ¡Por supuesto! Nuestro código de dibujo sólo se volverá un poco más complicado. Podríamos desplazar cosas con un movimiento lineal como todos los ejemplos vistos hasta ahora, pero probaremos algo diferente. He aquí una receta que usa las funciones *seno* y *coseno* para crear movimiento circular que coincida con nuestro movimiento lineal del programa 119 (página 315) (figura 13.4).

**Programa 121: mover dos objetos a la vez**

```
def rectanguloMovil2(directorio):
  for num in range(1,30):#29 cuadros
    lienzo = makeEmptyPicture(300,250)
    # agregar un rectángulo relleno que se mueva en forma lineal
    addRectFilled(lienzo,num*10,num*5, 50,50,red)

    # Hagamos que uno se mueva alrededor
    azulX = 100+int(10 * sin(num))
    azulY = 4*num+int(10* cos(num))
    addRectFilled(lienzo,azulX,azulY,50,50,blue)

    # Ahora, escribir el cuadro
    # Hay que lidiar con un solo dígito vs. doble dígito
    numStr=str(num)

    if num < 10:
      writePictureTo(lienzo,directorio+"//cuadro0"+numStr+".jpg")
    if num >= 10:
      writePictureTo(lienzo,directorio+"//cuadro"+numStr+".jpg")
```

■

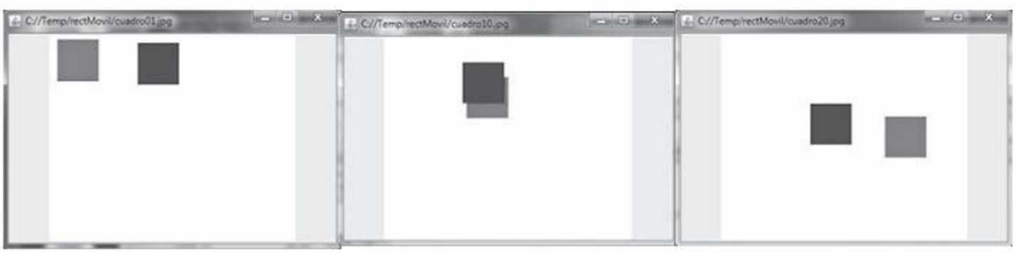


**FIGURA 13.4**
Mover dos rectángulos a la vez.

### Cómo funciona

Sabemos que las funciones `sin` y `cos` generan valores entre –1 y 1 (recuerda eso, ¿verdad?). La posición x de la caja azul se establece mediante `100 + int(10 * sin(num))`. Esto significa que la posición x de la caja azul será alrededor de 100, más o menos 10 píxeles. Se desplazará de izquierda a derecha y de derecha a izquierda, a medida que cambien los valores de `sin`. La posición y se determina mediante `4 * num + int(10 * cos(num))`. Entonces, la posición y siempre aumenta (la caja está cayendo), pero más o menos 10, por lo que hay un ligero movimiento hacia arriba y abajo mientras cae.

No tenemos que crear nuestras animaciones sólo de cosas que podamos crear con primitivas de gráficos. Podemos usar los mismos tipos de imágenes que hemos usado antes mediante `setColor()`. Este tipo de código se ejecuta con bastante lentitud.

**Tip de depuración: `PrintNow()` para imprimir AHORA**

Hay una función en JES llamada `printNow()`, la cual recibe una cadena y la imprime de *inmediato*: no espera hasta que termine la función para imprimir la línea en el área de comandos. Esto es útil cuando queremos saber en qué número de cuadro se encuentra una función. Tal vez desee analizar los primeros cuadros desde el sistema operativo, haciendo doble clic en ellos una vez que se generen.

■

El siguiente programa desplaza la cabeza de Mark por toda la pantalla. Esta función tardó cerca de un minuto en terminar en nuestra computadora. Los cuadros después de haber ejecutado el programa pueden verse en la figura 13.5.

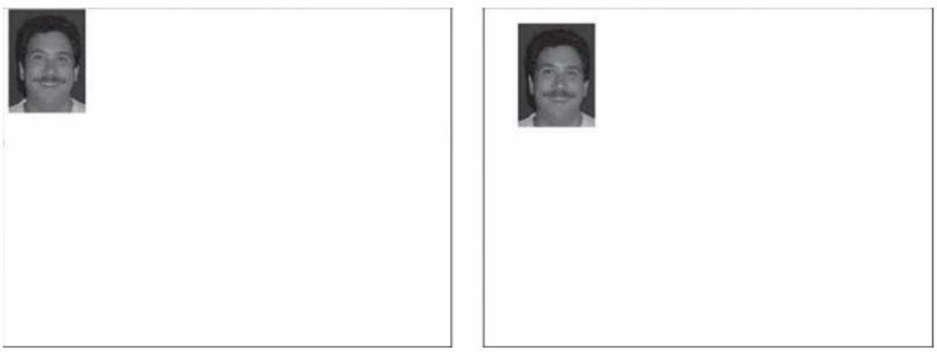


**FIGURA 13.5**
Un par de cuadros del proceso de la película para desplazar la cabeza.

**Programa 122: desplazar la cabeza de Mark**

```
def moverCabeza(directorio):
  markF=getMediaPath("blue-mark.jpg")
  mark = makePicture(markF)
  cabeza = recortar(mark,275,160,385,306)
  for num in range(1,30):#29 cuadros
    printNow("Número de cuadro: "+str(num))
    lienzo = makeEmptyPicture(640,480)
```

```
    # Ahora, realizar el proceso de copiado
    copiar(cabeza,lienzo,num*10,num*5)
    # Ahora, escribir el cuadro
    # Hay que lidiar con los números de cuadro de un solo dígito vs. los de
      doble dígito en forma diferente
    numStr=str(num)
    if num < 10:
      writePictureTo(lienzo,directorio+"//cuadro0"+numStr+".jpg")
    if num >=10:
      writePictureTo(lienzo,directorio+"//cuadro"+numStr+".jpg")

def recortar(imagen,xInicial,yInicial,xFinal,yFinal):
  anchura = xFinal - xInicial + 1
  altura = yFinal - yInicial + 1
  resImag = makeEmptyPicture(anchura,altura)
  resX = 0
  for x in range(xInicial,xFinal):
    resY=0 # restablece el índice y del resultado
    for y in range(yInicial,yFinal):
      pixelOrig = getPixel(imagen,x,y)
      resPixel = getPixel(resImag,resX,resY)
      setColor(resPixel,(getColor(pixelOrig)))
      resY=resY + 1
    resX=resX + 1
  return resImag
```

■

### Cómo funciona

Creamos un nuevo método `recortar` que crea y devuelve una nueva imagen con sólo los píxeles en el rectángulo definido por los parámetros `xInicial`, `yInicial`, `xFinal` y `yFinal` que se pasan. Usamos esta función `recortar` para crear una imagen que contenga sólo la cabeza de Mark. Después usamos la función `copiar` general (programa 30, página 97) para copiar la cabeza de Mark en distintas ubicaciones en el `lienzo`, con base en el número de cuadro `num`.

Nuestros programas de películas se están volviendo más complicados. Tal vez sea conveniente escribirlos en partes, manteniendo separada la parte de la escritura de los cuadros. Esto significa que podemos enfocarnos en el cuerpo principal de la función sobre lo que queremos en el cuadro y no sobre escribirlo. Éste es un ejemplo de **abstracción procedural**.

**Programa 123: simplificación de mover la cabeza de Mark**

```
def moverCabeza2(directorio):
  markF=getMediaPath("blue-mark.jpg")
  mark = makePicture(markF)
  cara = recortar(mark,275,160,385,306)
  for num in range(1,30):#29 cuadros
    printNow("Número de cuadro: "+str(num))
    lienzo = makeEmptyPicture(640,480)
    # Ahora, realizar el proceso de copiado
    copiar(cara,lienzo,num*10,num*5)
    # Ahora, escribir el cuadro
    escribirCuadro(num,directorio,lienzo)
```

```
def escribirCuadro(num,dir,imag):
    #Hay que lidiar con los números de cuadro de un solo dígito vs. los de
    doble dígito
    numStr=str(num)
    if num < 10:
      writePictureTo(imag,dir+"//cuadro0"+numStr+".jpg")
    if num >=10:
      writePictureTo(imag,dir+"//cuadro"+numStr+".jpg")
```

■

Esta función `escribirCuadro()` supone que se usan números de cuadro con un máximo de dos dígitos. Tal vez sean necesarios más cuadros. He aquí una versión que permite números de cuadros de tres dígitos.

**Programa 124: escribirCuadro() para más de 100 cuadros**

```
def escribirCuadro(num,dir,imag):
   #Hay que lidiar con los números de cuadro de un solo dígito vs. los de doble
   dígito numStr=str(num)
   if num < 10:
      writePictureTo(imag,dir+"//cuadro00"+numStr+".jpg")
   if num >=10 and num<100:
      writePictureTo(imag,dir+"//cuadro0"+numStr+".jpg")
   if num >= 100:
      writePictureTo(imag,dir+"//cuadro"+numStr+".jpg")
```

■

Podemos usar las manipulaciones de imágenes que creamos en el capítulo 3 sobre varios cuadros para crear películas bastante interesantes. ¿Recuerda el programa generador de atardeceres (programa 12, página 62)? Vamos a modificarlo para que genere *lentamente* un atardecer a lo largo de muchos cuadros. Lo modificaremos de modo que la diferencia entre cada cuadro sea sólo del 1%. En realidad esta versión va demasiado lejos y genera una supernova, pero el efecto es aún bastante interesante (figura 13.6).

**FIGURA 13.6**
Cuadros de la película del atardecer lento.

**Programa 125: creación de una película de atardecer lento**

```
def atardecerLento(directorio):
  # ¡fuera del ciclo!
  lienzo = makePicture(getMediaPath("beach-smaller.jpg"))
```

```
  for num in range(1,100):#99 cuadros
    printNow("Número de cuadro: "+str(num))
    crearAtardecer(lienzo)
    # Ahora, escribir el cuadro
    escribirCuadro(num,directorio,lienzo)

def crearAtardecer(imagen):
  for p in getPixels(imagen):
    valor=getBlue(p)
    setBlue(p,valor*0.99) # Una reducción de sólo 1%
    valor=getGreen(p)
    setGreen(p,valor*0.99)
```

■

## Cómo funciona

La clave para esta película es que creamos el lienzo *antes* del ciclo que crea cada cuadro. Dentro del ciclo sólo seguimos usando el mismo `lienzo` base. Esto significa que cada llamada a `crearAtardecer` aumenta la "intensidad" del atardecer en 1% (ya no mostraremos `escribirCuadro()`, pues daremos por sentado que usted lo incluirá en el área del programa y en el archivo).

El programa `cambiarFondo()` que hicimos en un capítulo anterior puede usarse también como un buen efecto para generar películas. Modificamos la función en el programa 46 (página 125) para que reciba un umbral como entrada y luego pasamos el número de cuadro como el umbral. El efecto es un desvanecimiento lento hacia la imagen de fondo (figura 13.7).

**FIGURA 13.7**
Cuadros de la película con el efecto de desvanecimiento lento.

**Programa 126: desvanecimiento lento**

```
def cambiarFondo(imag1, fondo, nuevoFdo, umbral):
  for x in range(0,getWidth(imag1)):
    for y in range(0,getHeight(imag1)):
      p1Pixel = getPixel(imag1,x,y)
      fondoPixel = getPixel(fondo,x,y)
      if (distance(getColor(p1Pixel),getColor(fondoPixel))
        < umbral):
        setColor(p1Pixel,getColor(getPixel(nuevoFdo,x,y)))
  return imag1
```

```
def desvanecerLento(directorio):
  origFondo = makePicture(getMediaPath("bgframe.jpg"))
  nuevoFondo = makePicture(getMediaPath("beach.jpg"))
  for num in range(1,60):#59 cuadros
    # hacer esto en el ciclo
    kid = makePicture(getMediaPath("kid-in-frame.jpg"))
    cambiarFondo(kid,origFondo,nuevoFondo,num)
    # Ahora, escribir el cuadro mediante
    escribirCuadro(num,directorio,kid)
```

■

### Cómo funciona

Aquí el número de cuadro es el valor de umbral por el que decidimos cambiar el nuevo fondo, en vez de mantener los píxeles originales en `cambiarFondo`. A medida que aumenta el umbral, reemplazamos cada vez más píxeles de la imagen original con los píxeles del nuevo fondo. En consecuencia, a medida que aumenta el número de cuadro, terminamos con más píxeles del fondo nuevo y menos píxeles, tanto del fondo viejo como del primer plano anterior. Observe que aquí creamos la imagen `kid` *dentro* del ciclo de cuadros. Queremos que el efecto sea nuevo para cada cuadro y calculamos las diferencias de los cuadros en `cambiarFondo` con el `umbral` que va cambiando.

## 13.2 TRABAJAR CON UNA FUENTE DE VIDEO

Como dijimos antes, lidiar con el video en tiempo real es muy difícil. Haremos un poco de trampa: guardaremos el video como una secuencia de imágenes JPEG, manipularemos las imágenes JPEG y luego las convertiremos de vuelta en una película. Esto nos permite usar video como fuente (por ejemplo, para las imágenes de fondo).

Para manipular películas ya existentes, tenemos que descomponerlas en cuadros. La aplicación MediaTools puede hacer esto por nosotros con las películas MPEG (figura 13.8). El botón MENU en la aplicación MediaTools nos permite guardar cualquier película MPEG

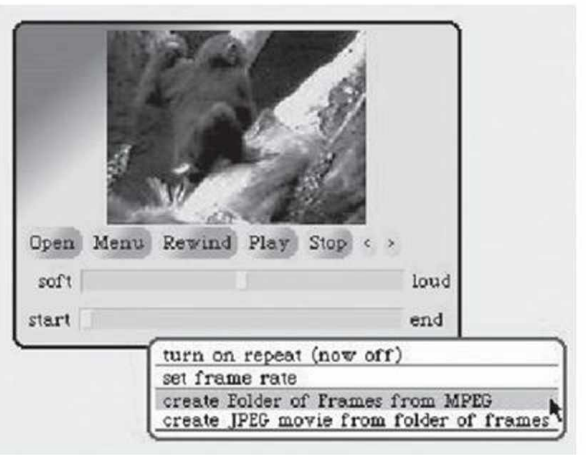



**FIGURA 13.8**
Guardar una película MPEG como una serie de cuadros en la aplicación MediaTools.

como una serie de imágenes de cuadros JPEG. Las herramientas como QuickTime Pro de Apple pueden hacer lo mismo para películas QuickTime y AVI.

### 13.2.1 Ejemplos de manipulación de video

En **mediasources** hay una pequeña película de nuestra hija Katie bailando por el cuarto. Crearemos una película de mami (Barb) viendo a su hija; sólo integraremos la cabeza de Barb en los cuadros de Katie bailando (figura 13.9).

**Programa 127: crear película de mami viendo a Katie**

```
import os

def mamiViendo(directorio):
  kidDir=r"C://ip-book//mediasources//kid-in-bg-seq"
  barbF=getMediaPath("barbaraS.jpg")
  barb = makePicture(barbF)
  cara = recortar(barb,22,9,93,97)
  num = 0
  for archivo in os.listdir(kidDir):
    if archivo.endswith(".jpg"):
      num = num + 1
      printNow("Número de cuadro: "+str(num))
      cuadroImag = makePicture(kidDir+"//"+archivo)
      # Ahora realizar el proceso de copiado
      copiar(cara,cuadroImag,num*3,num*3)
      # Ahora, escribir el cuadro
      escribirCuadro(num,directorio,cuadroImag)
```

■

**FIGURA 13.9**
Cuadros de la película de mami viendo a Katie.

#### Cómo funciona

El directorio en donde se almacenan las imágenes de origen del video es **C:/ip-book/mediasources/kid-in-bg-seq/**. Colocamos esta información en una variable y obtenemos la imagen de Barb. Después creamos una imagen de la cara de Barb usando el método `recortar`. El número de cuadro `num` se incrementa en el ciclo, ya que en realidad vamos a leer

cada cuadro de manera individual como una fuente de video mediante `os.listdir` en el directorio `kidDir` (en donde se encuentran los cuadros de video de la niña). Tenemos suerte de que `os.listdir` devuelve los cuadros en orden alfabético, que también es orden numérico (con ceros a la izquierda). Leemos el archivo, nos aseguramos que sea uno de nuestros cuadros JPG, luego lo abrimos y copiamos la cara de Barb en él. A continuación escribimos el cuadro usando `escribirCuadro`.

Sin duda podemos realizar un procesamiento de imágenes más sofisticado que integrar un rostro o crear un atardecer. Por ejemplo, podemos usar la técnica *chromakey* en cuadros de películas. Así es como se realizan muchos efectos generados por computadora en las películas reales. Para probar esto, tomamos un video sencillo de nuestros tres hijos (Matthew, Katie y Jenny) gateando y caminando enfrente de una pantalla azul (figura 13.10). No efectuamos bien la iluminación, por lo que el fondo resultó ser negro en vez de azul. Esto se convirtió en un error crítico. Como resultado, la técnica chromakey también modificó los pantalones de Matthew, el cabello de Katie y los ojos de Jenny, por lo que podemos ver la luna a través de ellos (figura 13.11). Al igual que el rojo, el negro es un color que *no* deberíamos usar para el fondo al usar la técnica chromakey.

**FIGURA 13.10**
Cuadros de los niños originales gateando y caminando frente a una pantalla azul.

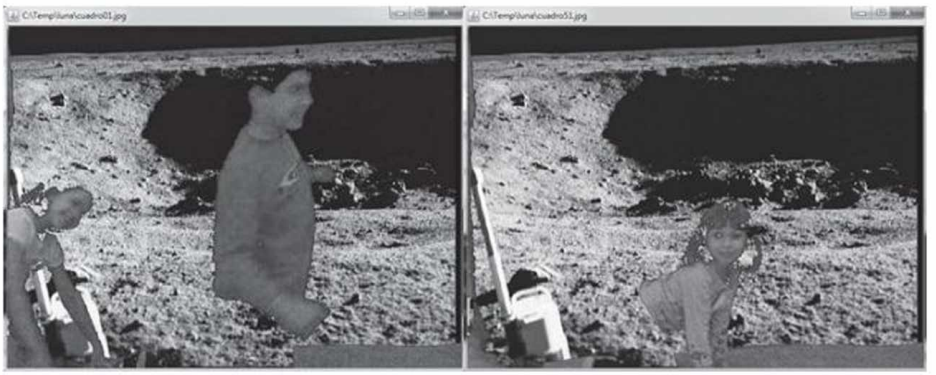


**FIGURA 13.11**
Cuadros de los niños en la película de la luna.

**Programa 128: uso de chromakey para poner a los niños en la luna**

```
import os

def chicosEnLaLuna(directorio):
  chicos="C://ip-book//mediasources//kids-blue"
  luna=getMediaPath("moon-surface.jpg")
  fondo=makePicture(luna)
  num = 0
  for cuadroArchivo in os.listdir(chicos):
    num = num + 1
    printNow("Cuadro: "+str(num))
    if cuadroArchivo.endswith(".jpg"):
      cuadro=makePicture(chicos+"//"+cuadroArchivo)
      for p in getPixels(cuadro):
        if distance(getColor(p),black) <= 100:
          setColor(p,getColor(getPixel(fondo,getX(p),getY(p))))
      escribirCuadro(num,directorio,cuadro)
```

■

Mark tomó un video de unos peces en el agua. El agua filtra la luz roja y amarilla, por lo que se ve demasiado azul (figura 13.12). Vamos a incrementar el rojo y verde en el video (amarillo es una mezcla de luz roja y verde). También crearemos una nueva función que multiplica los valores rojo y verde en una imagen por cierto factor de entrada. El resultado puede verse en la figura 13.13.

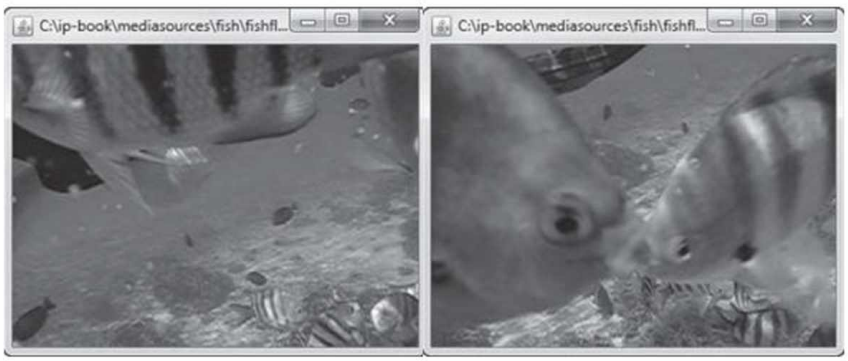



**FIGURA 13.12**
Cuadros de la película con demasiado azul bajo el agua.

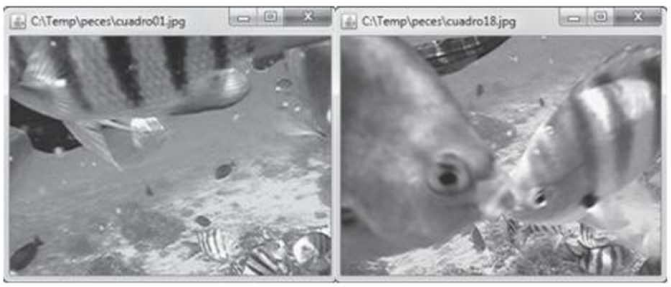



**FIGURA 13.13**
Cuadros de la película con menos azul.

**Programa 129: corrección de la película bajo el agua**

```
import os

def cambiarRojoYVerde(imag,factorRojo,factorVerde):
    for p in getPixels(imag):
        setRed(p,int(getRed(p) * factorRojo))
        setGreen(p,int(getGreen(p) * factorVerde))

def corregirBajoElAgua(directorio):
  num = 0
  dir = "C://ip-book//mediasources//fish"
  for cuadroArchivo in os.listdir(dir):
    num = num + 1
    printNow("Cuadro: "+str(num))
    if cuadroArchivo.endswith(".jpg"):
      cuadro=makePicture(dir+"//"+cuadroArchivo)
      cambiarRojoYVerde(cuadro,2.0,1.5)
      escribirCuadro(num,directorio,cuadro)
```

■

## 13.3 CREACIÓN DE UN EFECTO DE VIDEO DESDE ABAJO HACIA ARRIBA

Quizás haya visto los comerciales de televisión en donde los actores "dibujan" en el aire con una barra luminosa o una linterna, y las líneas quedan suspendidas como si se dibujaran sobre un pizarrón. ¿Cómo hacen eso? Dado lo que sabemos sobre procesamiento de imágenes y video, es probable que podamos averiguarlo. Vamos a escribir un programa para replicar el proceso; usaremos el método abajo-arriba para ejemplificar esa forma de escribir programas.

La figura 13.14 muestra cuatro cuadros de una película de alguien dibujando con una barra incandescente en el aire. Hicimos esta película cuando era oscuro afuera, para que la barra incandescente tuviera un contraste pronunciado con el resto. ¿Cómo podríamos lograr que los píxeles brillantes de la barra incandescente aparezcan en todos los cuadros sucesivos? Queremos que el último cuadro de la película creada se vea como la figura 13.15. Sabemos que el brillo es lo que medimos con luminancia. Lo que podríamos hacer es copiar de un cuadro al siguiente todos los píxeles sobre cierto nivel de luminancia. Si sólo repetimos esto para todos los cuadros, se recolectarán todos esos píxeles que son más brillantes.

En el proceso abajo-arriba empezamos por recopilar las piezas que necesitamos a partir de bibliotecas estándar, o escribimos las piezas que creemos necesitar. Es obvio que tendremos que implementar la luminancia en cierto punto.

**Programa 130: luminancia para combinar píxeles brillantes**

```
def luminancia(unpixel):
  return (getRed(unpixel)+getGreen(unpixel)+getBlue(unpixel))/3.0
```

■

También debemos probar nuestras piezas de código a medida que las vamos desarrollando. ¿Nuestra función `luminancia` hace lo que esperamos? Vamos a crear un pixel y establecer su color en ciertos valores extremos para ver si responde como podría esperarse.

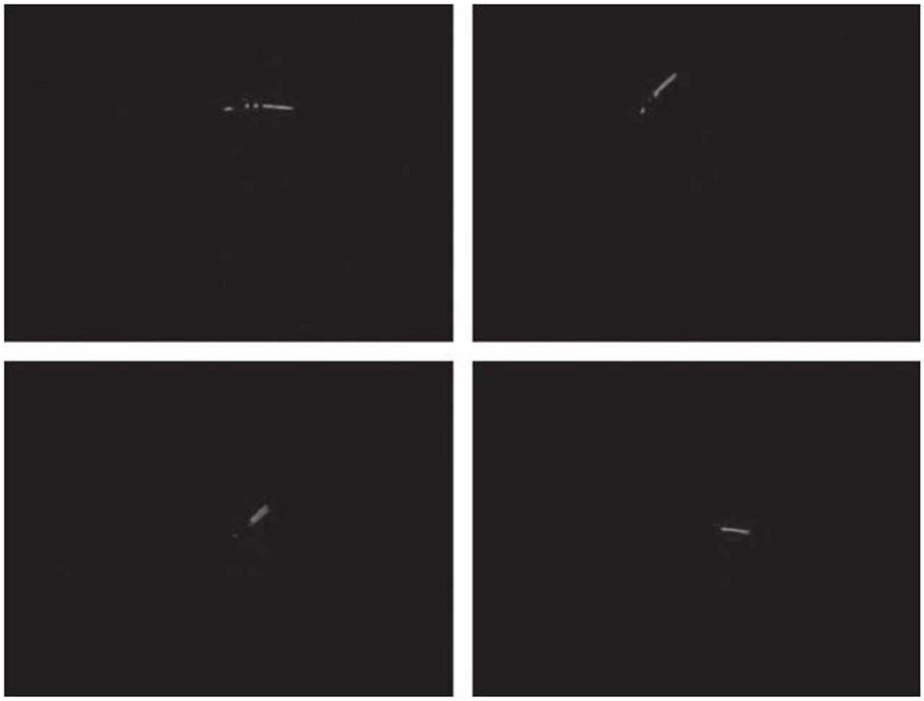


**FIGURA 13.14**
Cuadros de la película de origen, con la barra incandescente dibujándose en el aire.

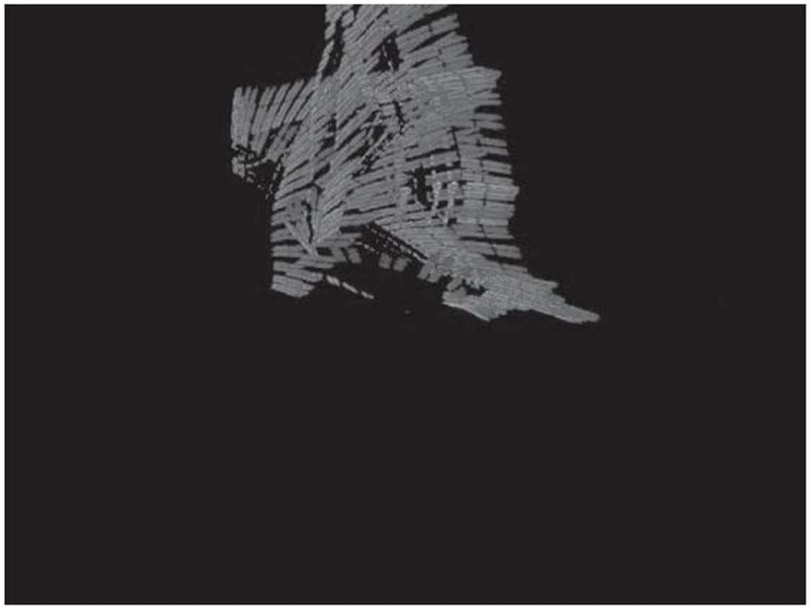


**FIGURA 13.15**
Cuadro final de la película de destino, con el rastro de la barra incandescente visible.

```
>>> imag = makeEmptyPicture(1,1)
>>> pixel=getPixelAt(imag,0,0)
>>> white
Color(255, 255, 255)
>>> setColor(pixel,white)
>>> luminancia(pixel)
255.0
>>> black
Color(0, 0, 0)
>>> setColor(pixel,black)
>>> luminancia(pixel)
0.0
```

Ahora necesitamos algo que nos indique si un pixel tiene *suficiente* brillo. Necesitamos un valor de *umbral* como punto de comparación. Podríamos codificar el método `brilloPixel` de varias formas distintas: en donde se provee un valor de umbral o en donde se usa un valor de umbral. Python nos ofrece la manera de realizar ambas cosas, mediante la capacidad de proporcionar *argumentos de palabras clave*, los que algunas veces se conocen también como *argumentos opcionales*. Para especificar el argumento de una función usamos un valor predeterminado.

**Programa 131:** `brilloPixel` **para combinar píxeles brillantes**

```
def brilloPixel(unpixel, umbral=100):
  if luminancia(unpixel) > umbral:
    return true
  return false
```

■

## Cómo funciona

La función `brilloPixel` recibe un pixel como entrada y, de manera opcional, un umbral cuyo valor predeterminado es 100. Si la luminancia del pixel es mayor que el valor de umbral, devuelve verdadero. En caso contrario, la ejecución llega a la última línea y devuelve falso. Podemos codificar `brilloPixel` de modo que saquemos ventaja al hecho de que ya escribimos la función `luminancia`.

Podemos usar `brilloPixel` especificando el valor de umbral o podemos omitirlo; incluso podemos especificarlo con una asignación.

```
>>> red
Color(255, 0, 0)
>>> setColor(pixel,red)
>>> luminancia(pixel)
85.0
>>> brilloPixel(pixel)
0
>>> brilloPixel(pixel,80)
1
>>> brilloPixel(pixel,umbral=80)
1
>>> setColor(pixel,white)
>>> brilloPixel(pixel,umbral=80)
1
```

```
>>> brilloPixel(pixel)
1
>>> setColor(pixel,black)
>>> brilloPixel(pixel,umbral=80)
0
>>> brilloPixel(pixel)
0
```

¿Qué otras funciones necesitamos? Podemos revisar nuestra descripción del enunciado. Tenemos que obtener una lista de todos los archivos de un directorio para poder obtener todos los cuadros. Dada una lista de archivos, necesitaremos la capacidad de obtener el `primerArchivo` como algo distinto al resto de los archivos (`restoArchivos`), así que recibimos el primer archivo, copiamos los píxeles al primero del resto de los archivos y luego continuamos el proceso. Escribiremos esas piezas de código.

**Programa 132: funciones para manipular listas de archivos y combinar píxeles brillantes**

```
import os

def todosLosArchivos(desdeDir):
  listaArchivos = os.listdir(desdeDir)
  listaArchivos.sort()
  return listaArchivos

def primerArchivo(listaarchivos):
  return listaarchivos[0]

def restoArchivos(listaarchivos):
  return listaarchivos[1:]
```

■

## Cómo funciona

Ya vimos estos tres tipos de código antes, por lo que vamos a basarnos en otros elementos. Aquí sólo les proporcionamos nombres adecuados y los convertimos en funciones parametrizadas. La función `todosLosArchivos` obtiene nuestra lista de archivos, después la ordena (para asegurarse que esté en orden ascendente) y la devuelve. El elemento `[0]` de una lista es el primero, y `[1:]` es a partir del segundo elemento hasta el final de la lista. Así, `primerArchivo` y `restoArchivos` nos proporcionan los elementos del archivo que queremos. Hay que probar estas funciones para asegurarnos de que produzcan el resultado que deseamos.

```
>>> archivos = todosLosArchivos("/")
>>> archivos
['Recycled', '_314109_', 'bin', 'boot', 'cdrom',
'dev', 'etc', 'home', 'initrd', 'initrd.img',
'initrd.img.old', 'lib', 'lost+found', 'media',
'mnt', 'opt', 'proc', 'root', 'sbin', 'srv', 'sys',
'tmp', 'usr', 'var', 'vmlinuz', 'vmlinuz.old']
>>> primerArchivo(archivos)
'Recycled'
>>> restoArchivos(archivos)
```

```
['_314109_', 'bin', 'boot', 'cdrom', 'dev', 'etc',
'home', 'initrd', 'initrd.img', 'initrd.img.old',
'lib', 'lost+found', 'media', 'mnt', 'opt', 'proc',
'root', 'sbin', 'srv', 'sys', 'tmp', 'usr', 'var',
'vmlinuz', 'vmlinuz.old']
```

Ya que hemos construido y evaluado todas las piezas individuales, podemos escribir la función de nivel superior con base en estas funciones más pequeñas.

**Programa 133: combinación de píxeles brillantes en una nueva película**

```
def brilloCombinar(desdeDir,destino):
  listaArchivos = todosLosArchivos(desdeDir)
  desdeArchImag = primerArchivo(listaArchivos)
  desdeImag = makePicture(desdeDir+desdeArchImag)
  for aArchImag in restoArchivos(listaArchivos):
    printNow(aArchImag)
    # Copiar todos los colores de alta luminancia de desdeImag a
      aImag
    aImag = makePicture(desdeDir+aArchImag)
    for p in getPixels(desdeImag):
      if brilloPixel(p):
        c = getColor(p)
        setColor(getPixel(aImag,getX(p),getY(p)),c)
    writePictureTo(aImag,destino+aArchImag)
    desdeImag = aImag
```

■

### Cómo funciona

La función `brilloCombinar` recibe dos directorios como entrada: dónde obtener los cuadros de los píxeles brillantes en movimiento y en dónde escribir los cuadros a donde se copian los píxeles brillantes. Obtenemos la lista de cuadros y creamos una imagen (`desdeImag`) a partir del primer cuadro en la lista. Ahora, `aArchImag` será cada uno de los otros nombres de archivo, uno a la vez, y crearemos una imagen a partir de él en la variable `aImag`. Revisamos todos los píxeles en `desdeImag` y los que tienen el suficiente brillo se escriben en `aImag`. Después hacemos que la imagen a sea la imagen *desde* mediante `desdeImag = aImag`, y avanzamos al siguiente cuadro. De esta forma copiamos todos los píxeles brillantes hacia delante.

Sin duda deberíamos probar `brilloCombinar`, pero dejaremos al lector la tarea de probar esta función. Lo que vemos en este proceso es crear elementos individuales, evaluarlos y crear más con base en esos elementos. El resultado final, la función de nivel superior, podría terminar en forma muy parecida a la que resulta del proceso arriba-abajo. En el proceso abajo-arriba tal vez no tengamos una idea clara de hacia dónde nos dirigimos (en este caso, teníamos una muy buena idea), por lo que nos enfocamos en crear elementos empezando por los más pequeños para terminar en los más grandes.

## PROBLEMAS

13.1 ¿Cuántos cuadros se necesitarían para una película de dos horas con un tamaño de imagen de 1 024 de ancho por 728 de alto y 60 cuadros por segundo? ¿Cuánto espacio en disco necesitaría esta película si guardáramos el color de todos los píxeles? Recuerde que cada pixel requeriría 24 bits para almacenar el color.

13.2 ¿Qué es AVI? ¿Qué es QuickTime? ¿Qué es MPEG? ¿Qué significa persistencia de la visión? ¿Qué es la animación basada en cuadros?

13.3 Sólo el primer ejemplo en este capítulo usa el objeto película. Vuelva a escribir algunos de los otros programas de ejemplo para crear un objeto película y devolverlo.

13.4 Cree una película en donde los cuadros obtengan un tono sepia en forma gradual y lenta.

13.5 Cree una función que tome una fotografía y haga una película en donde la imagen se convierta lentamente en el negativo.

13.6 Cree una nueva película en donde se muevan dos rectángulos una cantidad aleatoria (de –5 a 5) en cada dirección de cada cuadro.

13.7 Cree una nueva versión de la función `deteccionLineas` (programa 44, pág. 122) que reciba un umbral y haga una película usando `deteccionLineas`, en donde el umbral cambie dependiendo del número de cuadro.

13.8 Cree una nueva película en la que los niños gateen en la playa.

13.9 Cree una nueva película con el rostro de Mark viendo a Katie bailar.

13.10 Escriba una función para crear una película en donde un elemento se mueva desde la parte superior hasta la parte inferior y otro elemento se mueva de abajo hacia arriba.

13.11 Escriba una función para crear una película en donde un elemento se mueva de izquierda a derecha y otro elemento se mueva de derecha a izquierda.

13.12 Escriba una función para crear una película en donde un elemento se mueva en una línea diagonal de la parte superior izquierda a la parte inferior derecha, y que otro elemento se mueva de derecha a izquierda.

13.13 Escriba una función para crear una película en donde aparezcan círculos rellenos en intervalos aleatorios en el fondo, para luego aumentar y reducir su tamaño.

13.14 Escriba una función que dibuje un sol en la imagen de una playa en distintos lugares de la imagen, de modo que parezca como si el sol se moviera durante el día.

13.15 Escriba una función para crear una película con texto, que empiece con un tamaño grande cerca de la parte inferior de la película y luego avance hacia arriba, reduciendo su tamaño cada vez que se mueva. Puede crear un estilo de tipo de letra usando `makeStyle(familia,tipo,tamaño)`.

13.16 Haga una película de alguno de sus amigos bailando enfrente de una pantalla verde y use la técnica chromakey para hacer que parezca que está bailando en la playa.

13.17 Haga una película de una ubicación y una toma enfrente de una pantalla verde; use la técnica chromakey para mezclar las dos películas.

13.18 Cree una animación de cuando menos tres segundos de duración (30 cuadros a 10 fps, o 75 cuadros a 25 fps). Debe haber al menos tres cosas en movimiento durante esta secuencia. Debe usar al menos una imagen compuesta (una imagen JPEG que le permita aumentar o reducir su escala (si es necesario) y cópiela en la imagen) y una imagen dibujada (un rectángulo, línea, texto, óvalo o arco; cualquier cosa que usted dibuje). Cambie la *velocidad* de al menos una de las cosas que se estén moviendo a la mitad de la animación; cambie la dirección o la velocidad.

13.19 El sitio `http://abcnews.go.com` es un sitio popular de noticias. Crearemos una película de este sitio. Escriba una función que reciba un directorio como cadena. Después:

- Visite `http://abcnews.go.com` y seleccione los encabezados de las primeras tres historias de noticias (sugerencia: las anclas de todos los encabezados de historias de noticias tienen `<a href="/wire/>` como prefijo. Encuentre esa etiqueta y luego busque el principio del ancla `<a href="/wire/`, luego podrá encontrar el texto del ancla, que viene siendo el encabezado).
- Cree una película de un teletipo en el lienzo de $640 \times 480$ de las tres historias de noticias. Haga que una aparezca en la coordenada $y = 100$, otra en $y = 200$ y la tercera en $y = 300$. Genere 100 cuadros y no permita que las palabras se muevan más de 5 píxeles por cuadro (en 100 cuadros, no podrá cruzar totalmente hacia el otro lado de la pantalla; no se preocupe). Guarde los cuadros en archivos en el directorio de entrada.

13.20 Cree una película en donde una persona parezca estarse desvaneciendo de la escena. Puede basarse en la función `desvanecerLento` (programa 126, página 321).

13.21 ¿Recuerda la mezcla de imágenes en el programa 45 (páginas 123 y 124)? Intente mezclar una imagen en otra como una película, aumentando con lentitud el porcentaje de la segunda imagen (entrante) mientras reduce el porcentaje de la imagen original (saliente).

13.22 El ejemplo de rastreo de píxeles brillantes en este capítulo usó la carpeta **paint1** en **mediasources**. Hay otro ejemplo en **paint2**. Intente ejecutarlo.

13.23 Los píxeles del resultado final son bastante opacos en el ejemplo de rastreo de píxeles brillantes. Use la función `makeLighter` para hacer más brillantes los píxeles que se copiaron.

13.24 ¿Acaso la función de rastreo de píxeles brillantes funciona *sólo* en la oscuridad? Haga su propia película en donde alguien escriba con una barra incandescente o linterna, en una escena con un brillo normal. ¿La función todavía trabaja? ¿Necesita usar un valor de umbral diferente? ¿Tiene que intentar una nueva estrategia para identificar los píxeles "brillantes" (por ejemplo, tal vez al comparar la luminancia de un pixel específico con los que están a su alrededor)?

PARTE 5

# TEMAS EN CIENCIAS DE LA COMPUTACIÓN

**Capítulo 14 Velocidad**

**Capítulo 15 Programación funcional**

**Capítulo 16 Programación orientada a objetos**

CAPÍTULO

# 14 Velocidad

14.1 ENFOQUE EN CIENCIAS DE LA COMPUTACIÓN
14.2 ¿POR QUÉ LOS PROGRAMAS SON VELOCES?
14.3 ¿POR QUÉ LAS COMPUTADORAS SON VELOCES?

**Objetivos de aprendizaje del capítulo**

- Elegir entre lenguajes de programación compilados e interpretados, con base en una comprensión del lenguaje máquina y la forma en que trabajan las computadoras.
- Conocer las categorías de los algoritmos con base en su complejidad y evitar algoritmos intratables.
- Considerar la elección del procesador con base en una comprensión de las velocidades de reloj.
- Tomar decisiones sobre las opciones de almacenamiento de las computadoras cuando el objetivo es optimizar la velocidad.

## 14.1 ENFOQUE EN CIENCIAS DE LA COMPUTACIÓN

En este momento es probable que tenga muchas preguntas sobre lo que hemos estado haciendo en este libro. Por ejemplo, tal vez se pregunte:

- ¿Por qué Photoshop es más rápido que cualquier cosa que hagamos en JES?
- ¿Qué tan rápido podemos hacer que se ejecuten nuestros programas?
- ¿Siempre es así de tardado el proceso de escribir programas? ¿Puede escribir programas más pequeños que hagan lo mismo? ¿Puede escribirlos de una manera más fácil?
- ¿Cómo es la programación en otros lenguajes de programación?

Las respuestas a la mayoría de estas preguntas se conocen o estudian en las *ciencias de la computación*. Esta parte del libro es una introducción a algunos de estos temas, como indicador para que el lector pueda comenzar a explorar las ciencias de la computación con mayor profundidad.

## 14.2 ¿POR QUÉ LOS PROGRAMAS SON VELOCES?

¿A dónde se va la velocidad? Cuando tenemos una computadora muy veloz, Photoshop se ejecuta con verdadera rapidez. Los colores cambian tan pronto como cambiamos el control deslizable. Pero después ejecutamos programas en JES y tardan una *eternidad* (o 30 segundos, lo que ocurra primero). ¿Por qué?

### 14.2.1 Lo que las computadoras entienden en realidad

La verdad es que las computadoras no comprenden Python, Java ni cualquier otro lenguaje. La computadora básica comprende sólo un tipo de lenguaje: el **lenguaje máquina**. Las instrucciones en este lenguaje son sólo valores en los bytes en memoria e indican a la computadora que realice actividades de muy bajo nivel. De hecho, la computadora ni siquiera "comprende" el lenguaje máquina. Tan sólo es una máquina con muchos interruptores que hacen fluir a los datos hacia un sentido u otro. Este lenguaje es en sí un grupo de configuraciones de interruptores que cambian otros interruptores en la computadora. *Interpretamos* estos interruptores de datos como suma, resta, carga y almacenamiento de datos.

Cada tipo de computadora puede tener su propio lenguaje máquina. Los equipos Windows no pueden ejecutar programas creados para las computadoras Apple antiguas; esto no se debe a diferencias filosóficas o de marketing, sino a que cada tipo de computadora tiene su propio **procesador** (el núcleo de la computadora que ejecuta el lenguaje máquina). *Literalmente* no se entienden entre sí. Ésta es la razón por la que un programa .exe de Windows no se ejecuta en un equipo Macintosh antiguo y, de igual forma, una aplicación Macintosh no se ejecuta en un equipo Windows. Los archivos ejecutables son (casi siempre) programas en lenguaje máquina.

El lenguaje máquina parece un grupo de números; no es en definitiva amigable para el usuario. El **lenguaje ensamblador** es un conjunto de palabras (o "casi" palabras) comprensibles por el humano que corresponde uno a uno con el lenguaje máquina. Las instrucciones en lenguaje máquina indican a la computadora que realice acciones como almacenar números en ciertas ubicaciones de memoria o en ubicaciones especiales (variables o registros) en la computadora, probar la igualdad de los números o compararlos, o realizar sumas y restas de números.

El siguiente ejemplo es un programa en ensamblador para sumar dos números y almacenarlos en alguna parte (además del correspondiente lenguaje máquina generado por un **ensamblador**):

```
LOAD #10,R0   ; Carga la variable especial R0 con 10
LOAD #12,R1   ; Carga la variable especial R1 con 12
SUM R0,R1     ; Suma las variables especiales R0 y R1
STOR R1,#45   ; Almacena el resultado en la ubicación de
                memoria #45

01 00 10
01 01 12
02 00 01
03 01 45
```

Ahora veamos un programa en ensamblador que puede tomar una decisión:

```
LOAD R1,#65536        : Obtiene un carácter del teclado
TEST R1,#13           : ¿Es un 13 ASCII (Intro)?
JUMPTRUE #32768       : Si es verdadero, ir a otra parte del
                        programa
CALL #16384           : Si es falso, llamar a la función para procesar
                        la nueva línea

Lenguaje máquina

05 01 255 255
10 01 13
20 127 255
122 63 255
```

A menudo los dispositivos de entrada y salida son sólo ubicaciones de memoria para la computadora. Tal vez si almacenamos un 255 en la ubicación 65 542 de repente el componente rojo del pixel en (101, 345) se establezca en máxima intensidad. O quizás, cada vez que la computadora lea de la ubicación de memoria 897 784 sea una muestra que se lea del micrófono. De esta forma, estas operaciones simples de carga y almacenamiento pueden también manejar multimedia.

El lenguaje máquina se ejecuta con mucha rapidez. Mark escribió la primera versión de este capítulo en una computadora con un procesador de 900 megahertz (MHz). Es difícil definir con *exactitud* lo que eso significa, pero basta con decir que esta computadora procesa 900 *millones* de instrucciones de lenguaje máquina *por segundo*. Un procesador de 2 gigahertz (GHz) maneja 2 *mil millones* de instrucciones por segundo. Un programa en lenguaje máquina de 12 bytes del tipo `a = b + c` se ejecuta en la computadora antigua de Mark en alrededor de 12/900 000 000 de un segundo.

### 14.2.2 Compiladores e intérpretes

Por lo general, las aplicaciones como Adobe Photoshop y Microsoft Word se **compilan**. Esto significa que se escribieron en un lenguaje de computadora como C o C++ y luego se *traducen* al lenguaje máquina mediante el uso de un programa conocido como **compilador**. Así, los programas se ejecutan a la velocidad del procesador base.

Sin embargo, los lenguajes de programación como Python, Java, Scheme, Squeak, Director y Flash son *interpretados*. Esto quiere decir que se ejecutan a una velocidad más baja. Es la diferencia entre *traducir* y luego ejecutar las instrucciones, en comparación con la ejecución directa de las instrucciones.

Aquí podría ser útil un ejemplo detallado. Considere el siguiente ejercicio:

> Escriba una función llamada `crearGraficos` que reciba una lista como entrada. La función `crearGraficos` empezará creando un lienzo a partir del archivo **640 × 480.jpg** en la carpeta **mediasources**. Dibujará en el lienzo de acuerdo con los comandos en la lista de entrada.
>
> Cada elemento de la lista será una cadena. Habrá dos tipos de cadenas en la lista:
>
> - "b 200 120" significa dibujar un punto negro en la posición *x* 200 y en la posición *y* 120. Desde luego que los números cambiarán, pero el comando siempre será una "b". Podemos suponer que los números de entrada siempre tendrán tres dígitos.
> - "l 000 010 100 200" significa dibujar una línea de la posición (0, 10) a la posición (100, 200).
>
> Entonces, una lista de entrada podría ser `["b 100 200","b 101 200","b 102 200","l 102 200 102 300"]` (pero puede tener cualquier número de elementos).

Ahora veamos una solución al ejercicio. Analizamos cada cadena en la lista, comparamos el primer carácter como un comando de "pixel negro" o un comando "lista", después obtenemos las coordenadas correctas (y las convertimos a números mediante la función `int()`) y ejecutamos el comando de gráficos apropiado. Esta solución funciona; vea la figura 14.1.

```
>>> lienzo=crearGraficos(["b 100
    200","b 101 200","b 102
    200","l 102 200 102 300","l
    102 300 200 300"])
Dibujando pixel en 100 : 200
Dibujando pixel en 101 : 200
Dibujando pixel en 102 : 200
Dibujando línea en 102 200 102 300
Dibujando línea en 102 300 200 300
>>> show(lienzo)
```

**FIGURA 14.1**
Ejecución del intérprete `crearGraficos`.

**Programa 134: interpretar comandos de gráficos en una lista**

```
def crearGraficos(milista):
  lienzo = makePicture(getMediaPath("640 x 480.jpg"))
  for comando in milista:
    if comando[0] == "b":
      x = int(comando[2:5])
      y = int(comando[6:9])
      print "Dibujando pixel en ",x,":",y
      setColor(getPixel(lienzo, x,y),black)
    if comando[0] == "l":
      x1 = int(comando[2:5])
      y1 = int(comando[6:9])
      x2 = int(comando[10:13])
      y2 = int(comando[14:17])
      print "Dibujando línea en",x1,y1,x2,y2
      addLine(lienzo, x1, y1, x2, y2)
  return lienzo
```

■

## Cómo funciona

Aceptamos la lista de comandos gráficos como entrada en `milista`. Abrimos un lienzo **640 × 480.jpg** en blanco para dibujar en él. Para cada comando en la lista de entrada, verificamos el primer carácter (`comando[0]`) para averiguar qué tipo de comando es. Si es una "b" (pixel negro), obtenemos las coordenadas *x* y *y* de la cadena (como siempre son los números de la misma longitud, sabemos con exactitud en dónde se encontrarán) y luego dibujamos el pixel en el lienzo. Si es una "l" (línea), obtenemos las cuatro coordenadas y dibujamos la línea. Al final devolvemos el lienzo.

Lo que acabamos de hacer es implementar un nuevo lenguaje para gráficos. Incluso creamos un *intérprete* que lee las instrucciones para nuestro nuevo lenguaje y crea la imagen correspondiente. En un principio, esto es lo que PostScript, PDF, Flash y AutoCAD hacen. Sus formatos de archivos especifican las imágenes justo de la misma forma que nuestro len-

guaje de gráficos. Cuando dibujan (*generan*) la imagen en la pantalla, están *interpretando* los comandos en ese archivo.

Aunque tal vez no se note en un ejemplo tan pequeño, éste es un lenguaje relativamente lento. Considere el programa que se muestra a continuación: ¿se ejecutaría con más rapidez que leer la lista de comandos e interpretarlos? Tanto este programa como la lista en la figura 14.1 generan la misma imagen.

```
def crearGraficos():
  lienzo = makePicture(getMediaPath("640 x 480.jpg"))
  setColor(getPixel(lienzo, 100,200),black)
  setColor(getPixel(lienzo, 101,200),black)
  setColor(getPixel(lienzo, 102,200),black)
  addLine(lienzo, 102,200,102,300)
  addLine(lienzo, 102,300,200,300)
  show(lienzo)
  return lienzo
```

En general, es probable que adivinemos (correctamente) que las instrucciones que se indican antes se ejecutarán con más rapidez que leer la lista e interpretarla. He aquí una analogía que podría ser de utilidad. Mark llevó francés en la universidad, pero dice que es realmente malo para ello. Digamos que alguien le proporciona una lista de instrucciones en francés. Podría buscar de manera meticulosa cada palabra, averiguar las instrucciones y llevarlas a cabo. ¿Qué pasaría si le pidieran que realizara las instrucciones de nuevo? Vaya y busque cada palabra de nuevo. ¿Diez veces? Diez búsquedas. Ahora imaginemos que escribió la traducción al inglés (su idioma nativo) de las instrucciones en francés. Así Mark podría repetir con mucha rapidez el proceso de llevar a cabo las instrucciones tantas veces como fuera necesario, ya que tardaría muy poco tiempo en buscar cualquier palabra. En general, para descubrir el lenguaje se requiere cierto tiempo que se toma como carga adicional; siempre será más rápido sólo *ejecutar* las instrucciones o dibujar los gráficos.

Considere la siguiente idea: ¿podríamos *generar* el programa anterior? ¿Sería posible escribir un programa que recibiera como entrada la lista del lenguaje de gráficos que inventamos y después escribiera un programa en Python que dibujara las mismas imágenes? Eso no resulta ser tan difícil. Éste es un **compilador** para el lenguaje de gráficos.

**Programa 135: compilador para el nuevo lenguaje de gráficos**

```
def hacerGraficos(milista):
  archivo = open("graficos.py","wt")
  archivo.write('def crearGraficos():\n')
  archivo.write('  lienzo = makePicture(getMediaPath ("640 x 480.jpg"))\n');
  for i in milista:
    if i[0] == "b":
      x = int(i[2:5])
      y = int(i[6:9])
      print "Dibujando pixel en ",x,":",y
      archivo.write('  setColor(getPixel(lienzo, '+str(x)+', '+str(y)+'),¬
      black)\n')
    if i[0] == "l":
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un solo comando en Python no puede dividirse entre varias líneas.

```
        x1 = int(i[2:5])
        y1 = int(i[6:9])
        x2 = int(i[10:13])
        y2 = int(i[14:17])
        print "Dibujando línea en",x1,y1,x2,y2
        archivo.write('  addLine(lienzo, '+str(x1)+','+str(y1)+','+str(x2)+', '¬
        +str(y2)+')\n')
    archivo.write('  show(lienzo)\n')
    archivo.write('  return lienzo\n')
    archivo.close()
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un solo comando en Python no puede dividirse entre varias líneas.

■

### Cómo funciona

El compilador acepta la *misma* entrada que el intérprete, pero en vez de abrir un `lienzo` y escribir en él, abrimos un `archivo`. Escribimos en el archivo el inicio de la función `crearGraficos`: la instrucción `def` y el código para crear un `lienzo` (con una sangría de dos espacios para que esté dentro del bloque de la función `crearGraficos`). Tenga en cuenta que aquí no estamos *creando* el `lienzo`: tan sólo escribimos el comando para crearlo, el cual se ejecutará *más tarde* cuando se ejecute la función `crearGraficos`. Después, al igual que el intérprete, averiguamos cuál es el comando de gráficos ("b" o "l") y determinamos las coordenadas a partir de la cadena de entrada. Después escribimos en el archivo los comandos para realizar el dibujo. Al final, escribimos comandos para mostrar (`show`) y devolver (`return`) el lienzo y, por último, cerramos (`close`) el archivo.

Ahora el compilador tiene mucha carga adicional. Todavía necesitamos buscar qué significa el comando. Si sólo tenemos que ejecutar un pequeño programa de gráficos y lo necesitamos una sola vez, podríamos de igual forma ejecutar el intérprete y ya. Pero ¿qué tal si necesitáramos ejecutarlo 10 o 100 veces? Entonces pagaríamos la carga adicional de compilar el programa *una vez* y las siguientes 9 o 99 veces lo ejecutaríamos con la mayor velocidad posible. Esto sin duda será mucho más rápido que realizar el proceso adicional de interpretación 100 veces.

Esto es de lo que tratan los compiladores. Las aplicaciones como Photoshop y Word están escritas en lenguajes como C y C++, y luego se *compilan* en programas de lenguaje máquina *equivalentes*. El programa en lenguaje máquina hace lo mismo que el lenguaje C dice que haga, de igual forma que el programa de gráficos creado a partir de nuestro compilador hace lo mismo que nuestro lenguaje de gráficos dice que haga. Pero el programa en lenguaje máquina se ejecuta con *mucha mayor* rapidez que el proceso de interpretar el lenguaje C o C++.

En realidad, los programas en Jython se *interpretan* y no una, sino *dos veces*. Jython está escrito en Java y los programas en Java por lo general no se compilan en lenguaje máquina (*podemos* compilar Java en lenguaje máquina; sólo que no es lo que se hace normalmente con este lenguaje). Los programas en Java se compilan en un lenguaje máquina para un *procesador imaginario*: una **máquina virtual**. La *Máquina virtual de Java* no existe de verdad como un procesador físico. Es una definición de un procesador. ¿Qué ventaja tiene esto? Resulta ser que, como el lenguaje máquina es *muy* simple, también es bastante sencillo escribir un *intérprete* de lenguaje máquina.

El resultado es que el intérprete de una Máquina virtual de Java puede ejecutarse con mucha facilidad en casi cualquier procesador. Esto significa que un programa en Java se compila *una vez* y luego se ejecuta *en todas partes*. Los dispositivos tan pequeños como relojes de pulsera pueden ejecutar los mismos programas de Java que se ejecutan en computadoras grandes.

Al ejecutar un programa en JES, en realidad se compila en Java: se escribe un programa en Java equivalente de manera automática. Después ese programa de Java se compila para la Máquina virtual de Java. Por último, el intérprete de la Máquina virtual de Java ejecuta el lenguaje máquina de ese programa. Todo esto siempre será más lento que ejecutar una forma compilada de lo mismo.

Ésta es la primera parte de la respuesta a la pregunta: "¿Por qué Photoshop siempre es más rápido que JES?". JES se interpreta *dos veces*, lo cual siempre será más lento que Photoshop ejecutándose en lenguaje máquina.

Entonces, ¿de qué nos sirve un intérprete? Hay muchas buenas razones. Veamos tres de ellas:

- ¿Le gusta el área de comandos? ¿Ya intentó escribir algún código de ejemplo sólo para *probarla*? Ese tipo de programación interactiva, exploratoria y práctica sólo está disponible con los intérpretes. Los compiladores no nos permiten probar cosas línea por línea e imprimir los resultados. Los intérpretes son buenos para los que están aprendiendo.
- Una vez que se compila un programa en lenguaje máquina de Java, puede usarse *en cualquier parte* así como está, desde enormes computadoras hasta hornos tostadores programables. Esto representa un ahorro bastante considerable para los desarrolladores de software. Sólo tienen que producir un programa que se ejecuta en todas partes.
- Las máquinas virtuales son más seguras que el lenguaje máquina. Un programa que se ejecuta en lenguaje máquina podría realizar todo tipo de cosas inseguras. Una máquina virtual puede llevar un registro detallado de los programas que interpreta, para asegurarse de que sólo realicen acciones seguras.

### 14.2.3 ¿Qué es lo que limita la velocidad de una computadora?

El poder puro de los programas compilados, en comparación con los interpretados, es sólo una parte de la respuesta de por qué Photoshop es más rápido. La parte más profunda, que puede conducir a que los programas interpretados sean *más rápidos* que los programas compilados, está en el diseño de los *algoritmos*. Nos vemos tentados a pensar, "Oh, está bien si está lento ahora. Esperemos 18 meses a que se duplique la velocidad del procesador y entonces todo estará bien". Hay algunos algoritmos que son *tan* lentos que nunca terminarán durante el tiempo que duren nuestras vidas, y otros que en definitiva no pueden escribirse. El hecho de volver a escribir el algoritmo para que sea *más inteligente* en cuanto a lo que queremos que haga la computadora puede tener un impacto drástico sobre el rendimiento.

Un **algoritmo** es la descripción de la forma en que la computadora debe comportarse para resolver un problema. Un programa (funciones en Python) consiste en interpretaciones ejecutables de algoritmos. El mismo algoritmo puede implementarse en muchos lenguajes distintos. Siempre hay más de un algoritmo para resolver el mismo problema; algunos científicos en ciencias de la computación estudian los algoritmos y descubren formas de compararlos, además de indicar cuáles son mejores que otros.

Hemos visto varios algoritmos que aparecen de distintas formas pero que en realidad hacen lo mismo:

- Muestrear para aumentar o reducir la escala de una imagen, o para reducir o elevar la frecuencia de un sonido.
- Mezclar para fusionar dos imágenes o dos sonidos.
- Reflejar sonidos e imágenes.

Podemos comparar algoritmos con base en varios criterios. Uno de ellos establece cuánto *espacio* necesita el algoritmo para ejecutarse. ¿Cuánta memoria requiere el algoritmo? Esto puede convertirse en un problema considerable para la computación multimedia, ya que se requiere demasiada memoria para contener todos esos datos. Piense en lo malo que sería un algoritmo que necesitara guardar *todos* los cuadros de una película en una lista en la memoria al mismo tiempo.

El criterio más común que se utiliza para comparar algoritmos es el *tiempo*. ¿Cuánto tiempo requiere el algoritmo? No nos referimos al tiempo de reloj, sino a la cantidad de pasos que requiere. Los científicos computacionales usan la **notación Big-O**, también conocida como *O*(), para referirse a la magnitud o tiempo de ejecución de un algoritmo. La idea de Big-O es expresar qué tan lento se vuelve el programa a medida que aumentan los datos de entrada. Trata de ignorar las diferencias entre los lenguajes, incluso entre los que son compilados y los que son interpretados, y se enfoca en el número de *pasos* a ejecutar.

Piense en nuestras funciones básicas de procesamiento de imágenes y sonidos como `aumentarRojo()` o `aumentarVolumen()`. Parte de la complejidad de estas funciones está oculta en funciones como `getPixels()` y `getSamples()`. Sin embargo, en general nos referimos a estas funciones como $O(n)$. La cantidad de tiempo que tarda el programa en ejecutarse es linealmente proporcional a los datos de entrada. Si la imagen o sonido duplican su tamaño, lo lógico sería que el programa tardara el doble en ejecutarse.

Cuando llegamos a entender la notación Big-O, por lo común amontonamos el cuerpo del ciclo en un solo paso. Consideramos estas funciones como si se procesara cada muestra o pixel una vez, por lo que la verdadera pérdida de tiempo en estas funciones es el ciclo y en realidad no importa cuántas instrucciones contenga.

Aunque si hay otro ciclo en el cuerpo, sí es importante. Los ciclos son multiplicativos en términos del tiempo. Los ciclos anidados multiplican la cantidad de tiempo necesario para ejecutar el cuerpo. Considere el siguiente programa de ejemplo:

```
def ciclos():
  cuenta = 0
  for x in range(1,5):
    for y in range(1,3):
      cuenta = cuenta + 1
      print x,y,"--Se ejecutó ",cuenta,"veces"
```

Al ponerlo en acción podemos ver que se ejecuta ocho veces: cuatro para las x, dos para las y y $4 * 2 = 8$.

```
>>> ciclos()
1 1 --Se ejecutó  1 veces
1 2 --Se ejecutó  2 veces
2 1 --Se ejecutó  3 veces
2 2 --Se ejecutó  4 veces
3 1 --Se ejecutó  5 veces
3 2 --Se ejecutó  6 veces
4 1 --Se ejecutó  7 veces
4 2 --Se ejecutó  8 veces
```

Y ¿qué hay con el código de las películas? Como se tarda tanto en procesar, ¿es acaso un algoritmo más complejo? En realidad, no. En el código de las películas sólo se procesa cada

pixel una vez, por lo que sigue siendo $O(n)$. La única diferencia es que la $n$ es ¡muy, *pero muy* grande!

No todos los algoritmos son $O(n)$. Hay un grupo de algoritmos conocidos como **algoritmos de ordenamiento**, los cuales se utilizan para acomodar los datos por orden alfabético o numérico. Un algoritmo simple conocido como **ordenamiento de burbuja** tiene una complejidad de $O(n^2)$. En el ordenamiento de burbuja iteramos a través de los elementos en una lista, comparamos dos elementos adyacentes e intercambiamos sus valores si están desordenados. Seguimos haciendo esto hasta que al pasar por la lista ya no se produzcan intercambios, lo cual significa que los datos están ordenados.

Por ejemplo, si empezamos con una lista de (3, 2, 1), el siguiente ejemplo muestra los cambios en la lista.

```
(3,2,1) # comparar 3 y 2 e intercambiar el orden
(2,3,1) # comparar 3 y 1 e intercambiar el orden
(2,1,3) # comparar 2 y 1 e intercambiar el orden
(1,2,3) # no hay intercambios, por lo que la lista está ordenada
```

Si una lista tiene 100 elementos, se requerirán algo así como 10000 pasos para ordenar los 100 elementos con este tipo de ordenamiento. Sin embargo, existen algoritmos más inteligentes (como el ordenamiento rápido, o **Quicksort**) que tienen una complejidad de $O(n * log(n))$. En Quicksort, se selecciona como pivote uno de los valores en la lista a ordenar. Después, los valores en la lista original se dividen en dos listas: todos los valores que sean menores al pivote se mueven a una lista y todos los valores que sean mayores o iguales al pivote se mueven a la otra lista. Luego se ordenan también las dos listas mediante Quicksort. Una lista de un elemento ya está ordenada, por lo que si alguna de las listas llega a tener ese tamaño, Quicksort simplemente devuelve esa lista.

```
(5 1 3 2 7) # se elige el 3 como pivote
(1 2) 3 (5 7) # se eligen 2 y 7 como pivotes
(1) 2 3 (5) 7 # todas las listas tienen tamaño de 1,
   por lo que simplemente se devuelven
(1 2 3 5 7) # se combinan todas las listas devueltas
```

La misma lista de 100 elementos sólo requeriría 460 pasos para procesarse mediante el uso de Quicksort. Estos tipos de diferencias empiezan a ser enormes en el mundo real en cuanto al tiempo de reloj cuando por ejemplo hablamos sobre procesar 10000 clientes.

### 14.2.4 ¿En realidad existe una diferencia?

Tal vez piense que esto suena a jerigonza matemática. ¿$O(n)$? ¿$O(n^2)$? ¿$O(n!)$? Si creamos programas pequeños, ¿acaso importa si son lentos?

Considere el siguiente ejercicio mental: imagine que desea escribir un programa que genere canciones exitosas. Su programa recombinará bits de sonidos pertenecientes a algunos de los mejores ostinatos que haya escuchado en diversos instrumentos: unos 60 de ellos. Usted quiere generar todas las combinaciones de estos 60 bits (algunos entrantes, otros salientes; algunos al principio de la canción, otros al final). Desea encontrar la combinación que sea menor de 2 minutos 30 segundos (para un tiempo de reproducción óptimo en la radio) y que tenga la cantidad correcta de combinaciones de volumen alto y bajo (para ello cuenta con una función llamada `revisarSonido()`).

¿Cuántas combinaciones hay? Ignoremos el orden por ahora. Digamos que tiene tres sonidos: *a*, *b* y *c*. Sus posibles canciones son *a*, *b*, *c*, *bc*, *ac*, *ab* y *abc*. Inténtelo con dos o cuatro sonidos y verá que el patrón es el mismo que lo que vimos antes con bits: para *n* cosas, cada combinación de inclusión o exclusión es $2^n$ (si ignoramos el hecho de que hay una canción vacía, sería $2^n - 1$).

Por lo tanto, nuestros 60 sonidos resultarán en $2^{60}$ combinaciones a ejecutar a través de nuestras revisiones de longitud y sonido. Esto es 1,152,921,504,606,846,976 combinaciones. Supongamos que podemos realizar las revisiones en sólo una instrucción (algo increíble desde luego, pero estamos fingiendo). En una computadora de 1.5 GHz podemos manejar todas esas combinaciones en 768614336 segundos, cantidad que equivale a 12810238 minutos o 213504 horas, que nos da 8896 días. Esto significa que el programa tardaría 24 *años* en ejecutarse. Ahora, como la ley de Moore duplica las velocidades de procesamiento cada 18 meses, pronto podríamos ejecutar el programa en mucho menos tiempo. ¡Tan sólo *12 años*! Y si tomamos en cuenta el orden también (por ejemplo, *abc* vs. *cba* vs. *bac*), habría que agregar 63 ceros al número de combinaciones.

Por lo general se requiere mucho tiempo para encontrar la combinación totalmente óptima de casi cualquier cosa. Este tipo de $O(2^n)$ es un tiempo de ejecución bastante común para estos tipos de algoritmos. Aunque existen otros problemas que parecerían poder resolverse en un tiempo razonable, pero no es así.

Uno de éstos es el famoso *Problema del vendedor ambulante*. Imagine que usted es un vendedor y es responsable de muchos clientes diferentes; digamos que 30, la mitad del tamaño del problema de optimización. Para ser eficiente usted desea encontrar la ruta más corta en el mapa que le permita visitar a cada cliente sólo una vez y no más que eso.

El algoritmo mejor *conocido* que proporciona una solución óptima para este problema es $O(n!)$. Esto es factorial de *n*. Hay algoritmos que tardan menos tiempo en ejecutarse y producen rutas cercanas, pero no garantizan que sean las más cortas. Para 30 ciudades, el número de pasos a ejecutar con uno de estos algoritmos $O(n!)$ es 30!, o 265252859812191 058636308480000000. Si trata de probar esto en un procesador de 1.5 GHz, no terminará en toda su vida.

La parte en verdad agravante es que el problema del vendedor ambulante no es algo sacado de la manga. En realidad hay personas que tienen que planear las rutas más cortas en todo el mundo. Existen problemas similares que son en esencia el mismo algoritmo, como planificar la ruta de un robot en el piso de una fábrica. Éste es un problema grande y difícil.

¿Qué tanta diferencia hay entre $O(n)$ y $O(n!)$? Vamos a graficar esto. Nuestra entrada *n* variará de 1 a 10. Las curvas en la figura 14.2 representan diversas curvas en una escala logarítmica (vea la rapidez con que aumentan los números en el eje vertical). Si para procesar *n* piezas de datos se requieren cerca de *n* pasos (incluso $5n$ o $1000n$), entonces nuestra curva es *lineal* y nuestro programa no es demasiado lento. Pero si llegamos a $n!$, incluso con sólo 10 piezas de datos, estamos hablando de cerca de 10 millones de pasos. Si cada paso requiere una milmillonésima parte de un segundo, podemos procesar 10 piezas de datos. Pero ¿y 20? ¿100? $O(n!)$ es demasiado lento para la mayoría de los problemas reales.

Ahora bien, ¿acaso la comparación entre $O(n * log(n))$ y $O(n^2)$ marca una diferencia considerable? Si analizamos la figura 14.2, no parece que haya mucha diferencia. Pero a medida que *n* se hace más grande, la diferencia aumenta. Vea la figura 14.3. La curva para $O(n^2)$ aumenta con mucha mayor rapidez que para $O(n * log(n))$. A medida que nuestro programa procese más datos, un algoritmo $O(n^2)$ tardará *cada vez más* tiempo en ejecutarse en comparación con un algoritmo $O(n * log(n))$.

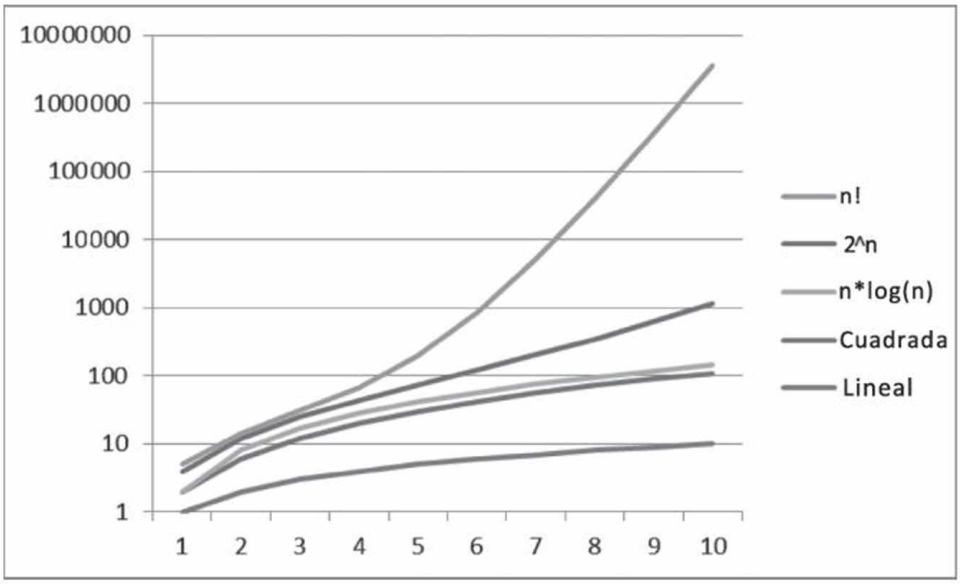



**FIGURA 14.2**
Varias curvas de *O*, en una escala logarítmica.

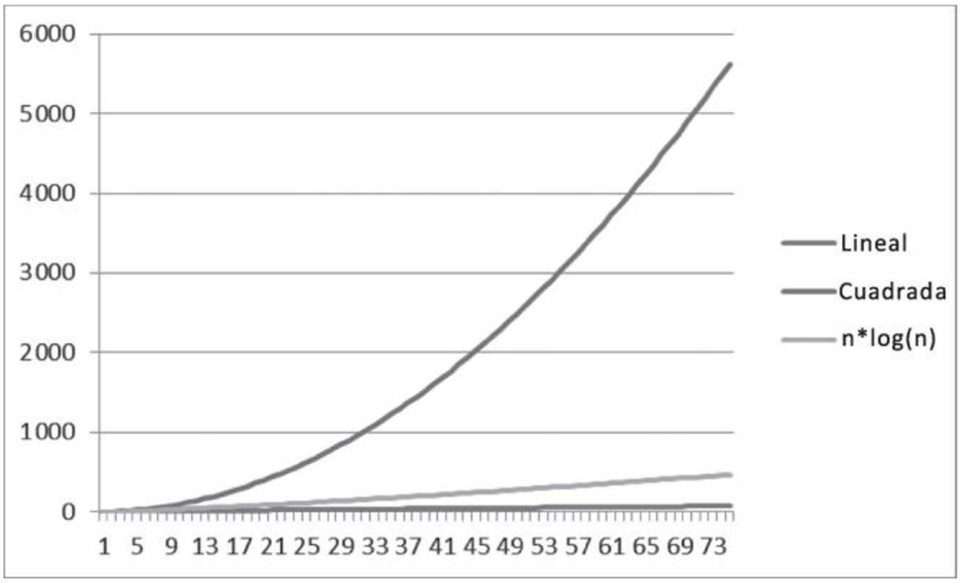



**FIGURA 14.3**
Unas cuantas curvas de *O*, en una escala lineal.

### 14.2.5 Realizar búsquedas más rápidas

Considere cómo podríamos buscar una palabra en el diccionario. Una forma sería revisar la primera página, luego la siguiente, después la otra y así en lo sucesivo. A esto se le conoce como *búsqueda lineal* y es $O(n)$. En realidad no es muy eficiente. El *mejor caso* (lo más

rápido que podría llegar a ser el algoritmo) es que el problema se resuelva en un paso: que la palabra esté en la primera página. El *peor caso* serían $n$ pasos, en donde $n$ es el número de páginas: tal vez no se encuentre la palabra. El *caso promedio* es $n/2$ pasos: la palabra se encontró a la mitad.

Podemos implementar esto como una búsqueda en una lista.

**Programa 136: búsqueda lineal de una lista**

```
def buscarEnListaOrdenada(algo, unalista):
  for elemento in unalista:
    if elemento == algo:
      return "Elemento encontrado"
  return "Elemento no encontrado"

>>> buscarEnListaOrdenada ("oso",["manzana","oso","gato","perro", "elefante"])
'Elemento encontrado'
>>> buscarEnListaOrdenada ("jirafa",["manzana","oso","gato",
"perro", "elefante"])
'Elemento no encontrado'
```

■

Ahora vamos a aprovecharnos del hecho de que los diccionarios ya están ordenados. Podemos ser más inteligentes en cuanto a la forma de buscar una palabra y hacerlo en un tiempo $O(log(n))$ ($log(n) = x$, en donde $2^x = n$). Divida el diccionario a la mitad. ¿La palabra está antes o después de la página que está viendo? Si está después, busque a partir de la mitad hasta el final (es decir, vuelva a dividir el libro, pero desde la mitad hasta el final). Si está antes, busque desde el principio hasta la mitad (vuelva a dividir el diccionario entre el principio y la mitad). Siga repitiendo este proceso hasta encontrar la palabra o cuando esté seguro de que no se encuentre en el diccionario. Éste es un algoritmo más eficiente. En el mejor caso, la palabra estará en el primer lugar en donde busque. En el caso promedio y en el peor de los casos, se necesitarán $log(n)$ pasos: siga dividiendo las $n$ páginas a la mitad y tendrá a lo más $log(n)$ divisiones.

He aquí una implementación simple (no es la mejor posible, pero sirve para fines ilustrativos) de este tipo de búsqueda, conocida como **búsqueda binaria**.

**Programa 137: búsqueda binaria simple**

```
def buscarEnListaOrdenada(algo, unalista):
  inicio = 0
  final = len(unalista) - 1
  while inicio <= final:  #Mientras haya más en dónde buscar
    puntoverificacion = int((inicio+final)/2.0)
    if unalista[puntoverificacion]==algo:
      return "Elemento encontrado"
    if unalista[puntoverificacion]<algo:
      inicio=puntoverificacion+1
    if unalista[puntoverificacion]>algo:
      final=puntoverificacion-1
  return "Elemento no encontrado"
```

■

### Cómo funciona

Empezamos con el marcador `inicio` del extremo inferior como el principio de la lista (0) y con `final` al final de la lista (longitud de la lista menos 1). Mientras que `inicio` sea menor o igual a `final`, seguiremos buscando. Calculamos el `puntoverificacion` como la mitad entre `inicio` y `final`. Después revisamos para ver si encontramos la palabra. Si es así, terminamos y devolvemos el resultado mediante `return`. En caso contrario, averiguamos si tenemos que mover `inicio` una posición más allá del `puntoverificacion` o final a una posición menor que el `puntoverificacion`. Luego seguimos buscando. Si llegamos a recorrer todo el ciclo, significa que no se ejecutó la instrucción `return "Elemento encontrado"`, por lo que devolvemos el resultado de ejecutar `return "Elemento no encontrado"`.

Para ver lo que hace este programa, agregue una línea después del cálculo del `puntoverificacion` que imprima los valores de `puntoverificacion`, `inicio` y `final`:

```
printNow("Verificando en: "+str(puntoverificacion)+"
   Inicio:"+str(inicio)+" Fin:"+str(final))
>>> buscarEnListaOrdenada("jirafa",["gato","manzana","oso",
  "perro"])
Verificando en: 1 Inicio:0 Fin:3
Verificando en: 1 Inicio:0 Fin:3
Verificando en: 3 Inicio:3 Fin:3
'Elemento no encontrado'
>>> buscarEnListaOrdenada("gato",["gato","manzana","oso",
  "perro"])
Verificando en: 1 Inicio:0 Fin:3
Verificando en: 0 Inicio:0 Fin:0
'Elemento encontrado'
>>> buscarEnListaOrdenada("perro",["gato","manzana","oso",
  "perro"])
Verificando en: 1 Inicio:0 Fin:3
Verificando en: 2 Inicio:2 Fin:3
Verificando en: 3 Inicio:3 Fin:3
'Elemento encontrado'
>>> buscarEnListaOrdenada("manzana",["gato","manzana","oso",
  "perro"])
Verificando en: 1 Inicio:0 Fin:3
'Elemento encontrado'
```

Observe que usamos `puntoverificacion = int((inicio+final)/2.0)` y no simplemente `puntoverificacion = (inicio+final)/2`. ¿Hay alguna diferencia? La segunda opción parece ser más simple.

## 14.2.6 Algoritmos que nunca terminan o que no pueden escribirse

Los científicos computacionales clasifican los problemas en tres pilas:

- Muchos problemas, como el ordenamiento, pueden resolverse con un algoritmo cuyo tiempo de ejecución tenga una complejidad polinomial, como $O(n^2)$. A estos problemas los llamamos de *clase P* (polinomial).

  Considere todos los programas que hemos escrito en este libro. Todos son de clase P. ¿Quiere eliminar los ojos rojos? Podemos hacerlo. ¿Desea encontrar las orillas de una imagen? También es posible. ¿Quiere crear un efecto de eco en un sonido? Podemos hacer eso también, todo en tiempo polinomial.

Muchas otras cosas que tal vez queramos hacer también pueden resolverse en tiempo polinomial. La búsqueda del precio de un libro con base en un código ISBN puede llevarse a cabo justo como la búsqueda que acabamos de realizar (o incluso con más rapidez, mediante el uso de otros algoritmos). Averiguar la mejor ruta desde su casa hasta la de su abuelita también se puede calcular fácilmente gracias a su dispositivo GPS.

- Otros problemas, como la optimización, tienen algoritmos conocidos que sí funcionan, pero son demasiado lentos incluso para pequeñas cantidades de datos. A estos problemas los llamamos *intratables*. Esto no significa que *no podamos* resolverlos. Simplemente se ejecutan con más lentitud.

  *Podemos* hacer que algunos de ellos se ejecuten más rápido, pero tendríamos que hacer trampa, ya que calcularíamos una solución que no fuera lo *bastante* óptima.
- Existen además otros problemas como el del vendedor ambulante, que *parecen* intratables pero tal vez haya una solución en la clase P que no hayamos encontrado todavía. A éstos les llamamos *clase NP*.

Uno de los problemas más grandes sin resolver en el ámbito teórico de las ciencias computacionales es demostrar que la clase NP y la clase P son totalmente distintas (es decir, nunca podremos resolver el problema del vendedor ambulante de una manera óptima en tiempo polinomial) o que la clase NP está dentro de la clase P.

Tal vez se pregunte si puede demostrarse *alguna cosa* sobre los algoritmos. Hay tantos lenguajes y formas diferentes de escribir el mismo algoritmo. ¿Cómo podemos *demostrar* de manera positiva que algo puede o no hacerse? Resulta ser que sí podemos. De hecho, Alan Turing demostró que hay incluso algoritmos que *no pueden escribirse*.

El algoritmo más famoso que no puede escribirse es la solución al *problema de la parada*. Hemos escrito programas que pueden leer otros programas y escribir otros programas. Imagine un programa que pueda leer otro programa y decirnos información sobre él (por ejemplo, cuántas instrucciones `print` contiene). ¿Podemos escribir un programa que reciba como entrada otro programa (por ejemplo, de un archivo) y nos indique si el programa de entrada se *detendrá* alguna vez? Considere que el programa de entrada tiene ciertos ciclos `while` complejos, en los que es difícil saber si la expresión en el ciclo `while` será alguna vez `false`. Ahora imagine un montón de estos ciclos, todos anidados uno dentro del otro.

Alan Turing demostró que dicho programa de análisis nunca podría escribirse. No es posible escribir un programa que pueda analizar un programa de entrada y decirnos de forma decisiva si alguna vez se detendrá. Alan usó la prueba con base en lo absurdo. Demostró que si dicho programa (llamémoslo H) pudiera escribirse, podríamos tratar de proporcionarle a ese programa una copia de sí mismo como entrada. Ahora bien, H recibe como entrada un programa, ¿verdad? ¿Qué pasaría si modificáramos H (llamémoslo H2) de modo que si H dijera, "Éste se detiene", H2 iterara para siempre (por ejemplo, `while 1:`)? Turing demostró que dicha configuración anunciaría que el programa se detendría sólo si iterara para siempre, y se detendría sólo si anunciara que iteraría para siempre.

Lo verdaderamente sorprendente es que Turing logró demostrar esto en 1936: casi 10 años antes de que se construyeran las primeras computadoras. Él definió un concepto matemático de una computadora, conocida como *máquina de Turing*, y pudo descubrir dichas demostraciones antes de que existieran las computadoras físicas.

Ahora considere otro ejercicio mental: ¿se puede calcular la inteligencia humana? Nuestros cerebros ejecutan un proceso que nos permite pensar, ¿verdad? ¿Podemos escribir ese proceso como un algoritmo? Y si una computadora ejecuta ese algoritmo, ¿significa que

está pensando? ¿Puede un humano reducirse a una computadora? Ésta es una de las grandes incógnitas en el campo de la **inteligencia artificial**.

La inteligencia artificial nos ofrece una gama completa de técnicas prácticas para lidiar con problemas bastante difíciles. Estas técnicas se conocen como *heurística*, reglas empíricas que nos permiten obtener una respuesta que sea lo bastante buena. Piense en el problema de traducir un idioma humano a otro. Hacer esto de una manera totalmente *correcta*, comprender cada palabra y su significado, y comprender la sintaxis del idioma humano representa un problema bastante difícil. ¿Cómo es que los servicios Web pueden hacerlo en la actualidad? A menudo usan un enfoque de *aprendizaje de máquina*. Comparan muchos documentos en ambos idiomas y calculan probabilidades estadísticas de que (por ejemplo) cierta palabra en francés coincida con cierta palabra en inglés. Al usar estas probabilidades pueden crear un documento que *muy probablemente* sea una traducción comprensible, pero que en realidad no es *100%* correcta. Aunque la mayoría de las veces, es todo lo que se necesita.

### 14.2.7 ¿Por qué Photoshop es más veloz que JES?

Ahora podemos responder la pregunta de por qué Photoshop es más veloz que JES. En primer lugar, Photoshop es una aplicación compilada, por lo que se ejecuta a velocidades de lenguaje máquina puro.

Pero en segundo lugar, Photoshop tiene algoritmos que son más inteligentes que lo que nosotros hacemos. Por ejemplo, considere los programas en donde buscamos colores, como en la técnica chromakey o al hacer rojo el cabello de Katie. Sabemos que el color de fondo y el color del cabello estaban agrupados uno al lado del otro. ¿Qué pasaría si, en vez de realizar una búsqueda lineal en todos los píxeles, empezara a buscar a partir de donde el color fuera lo que estaba buscando y continuara hasta donde ya no encontrara ese color? Buscar el límite de esta forma sería una búsqueda más inteligente. Ése es el tipo de cosas que hace Photoshop.

## 14.3 ¿POR QUÉ LAS COMPUTADORAS SON VELOCES?

Las computadoras se están volviendo cada vez más rápidas; la ley de Moore nos lo asegura. Pero saber esto no nos ayuda a comparar computadoras que pertenezcan a la misma generación de la ley de Moore. ¿Cómo comparamos los anuncios en el periódico para saber cuál de las computadoras en la lista es la más rápida *en realidad*?

La rapidez es tan sólo un criterio para elegir una computadora, claro está. También están las cuestiones del costo, cuánto espacio en disco necesitamos, qué tipo de características de expansión requerimos, y así en lo sucesivo. Pero en esta sección nos enfocaremos en el significado de los diversos factores en los anuncios de computadoras, en términos de su velocidad (en la figura 14.4 encontrará algunos ejemplos).

| |
|---|
| **Procesador:** Intel® Atom™ Dual-Core N570 (1.66GHz, caché L2 de 1MB, FSB de 667MHz)<br>**Gráficos:** Intel® Graphics Media Accelerator 3150<br>**Pantalla:** WSVGA de 10.1" (1024×600)<br>**Memoria:** SDRAM DDR3 de 1024MB<br>**Unidad de almacenamiento:** Disco duro SATA de 250 GB y 5400 RPM |
| Características clave: Procesador Intel® Core™ i3-2120, memoria DDR3 de 4GB, disco duro de 1TB, Windows 7 Home Premium, tarjeta de red inalámbrica integrada, salida HDMI, se incluye monitor LED de pantalla ancha de 21.5" |

**FIGURA 14.4**
Un par de anuncios de computadoras de ejemplo.

### 14.3.1 Velocidades de reloj y cálculos reales

Cuando los anuncios de computadoras indican que tienen un "Procesador de 2.8 GHz de cierta marca" o un "Procesador de 3.0 GHz de esta otra marca", en realidad están hablando de la *velocidad de reloj*. El procesador es la parte inteligente de su computadora: toma decisiones y realiza cálculos. Para ello realiza todo este trabajo de cálculo a cierto *ritmo*. Imagine a un sargento de instrucción gritando: "¡Vamos! ¡Vamos! ¡Vamos! ¡Vamos!". Esto es lo que representa la velocidad de reloj: nos indica qué tan rápido grita el sargento de instrucción "¡Vamos!". Una velocidad de reloj de 1.66 GHz significa que el reloj *emite pulsos* (el sargento de instrucción grita "¡Vamos!") 1.66 *mil millones* de veces por segundo.

Esto no significa que el procesador haga algo útil con cada "¡Vamos!". Algunos cálculos constan de varios pasos, por lo que se pueden requerir varios pulsos del reloj para completar un solo cálculo útil. Pero *en general*, una velocidad de reloj más alta implica cálculos más rápidos. Sin duda, para el mismo *tipo* de procesador, una velocidad de reloj mayor implica cálculos más rápidos.

¿En realidad hay alguna diferencia entre 1.66 GHz y 2.0 GHz? ¿O acaso 1.0 GHz con el procesador X es lo mismo que 2.0 GHz con el procesador Y? Estas preguntas son mucho más difíciles. En realidad no es tan distinto a discutir sobre la comparación entre camionetas Dodge y Ford. La mayoría de los procesadores tienen sus fanáticos y sus críticos. Algunos argumentarán que el procesador X puede realizar cierta búsqueda en muy pocos pulsos de reloj debido a su excelente diseño, por lo que no cabe duda que es más rápido, incluso a una velocidad de reloj más baja. Otros dirán que el procesador Y sigue siendo más rápido en general, debido a que su número promedio de pulsos de reloj por cálculo es muy bajo; y de todas formas ¿qué tan común es el tipo de búsqueda que realiza X tan rápido? Es casi como discutir sobre cuál religión es la mejor.

Muchas de las computadoras que se venden hoy día tienen varios *núcleos*. Cada núcleo es, en sentido literal, un procesador por sí solo. Así, una computadora *dual core* (*doble núcleo*) tiene entonces *dos* procesadores en su chip principal. Las computadoras *quad core* (*cuádruple núcleo*) tienen *cuatro* procesadores. ¿Acaso significa que estas computadoras son de dos a cuatro veces más rápidas? Por desgracia, no es así de sencillo. No todos los programas están escritos para sacar provecho a los múltiples núcleos. Es difícil escribir un programa que en esencia diga: "Muy bien, ahora la siguiente pieza del cálculo puede hacerse en *paralelo*, por lo que aquí están estas dos piezas y aquí está cómo las vamos a juntar al final". Si ningún software está escrito para aprovechar los múltiples núcleos, entonces tenerlos no ayudará a que aumente la velocidad de manera considerable. Uno de los grandes desafíos de las ciencias computacionales en la actualidad es cómo aprovechar todos estos múltiples núcleos para hacer que las computadoras trabajen más rápido para las personas.

La verdadera respuesta es probar cierto trabajo real en la computadora que estemos considerando. ¿Se siente lo bastante rápido? Vea las reseñas en las revistas de computadoras; a menudo usan tareas reales (como ordenar en Excel y desplazarse en Word) para probar la velocidad de las computadoras.

La mayoría de las computadoras que se usan en estos días están ocultas o son difíciles de comparar en términos de la velocidad de su procesador. ¿Es un Blackberry más rápido o lento que un iPhone o un Droid? Por lo general, los teléfonos inteligentes no se venden en términos de velocidad del procesador. Pueden compararse con respecto a muchos otros factores de más importancia, como velocidad de red y calidad de pantalla. La energía es en realidad algo muy importante con respecto a la velocidad: los procesadores más rápidos comúnmente absorben más energía, por lo que algunas veces un fabricante usa un procesador más lento para extender la vida de la batería. La mayor parte de las computadoras con las

que interactuamos a diario están ocultas de nuestra vista (algunas veces se les conoce como *computación integrada* o *embedded computing*), desde nuestro reloj hasta nuestro microondas, e incluso el interruptor de detección de movimiento que podría apagar las luces cuando nadie se esté moviendo (o desactivar la cámara de video en la noche, cuando *se espera* que no haya movimiento). En estos casos, realmente no importa la velocidad de reloj. Sólo nos preocupa que la computadora pueda hacer su trabajo.

### 14.3.2 Almacenamiento: ¿qué hace a una computadora lenta?

La velocidad de su procesador es sólo uno de los factores que influye en la rapidez o lentitud de una computadora. Tal vez un factor más importante sea el lugar de donde el procesador obtiene los datos con los que trabaja. ¿En dónde se encuentran sus imágenes cuando su computadora se pone a trabajar con ellas? Ésa es una pregunta mucho más compleja.

Podemos considerar nuestro almacenamiento como si estuviera en una jerarquía, desde el más rápido hasta el más lento.

- El almacenamiento más rápido es la **memoria caché**. Esta memoria está ubicada físicamente en el mismo chip de silicio que su procesador (o muy, muy cerca de él). Su procesador se hace cargo de colocar la mayor cantidad posible de información en la caché y dejarla ahí tanto tiempo como se requiera. El acceso a la caché es mucho más rápido que cualquier otra cosa en su computadora. Entre más memoria caché tengamos, la computadora tendrá un acceso veloz a más cosas. Pero la caché (desde luego) es el almacenamiento más costoso.
- Su almacenamiento *RAM* (ya sea que se llame **SDRAM** o cualquier otro tipo de **RAM**) es la *memoria* principal de la computadora. Una **RAM** (acrónimo de **memoria de acceso aleatorio**) de 256 Mb (megabytes) significa 256 *millones* de bytes de información. Una memoria de 1 Gb (gigabyte) significa 1 mil millones de bytes de información. El almacenamiento en RAM es donde residen sus programas al momento de ejecutarlos; también es donde se encuentran los datos sobre los que actúa directamente nuestra computadora. Las cosas están en el almacenamiento RAM antes de cargarse en la caché. La RAM es menos costosa que la memoria caché y probablemente sea su mejor inversión, en términos de aumentar la velocidad de la computadora.
- Su **disco duro** es donde guarda todos sus **archivos**. El programa que está ejecutando ahora en RAM empezó como un archivo .exe (ejecutable) en su disco duro. Todas sus imágenes digitales, música digital, archivos de procesamiento de palabras, archivos de hojas de cálculo, etcétera, se almacenan en su disco duro. Éste es el almacenamiento *más lento*, pero también es el más grande. Un disco duro de 256 Gb (gigabytes) significa que podemos almacenar 256 *mil millones* de bytes en su interior. Esto es *mucho* espacio, aunque probablemente sea poco para los estándares de hoy. Un disco duro de 1 Tb (*terabyte*) puede almacenar 1 000 gigabytes, o un *billón* de bytes.

El movimiento entre los niveles de la jerarquía implica una enorme diferencia en velocidad. Se dice que si la velocidad de acceso de la memoria caché es como alcanzar un clip de papel en su escritorio, entonces obtener algo del disco duro significa viajar a Alpha Centauri, a cuatro años luz de distancia de la Tierra. Es obvio que *obtenemos* información de nuestro disco duro a velocidades razonables (lo que en verdad implica que la memoria caché ¡es en extremo rápida!), pero esta analogía enfatiza la diferencia de los niveles de la jerarquía con respecto a la velocidad. Al final lo importante es que, entre más tengamos de la memoria más veloz, mayor será la velocidad con que nuestro procesador pueda obtener la información que deseamos y mayor será la velocidad de nuestro procesamiento en general.

En ocasiones podrá ver anuncios que mencionan el **bus del sistema**. Este elemento indica cómo se envían las señales alrededor de la computadora: desde video o red hasta disco duro, desde RAM hasta la impresora. Un bus del sistema más rápido implica sin dudas un sistema más rápido en general, pero tal vez no influya (por ejemplo) en la velocidad de nuestra experiencia con JES o Photoshop. En primer lugar, incluso hasta el bus más rápido es mucho más lento que el procesador: 400 millones de pulsos por segundo, en comparación con 4 mil millones de pulsos por segundo. En segundo lugar, el bus del sistema no influye comúnmente en el acceso a la caché o la memoria, y ahí es en donde se gana o pierde la mayor parte de la velocidad de todas formas.

Hay cosas que podemos hacer para que nuestro disco duro sea lo más rápido posible para nuestros cálculos. La velocidad del disco no es tan importante para el tiempo de procesamiento; incluso hasta los discos más rápidos siguen siendo mucho más lentos que la RAM de menor velocidad. Es importante dejar suficiente espacio libre en su disco para **intercambio** (**swapping**). Cuando su computadora no tiene suficiente RAM para lo que le pedimos hacer, almacena parte de los datos que no utiliza en ese momento de la RAM en el disco duro. Mover datos hacia y desde su disco duro es un proceso lento (hablando de manera relativa, en comparación con el acceso a la RAM). Tener un disco rápido con suficiente espacio libre de modo que la computadora no tenga que buscar **espacio de intercambio** (**swap space**) ayuda en cuanto a la velocidad de procesamiento.

¿Qué hay sobre la red? En términos de velocidad, la red no es muy útil ya que es mucho más lenta que el disco duro. Hay diferencias en las velocidades de red que sí influyen en nuestra experiencia en general, pero no necesariamente en la velocidad de procesamiento de la computadora. Las conexiones Ethernet con cable tienden a ser más rápidas que las conexiones Ethernet inalámbricas. Las conexiones por módem son más lentas.

### 14.3.3 Pantalla

¿Qué hay sobre la pantalla? ¿Acaso afecta la velocidad de su pantalla con respecto a la velocidad de su computadora? En realidad no. Las computadoras son mucho *muy* rápidas. La computadora puede volver a pintar todo incluso en pantallas bastante grandes, con una rapidez mayor de lo que usted puede percibir.

La única aplicación donde puede importar la velocidad de la pantalla es en los juegos de computadora de gama alta. Algunos jugadores afirman que pueden percibir una diferencia entre las actualizaciones de la pantalla a 50 y 60 cuadros por segundo. Si su pantalla fuera muy grande y hubiera que volver a pintar todo con cada actualización, entonces *tal vez* un procesador más veloz implicaría una diferencia que pudiera percibir. Pero la mayoría de las computadoras modernas actualizan la pantalla con tanta rapidez que no sería posible notar la diferencia.

## PROBLEMAS

14.1 Describa los siguientes términos:

- Intérprete
- Compilador
- Lenguaje máquina
- RAM
- Caché
- Problemas de clase P
- Problemas de clase NP

14.2 Busque animaciones de distintos algoritmos de ordenamiento en Web. ¿Cuál es el más rápido? ¿Cuál es el más lento?

14.3 Escriba una función para realizar un ordenamiento por inserción en una lista.

14.4 Escriba una función para realizar un ordenamiento por selección en una lista.

14.5 Escriba una función para realizar un ordenamiento por burbuja en una lista.

14.6 Escriba una función para realizar un ordenamiento por quicksort en una lista.

14.7 ¿Cuántas veces imprimirá el mensaje el siguiente código?

```
for x in range(0,5):
  for y in range(0,10):
    print "Me voy a portar bien"
```

14.8 ¿Cuántas veces imprimirá el mensaje el siguiente código?

```
for x in range(1,5):
  for y in range(0,10,2):
    print "Me voy a portar bien"
```

14.9 ¿Cuántas veces imprimirá el mensaje el siguiente código?

```
for x in range(0,3):
  for y in range(1,5):
    print "Me voy a portar bien"
```

14.10 ¿Cuál es la notación Big-O del método `borrarAzul`?

14.11 ¿Cuál es la notación Big-O del método `detectarLinea`?

14.12 Rastree el algoritmo de búsqueda binaria en `buscarEnListaOrdenada`, dada la siguiente entrada.

```
buscarEnListaOrdenada("8",["3","5","7","9","10"])
```

14.13 Rastree el algoritmo de búsqueda binaria en `buscarEnListaOrdenada`, dada la siguiente entrada.

```
buscarEnListaOrdenada("3",["3","5","7","9","10"])
```

14.14 Rastree el algoritmo de búsqueda binaria en `buscarEnListaOrdenada`, dada la siguiente entrada.

```
buscarEnListaOrdenada("1",["3","5","7","9","10"])
```

14.15 Rastree el algoritmo de búsqueda binaria en `buscarEnListaOrdenada`, dada la siguiente entrada.

```
buscarEnListaOrdenada("7",["3","5","7","9","10"])
```

14.16 Ya vio algunos ejemplos de problemas de clase P (como búsqueda y ordenamiento), problemas intratables (optimización para los elementos de canción) y problemas de clase NP (el problema del vendedor ambulante). Busque en Web y encuentre al menos un ejemplo más de cada clase de problema.

14.17 Pruebe algo que tarde una cantidad considerable de tiempo en JES (por ejemplo, la técnica chromakey en una imagen grande), de modo que pueda medir el tiempo con un cronómetro. Ahora mida la misma tarea de JES en varias computadoras distintas con diferentes cantidades de memoria y velocidades de reloj (y distintas cantidades de caché, si es posible). Vea la diferencia que marcan los distintos factores en términos del tiempo requerido para completar la tarea en JES.

14.18 Alan Turing es conocido por otro hallazgo importante en las ciencias computacionales, además de la prueba de que el problema de la parada no puede resolverse. Nos proporcionó nuestra prueba para ver si una computadora ha obtenido inteligencia o no. ¿Cuál es el nombre de esta prueba y cómo funciona? ¿Está usted de acuerdo en que es una prueba de inteligencia?

14.19 ¿Cómo obtienen las personas respuestas a los problemas que tienen algoritmos que tardarían demasiado tiempo para encontrar el resultado óptimo? Algunas veces usan *heurística*: reglas que no conducen a una solución perfecta, pero encuentran una solución. Busque algunas técnicas de heurística que se utilicen para calcular el siguiente movimiento en un programa de juego de ajedrez.

14.20 Otra forma de lidiar con los problemas intratables es mediante el uso de algoritmos `satisfactorios`. Éstos son algoritmos que encuentran una solución muy buena, pero no necesariamente la mejor. Busque un algoritmo que resuelva el problema del vendedor ambulante en un tiempo de ejecución razonable, pero que no sea el óptimo.

## PARA PROFUNDIZAR

Para aprender más sobre lo que hace que un programa funcione *bien*, le recomendamos leer *Structure and Interpretation of Computer Programs* [2]. No trata sobre gigahertz o memorias caché, pero habla mucho sobre cómo debemos idear los programas para que funcionen bien.

CAPÍTULO

# 15 Programación funcional

15.1 USO DE FUNCIONES PARA FACILITAR LA PROGRAMACIÓN

15.2 PROGRAMACIÓN FUNCIONAL MEDIANTE ASOCIACIÓN (MAP) Y REDUCCIÓN (REDUCE)

15.3 PROGRAMACIÓN FUNCIONAL PARA MULTIMEDIA

15.4 RECURSIVIDAD: UNA IDEA PODEROSA

**Objetivos de aprendizaje del capítulo**

- Escribir programas con mayor facilidad usando más funciones.
- Usar la programación funcional para crear programas poderosos con rapidez.
- Comprender la diferencia entre la programación funcional y la programación por procedimientos o imperativa.
- Poder usar la instrucción `else` en Python.
- Poder usar la instrucción `global` en Python.

## 15.1 USO DE FUNCIONES PARA FACILITAR LA PROGRAMACIÓN

¿Por qué usamos funciones? ¿Cómo podemos usarlas para facilitar la programación? Hemos estado hablando sobre varias funciones en diversos puntos del libro. Aquí sintetizaremos algunos de los beneficios antes de empezar a hablar sobre la programación con funciones de una manera más poderosa.

Las funciones son para administrar la complejidad. ¿Podemos escribir todos nuestros programas como una función extensa? Seguro, pero se vuelve *complicado*. A medida que los programas aumentan de tamaño, también aumenta su complejidad. Usamos las funciones para:

- Ocultar los detalles y poder enfocarnos sólo en lo que nos importa.
- Buscar el lugar correcto para realizar cambios según sea necesario en el programa; es más fácil encontrar la función correcta que una línea individual en unos cuantos miles de líneas de código.
- Facilitar los procesos de prueba y depuración de nuestros programas.

Si dividimos un programa en piezas más pequeñas, podemos probar cada pieza por separado. Considere nuestros programas de HTML. Es posible probar funciones como `doctype()`, `title()`, `and body()` por separado desde el área de comandos, en vez de tener siempre que probar todo el programa completo. De esta forma podemos lidiar con las funciones más pequeñas y los problemas más pequeños hasta resolverlos; así podemos ignorarlos y enfocarnos en los problemas más grandes.

```
>>> print doctype()
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/
  loose.dtd">
>>> print title("Mi cadena de título")
<html><head><title>Mi cadena de título</title></head>
>>> print body("<h1>Mi encabezado</h1><p>Mi
  párrafo</p>")
<body><h1>Mi encabezado</h1><p>Mi párrafo</p></body>
  </html>
```

Es útil poder probar funciones pequeñas como las del ejemplo anterior cuando se buscan problemas (errores). Esto también nos permite *confiar* en nuestras funciones. Probarlas de maneras distintas. Convencernos de que la función siempre realizará la tarea que le pedimos. Una vez que tengamos esta confianza, podremos pedir a la función que realice su tarea sin pensar en ello; entonces es cuando podremos realizar cosas sorprendentes y poderosas (vea la siguiente sección).

Al agregar funciones adicionales el programa se vuelve más sencillo, si es que elegimos bien esas funciones. Si tenemos subfunciones que realizan partes más pequeñas de la tarea en general, hablamos sobre cambiar la **granularidad** de las funciones. Si la granularidad se vuelve demasiado pequeña, un tipo de complejidad sustituye al otro. Pero en el nivel correcto, hace que el programa en general sea más fácil de comprender y de modificar.

Considere el programa de las páginas de inicio de esta manera, con una granularidad mucho más pequeña que lo que estábamos haciendo antes:

**Programa 138: generador de páginas de inicio con menor granularidad**

```
def crearPaginaInicio(nombre, interes):
  archivo=open("paginainicio.html","wt")
  archivo.write(doctype())
  archivo.write(inicioHTML())
  archivo.write(inicioHead())
  archivo.write(title("Página de inicio de "+nombre))
  archivo.write(finHead())
  archivo.write(inicioBody())
  archivo.write(encabezado(1,"Bienvenido a la página de inicio de "+ nombre))
  miParrafo = parrafo( "¡Hola! Soy " + nombre + ". Esta es mi página
  de inicio. Este es mi interés: " + interés +" )
  archivo.write(miParrafo)
  archivo.write(finBody())
  archivo.write(finHTML())
  archivo.close()

def doctype():
  return '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">'

def inicioHTML():
  return '<html>'
```

¬ Estas líneas del programa deben continuar con las siguientes líneas. Un solo comando en Python no puede dividirse en varias líneas.

```
def inicioHead():
  return '<head>'

def finHead():
  return '</head>'

def encabezado(nivel,cadena):
  return "<h"+str(nivel)+">"+cadena+"</h"+str(nivel)+">"

def inicioBody():
  return "<body>"

def parrafo(cadena):
  return "<p>"+cadena+"</p>"

def title(cadenatitulo):
  return "<title>"+cadenatitulo+"</title>"

def finBody():
  return "</body>"

def finHTML():
  return "</html>"
```

■

Esta versión es incluso más fácil de probar, ¡pero ahora surgió la complejidad de tener que recordar todos esos nombres de funciones que acaba de inventar!

**Tip de funcionamiento: Utilice subfunciones para las partes difíciles**

Cuando el programa contenga alguna parte difícil, divídalo en subfunciones, de manera que pueda depurar y reparar cada una de ellas por separado.

■

Considere el programa de HTML para generar una página de muestras. El ciclo fue la parte difícil del programa, así que vamos a dividirlo en una subfunción separada. Esto hace más fácil cambiar la manera en que se da formato a los vínculos; todo está en la subfunción.

**Programa 139: Creador de una página de muestras con una subfunción**

```
def crearPaginaMuestra(directorio):
  archivoMuestras=open(directorio+"//muestras.html","wt")
  archivoMuestras.write(doctype())
  archivoMuestras.write(title("Muestras de "+directorio))
  # Ahora, vamos a crear la cadena que formará el cuerpo.
  muestras="<h1>Muestras de "+directorio+" </h1>"
  for archivo in os.listdir(directorio):
    if archivo.endswith(".jpg"):
      muestras = muestras + entradaArchivo(archivo)
  archivoMuestras.write(body(muestras))
  archivoMuestras.close()
```

```
def entradaArchivo(archivo):
  muestras="<p>Nombre del archivo: "+archivo+"<br />"
  muestras=muestras+'<image src="'+archivo+'" height="100" width="100"/></p>'
  return muestras
```

■

La acción de dividir las piezas de esta forma es parte de la **abstracción procedural**. Los pasos en la abstracción procedural son:

- Declarar el problema. Averiguar lo que desea hacer.
- Dividirlo en subproblemas.
- Seguir dividiendo los subproblemas en problemas más pequeños hasta que sepa cómo escribir el programa que resuelva ese problema más pequeño.
- Su objetivo es que la función principal indique en esencia a todas las subfunciones qué deben hacer. Cada subfunción debe realizar *una y sólo una* tarea lógica.

Podemos pensar en la abstracción procedural como llenar un *árbol* de funciones (vea la figura 15.1). Realizar modificaciones es cuestión de cambiar *un* nodo (función) en este árbol y realizar adiciones es cuestión de agregar un nodo. Por ejemplo, para agregar la funcionalidad de manejar archivos WAV en nuestra página de muestras sólo hay que cambiar la función `entradaArchivo` cuando se divide de esta forma (figura 15.2).

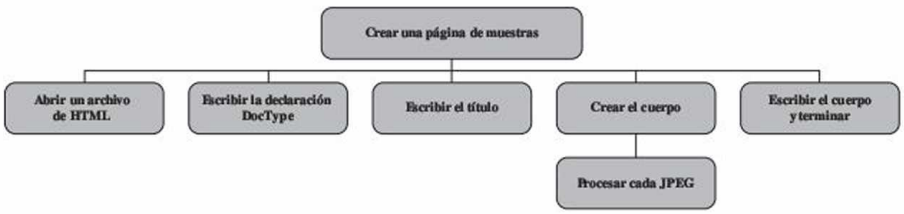



**FIGURA 15.1**
Una jerarquía de funciones para crear una página de muestras.

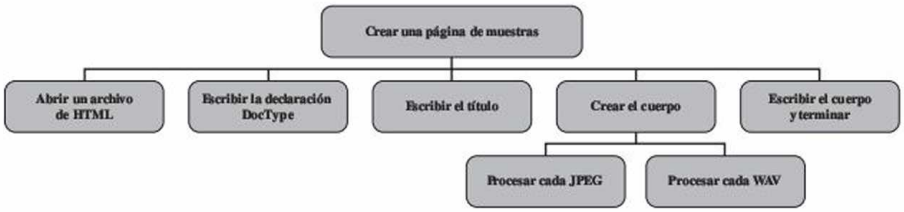



**FIGURA 15.2**
Cambiar el programa sólo es cuestión de realizar un ligero cambio en la jerarquía.

## 15.2 PROGRAMACIÓN FUNCIONAL MEDIANTE ASOCIACIÓN (MAP) Y REDUCCIÓN (REDUCE)

Si está dispuesto a confiar en sus funciones, puede escribir menos líneas de código y lograr escribir los mismos programas. Cuando llega a comprender las funciones de verdad y confía en

que realizarán lo que usted les indicó, puede hacer cosas sorprendentes en unas cuantas líneas de código. Podemos escribir funciones que apliquen otras funciones a los datos, e incluso hagan que se llamen a sí mismas en un proceso conocido como **recursividad**.

Las funciones son sólo nombres que se asocian con valores que son piezas de código, en vez de listas, secuencias, números o cadenas. Para invocar o llamar a una función declaramos su nombre seguido de las entradas entre paréntesis. Sin ellos, el nombre de la función aún tiene un valor: es el código de la función. Las funciones también pueden ser *datos*: ¡pueden pasarse como entradas a *otras* funciones!

```
>>> print crearPaginaMuestra
<function crearPaginaMuestra at 4222078>
>>> print entradaArchivo
<function entradaArchivo at 10206598>
```

La función `apply()` recibe otra función como entrada, además de las entradas (como una secuencia o lista) para esa función. En sentido literal, `apply()` aplica la función a la entrada.

```
def hola(alguien):
  print "Hola,",alguien
>>> hola("Mark")
Hola, Mark
>>> apply(hola,["Mark"])
Hola, Mark
>>> apply(hola,["Betty"])
Hola, Betty
```

Hay una función más útil que recibe otras como entrada: su nombre es `map()`. Es una función que recibe una función y una secuencia como entradas. Sólo que `map` aplica la función a *cada* entrada en la secuencia y devuelve cualquier cosa que devuelva la función para cada entrada.

```
>>> map(hola, ["Mark","Betty","Matthew","Jenny"])
Hola, Mark
Hola, Betty
Hola, Matthew
Hola, Jenny
[None, None, None, None]
```

La función `filter()` también recibe una función y una secuencia como entradas. Aplica la función a cada elemento de la secuencia. Si el *valor de retorno* de la función en ese elemento es verdadero (1), entonces el filtro devuelve ese elemento. Si el valor de retorno es falso (0), entonces el filtro omite ese elemento. Podemos usar `filter()` para extraer con rapidez ciertos datos que sean de nuestro interés.

```
def rEnNombre(unNombre):
  # find devuelve -1 cuando no encuentra nada
  if unNombre.find("r") == -1:
    return 0
  # si no es -1 entonces encontró algo
  if unNombre.find("r") != -1:
    return 1
```

```
>>> rEnNombre("Enero")
1
>>> rEnNombre("Julio")
0
>>> filter(rEnNombre, ["Mark","Betty","Matthew","Jenny"])
['Mark']
```

Podemos reescribir la función `rEnNombre()` anterior (una función que devuelve verdadero si la palabra de entrada contiene una 'r') de una manera mucho más corta. En realidad las expresiones se evalúan en 1 o 0 (verdadero o falso). Podemos realizar operaciones sobre estos *valores lógicos*. Uno de estos **operadores lógicos** es `not`: devuelve lo opuesto de su valor de entrada. Entonces aquí está `rEnNombre()` escrita usando operadores lógicos.

```
def rEnNombre2(unNombre):
  return not(unNombre.find("r") == -1)

>>> filter(rEnNombre2, ["Mark","Betty","Matthew","Jenny"])
['Mark']
```

La función `reduce()` también recibe una función y una secuencia, sólo que `reduce` *combina* los resultados. A continuación le mostramos un ejemplo donde sumamos el total de todos los números: 1 + 2, después (1 + 2) + 3, luego (1 + 2 + 3) + 4 y, por último, (1 + 2 + 3 + 4) + 5. El total hasta ese punto se pasa como la entrada `a`.

```
def sumar(a,b):
  return a+b

>>> reduce(sumar,[1,2,3,4,5])
15
```

Analicemos de nuevo el ejemplo: ¿acaso no parece un desperdicio tener que crear esa función `sumar`, puesto que es tan pequeña y casi no hace nada? Resulta ser que no tenemos que asignar un nombre a una función para poder usarla. A una función sin nombre se le conoce como `lambda`. Éste es un término muy antiguo en ciencias computacionales, que se remonta a uno de los lenguajes de programación originales: Lisp. En cualquier parte donde se utilice el nombre de una función, podemos sustituirlo por una `lambda`. La sintaxis de una `lambda` es la palabra `lambda` seguida de variables de entrada separadas por comas, después un signo de dos puntos y luego el cuerpo de la función. Ahora veremos algunos ejemplos, incluyendo el ejemplo anterior de `reduce` recreado con `lambda`. Como podemos ver, es posible definir funciones que sean casi idénticas a las que se escriben en el área del programa si se asigna un nombre a una lambda.

```
>>> reduce(lambda a,b:a+b, [1,2,3,4,5])
15
>>> (lambda a:"Hola,"+a)("Mark")
'Hola,Mark'
>>> f=lambda a:"Hola, "+a
>>> f
<function <lambda> 6>
>>> f("Mark")
'Hola, Mark'
```

Podemos realizar cálculos reales por medio de `reduce` y `lambda`. He aquí una función que calcula el **factorial** del número que recibe. El factorial de un número *n* es el producto

de todos los enteros positivos menores o iguales a n. Por ejemplo, el factorial de 4 es (4 * 3 * 2 * 1).

**Programa 140: cálculo del factorial mediante el uso de `lambda` y `reduce`**

```
def factorial(a):
  return reduce(lambda a,b:a*b, range(1,a+1))
```

■

## Cómo funciona

Es más fácil leer este programa de derecha a izquierda. Lo primero que hacemos es crear una lista de todos los números desde 1 hasta `a` con la instrucción `range(1,a+1)`. Después usamos `reduce` para aplicar una función (la `lambda`) que multiplica todos los números en la lista de entrada, uno por el siguiente, por el siguiente y así hasta terminar.

```
>>> factorial(2)
2
>>> factorial(3)
6
>>> factorial(4)
24
>>> factorial(10)
3628800
```

Tal vez esté pensando en este momento: "Muy bien, se ve que `map`, `filter` y `reduce` podrían ser útiles. *Tal vez*. Algunas veces. Pero ¿por qué rayos querría alguien usar `apply`? Es lo mismo que sólo escribir la función nosotros mismos, ¿o no?". Esto es cierto, aunque en sí podemos *crear* `map`, `filter` y `reduce` si usamos `apply`. Podemos literalmente crear cualquier versión de estas funciones que deseemos si usamos `apply`.

```
def miMap(funcion,lista):
  for i in lista:
    apply(funcion,[i])

>>> miMap(hola, ["Pedro","Pablo","Vilma","Betty"])
Hola, Pedro
Hola, Pablo
Hola, Vilma
Hola, Betty
```

Este estilo de programación se conoce como **programación funcional**. Lo que hemos estado haciendo en Python hasta ahora podría llamarse *programación por procedimientos*, ya que nos enfocamos en definir procedimientos, o *programación interactiva* debido a que en esencia indicamos a la computadora que realice acciones y modifique los valores de variables (lo que también se conoce como *estado*). En la programación funcional las ideas clave son enfocarse en las funciones como datos y en las funciones que usen otras funciones como entrada.

La programación funcional es sorprendentemente poderosa. Aplicamos capas sobre capas de funciones a otras funciones y terminamos haciendo mucho en unas cuantas líneas de código

de programa. La programación funcional se usa en la creación de sistemas de inteligencia artificial y en la construcción de prototipos. Éstas son áreas en donde los problemas son difíciles y no están bien definidos, por lo que es conveniente poder hacer mucho con sólo unas cuantas líneas de código de programa; incluso aunque esas líneas sean bastante difíciles de leer para la mayoría de la gente.

## 15.3 PROGRAMACIÓN FUNCIONAL PARA MULTIMEDIA

¿Recuerda la función para convertir a Katie en pelirroja (programa 36, página 110)? Aquí está de nuevo:

```
def convertirEnRojo():
  cafe = makeColor(42,25,15)
  archivo="C:/ip-book/mediasources/katieFancy.jpg"
  imagen=makePicture(archivo)
  for px in getPixels(imagen):
    color = getColor(px)
    if distance(color,cafe)<50.0:
      nivelrojo=int(getRed(px)*2)
      nivelazul=getBlue(px)
      nivelverde=getGreen(px)
      setColor(px,makeColor(nivelrojo,nivelazul,nivelverde))
  show(imagen)
  return(imagen)
```

Podemos escribirlo como una *sola línea* de código de programa. Necesitamos dos funciones utilitarias: una que verifique un pixel individual para ver si deseamos convertirlo en rojo y otra que realice esa acción. Nuestra línea individual de código de programa filtra los píxeles que coinciden con nuestro criterio de cambio y después asocia la función de cambio a esos píxeles. En la programación funcional no escribimos funciones con ciclos extensos. En vez de ello escribimos funciones pequeñas y las aplicamos a los datos. Es como si llevá-ramos la función a los datos, en vez de hacer que la función vaya y obtenga todos los datos.

**Programa 141: convertir el cabello en rojo, mediante programación funcional**

```
def convertirCabelloRojo(imag):
  map(convertirEnRojo,filter(verificarPixel,getPixels(imag)))

def verificarPixel(unPixel):
  cafe = makeColor(42,25,15)
  return distance(getColor(unPixel),cafe)<50.0

def convertirEnRojo(unPixel):
  setRed(unPixel,getRed(unPixel)*2)
```

■

### Cómo funciona

La función `convertirEnRojo` recibe un pixel individual y duplica su porcentaje de nivel de rojo. La función `verificarPixel` devuelve verdadero o falso si un pixel de entrada está lo

bastante cerca del color café. La función `convertirCabelloRojo` recibe una imagen como entrada y luego aplica `verificarPixel` mediante `filter` a todos los píxeles en la imagen de entrada (usando `getPixels`). Si el pixel es lo bastante café, `filter` lo devuelve. Después usamos `map` para aplicar `convertirEnRojo` a todos los píxeles devueltos por `filter`.

Así es como se usa:

```
>>> imag=makePicture(getMediaPath("KatieFancy.jpg"))
>>> map(convertirEnRojo, filter(verificarPixel, getPixels(imag)))
```

### 15.3.1 Manipulación de medios sin cambiar de estado

Otro aspecto importante de la programación funcional es el de programar *sin estado*. Nuestras funciones de manipulación de colores (por ejemplo) modifican el objeto que se provee a la función como entrada. Los buenos programas funcionales no hacen eso. Si va a modificar un objeto, un buen programa funcional realiza una copia del objeto, luego lo modifica y devuelve el objeto copiado. La ventaja es que podemos anidar funciones, pasando la salida de una a la entrada de la otra, así como podemos anidar las funciones matemáticas. Nadie espera que *seno*(*coseno*(*x*)) modifique a *x* más de lo que *seno*(*x*) modifica a *x*. Las funciones en el estilo de programación funcional deben trabajar de la misma forma. Decimos que estas funciones no tienen *efectos secundarios*. Las funciones sólo hacen lo que se supone deben hacer y *devuelven* un resultado. No modifican las entradas de ninguna forma.

Veamos cómo sería crear funciones de manipulación de medios que no cambien el estado.

**Programa 142: modificación de colores sin cambiar la imagen**

```
def reducirRojo(unaImagen):
  imagDev = makeEmptyPicture(getWidth(unaImagen),getHeight(unaImagen))
  for x in range(getWidth(unaImagen)):
    for y in range(getHeight(unaImagen)):
      pixelOrig = getPixelAt(unaImagen,x,y)
      pixelDev = getPixelAt(imagDev,x,y)
      setColor(pixelDev,getColor(pixelOrig))
      setRed(pixelDev, 0.8*getRed(pixelOrig))
  return imagDev

def aumentarAzul(unaImagen):
  imagDev = makeEmptyPicture(getWidth(unaImagen),getHeight(unaImagen))
  for x in range(getWidth(unaImagen)):
    for y in range(getHeight(unaImagen)):
      pixelOrig = getPixelAt(unaImagen,x,y)
      pixelDev = getPixelAt(imagDev,x,y)
      setColor(pixelDev,getColor(pixelOrig))
      setBlue(pixelDev, 1.2*getBlue(pixelOrig))
  return imagDev
```

■

### Cómo funciona

Ambas funciones tienen la misma estructura básica. Se crea un objeto a devolver (`imagDev`) del mismo tamaño que la imagen de entrada. Para cada pixel en la imagen de entrada, copiamos el color al pixel correspondiente en la imagen `imagDev`. Después reducimos el rojo o aumentamos el azul. Por último, devolvemos la `imagDev`.

Podemos aplicar estas funciones a una imagen sin cambiar la imagen original. Podemos anidar las funciones de cualquier forma que deseemos. Ahora las funciones son mucho más parecidas a las funciones matemáticas.

```
>>> nuevaImag = aumentarAzul(reducirRojo(imag))
>>> show(nuevaImag)
>>> show(reducirRojo(imag))
>>> show(reducirRojo(aumentarAzul(imag)))
>>> show(aumentarAzul(imag))
```

## 15.4 RECURSIVIDAD: UNA IDEA PODEROSA

La *recursividad* implica escribir funciones que se llamen a *sí mismas*. En vez de escribir ciclos, escribimos una función que itera al llamarse a sí misma una y otra vez sin detenerse. Cuando escribimos una función recursiva, siempre tenemos al menos dos partes:

- Qué hacer cuando termine (por ejemplo, cuando se procesa el último elemento en los datos), y
- Qué hacer cuando los datos son más grandes, lo cual por lo general implica procesar un elemento de los datos y luego volver a llamar la función para lidiar con el resto.

Vamos a explorar la recursividad con algunas funciones textuales simples, antes de atacar los medios con recursividad. Considere la forma en que podríamos escribir una función que haga lo siguiente:

```
>>> abajoArriba("Hola")
Hola
ola
la
a
la
ola
Hola
```

La recursividad puede ser algo difícil de digerir mentalmente. En realidad depende de que *confiemos* en nuestras funciones. ¿Hace la función lo que se supone debe hacer? Entonces sólo hay que llamarla y hará lo correcto.

Vamos a hablar sobre la recursividad de tres formas para ayudarle a comprender este concepto. La primera forma es la abstracción procedural: dividir el problema en las piezas más pequeñas que podamos escribir con facilidad como funciones y reutilizarlas lo más que se pueda.

Consideremos la función `abajoArriba` para palabras de un solo carácter. Eso es fácil:

```
def abajoArriba1(palabra):
  print palabra

>>> abajoArriba1("I")
I
```

Ahora veamos `abajoArriba` para palabras de dos caracteres. Reutilizaremos `abajoArriba1` porque ya la tenemos.

```
def abajoArriba2(palabra):
  print palabra
  abajoArriba1(palabra[1:])
  print palabra
>>> abajoArriba2("tu")
tu
u
tu
>>> abajoArriba2("yo")
yo
o
yo
```

Ahora para palabras de tres caracteres:

```
def abajoArriba3(palabra):
  print palabra
  abajoArriba2(palabra[1:])
  print palabra

>>> abajoArriba3("pop")
pop
op
p
op
pop
>>> abajoArriba3("tos")
tos
os
s
os
tos
```

¿Ya descubrió el patrón? Vamos a probarlo:

```
def pruebaAbajoArriba(palabra):
  print palabra
  pruebaAbajoArriba(palabra[1:])
  print palabra
```

```
>>> pruebaAbajoArriba("hola")
hola
ola
la
a
The error was:java.lang.StackOverflowError
I wasn't able to do what you wanted.
The error java.lang.StackOverflowError has occured
Please check line 101 of C:\ip-book\programs\
  functional
```

El error es un poco distinto en Python regular, pero en esencia significa lo mismo:

```
>>> pruebaAbajoArriba("hola")
...
  File "<stdin>", line 3, in pruebaAbajoArriba
  File "<stdin>", line 3, in pruebaAbajoArriba
RuntimeError: maximum recursion depth exceeded
```

¿Qué ocurrió? Al llegar a un carácter, seguimos llamando a `pruebaArribaAbajo` en forma repetida hasta quedarnos sin memoria en un área conocida como la *Pila*. Necesitamos poder decir a nuestra función: "Si nos queda un solo carácter, sólo imprímelo y *no* te vuelvas a llamar a ti misma". La siguiente función se ejecuta sin problemas.

**Programa 143: función arribaAbajo recursiva**

```
def abajoArriba(palabra):
  print palabra
  if len(palabra)==1:
    return
  abajoArriba(palabra[1:])
  print palabra
```

■

He aquí nuestra segunda forma de pensar sobre la recursividad: ¡el infalible rastreo! Insertaremos comentarios con sangrías.

```
>>> abajoArriba("Hola")
```

| | |
|---|---|
| | `len(palabra)` no es 1, por lo que imprimimos la palabra |
| `Hola` | |
| | Ahora llamamos a `abajoArriba("ola")`<br>Todavía no es un carácter, por lo que se imprime |
| `ola` | |
| | Ahora llamamos a `abajoArriba("la")`<br>Todavía no es un carácter, por lo que se imprime |

| | |
|---|---|
| `la` | |
| | Ahora llamamos a `abajoArriba("a")`<br>*¡Es un carácter!* Lo imprime y regresa mediante `return` |
| `a` | |
| | Ahora `abajoArriba("la")` continúa a partir de su llamada a `abajoArriba("a")`<br>Imprime de nuevo y termina. |
| `la` | |
| | Ahora `abajoArriba("ola")` continúa.<br>Imprime y termina. |
| `ola` | |
| | Por último, ahora puede ejecutarse la última línea de la función `abajoArriba("Hola")` original. |
| `Hola` | |

Una tercera forma de pensar en esto es imaginar la *invocación a una función* como si fuera un *elfo*: una persona pequeña dentro de la computadora que va a hacer lo que usted le ordene.

He aquí las instrucciones de `abajoArriba` para los elfos:

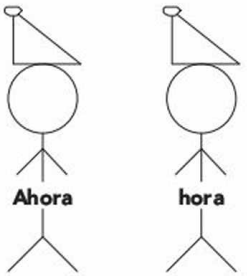
1. Aceptar una palabra como entrada.
2. Si la palabra tiene sólo un carácter, escribirla en la pantalla y terminar. Detenerse y sentarse.
3. Escriba su palabra en la pantalla.
4. Contrate otro elfo para que haga las mismas instrucciones y dé a ese elfo su palabra menos el primer carácter.
5. Espere hasta que el elfo que contrató termine.
6. Escriba su palabra en la pantalla de nuevo. Listo.

Le sugiero probar esto en su propia clase; es divertido y ayuda a que la recursividad tenga sentido. Funciona de manera similar a esto:

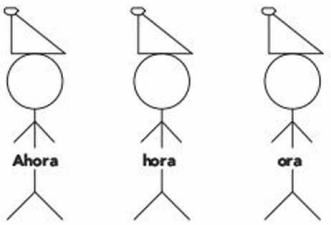
- Empezamos contratando a nuestro primer elfo con la entrada “Ahora”.

(Considere esto como la abstracción de un elfo).

- El elfo que lleva la palabra "Ahora" sigue las instrucciones. Acepta la palabra como entrada, ve que tiene más de un carácter y escribe en la pantalla: `Ahora`. Después contrata a un nuevo elfo y le proporciona la entrada "hora".




- El elfo que lleva "hora" acepta la entrada, ve que tiene más de un carácter y la escribe en la pantalla (`hora`, justo debajo de `Ahora`). Después contrata un nuevo elfo y le entrega la entrada "ora".

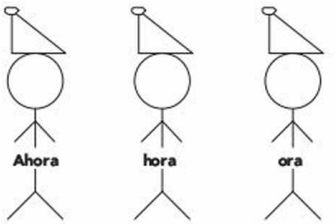



- En este punto podemos hacer unas cuantas observaciones. Cada elfo sólo está consciente del elfo a su izquierda: el que contrató. Tiene que esperar a que ese elfo termine para que a su vez pueda terminar. Cuando los elfos empiezan a terminar, los primeros en terminar serán los que están a la derecha, que fueron los últimos contratados. A esto le llamamos **pila**: los elfos se "apilan" de izquierda a derecha y el *último* en entrar es el *primero* en salir.

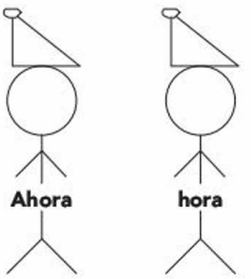
  Si la entrada original fuera una palabra muy grande (por ejemplo, "supercalifragilisticoexpialidoso"), podríamos considerar que no haya espacio suficiente para que todos los elfos se apilen. A esto le llamamos **desbordamiento de pila**: es el error que Python nos proporciona si la recursividad se vuelve demasiado profunda (es decir, demasiados elfos).

- Imagine que continuamos la simulación. Contratamos elfos para "ra" y "a". El elfo de "a" escribe su `a` y luego se sienta.

- Ahora el elfo de "ra" termina. Escribe `ra` en la pantalla (que está debajo de la `a`, que a su vez está debajo de `ra`). Y el elfo de "ra" se sienta. Esto nos deja con tres elfos más que esperan sus turnos.




- El elfo de "ora" escribe `ora` y se sienta.




- El elfo de "hora" ha estado esperando a que el elfo de "ora" termine. Escribe `hora` en la pantalla y se sienta.

- Por último, el elfo de "Ahora" escribe `Ahora` por segunda vez y se sienta. Ahora la pila está vacía.

¿Por qué usar programación funcional y recursividad? Porque nos permite hacer mucho en muy pocas líneas de código. Es una técnica muy útil para lidiar con problemas difíciles. Es posible implementar cualquier tipo de ciclo con la recursividad. Muchas personas sienten que es la forma más flexible, elegante y poderosa de iterar.

### 15.4.1 Recorridos recursivos de directorios

La primera estructura recursiva que vimos en este libro fue un árbol de directorios. Una carpeta puede contener otras carpetas y así en lo sucesivo, hasta cualquier nivel de carpetas. La forma más sencilla de *recorrer* (tocar todos los archivos) una estructura de directorios es mediante un método recursivo.

Sabemos cómo obtener todos los archivos en un directorio, mediante `os.listdir`. El desafío es identificar cuáles de estos archivos son directorios. Por fortuna, Java sabe cómo lidiar con esto mediante su objeto `File`, que podemos usar con facilidad desde Jython. Al encontrar un directorio, podemos procesarlo justo igual que como con el primer directorio solicitado.

**Programa 144: impresión de todos los nombres de archivos en un árbol de directorios**

```
import os
import java.io.File as File

def imprimirTodosLosArchivos(directorio):
  archivos = os.listdir(directorio)
  for archivo in archivos:
    nombrecompleto = directorio+"/"+archivo
    if esDirectorio(nombrecompleto):
      imprimirTodosLosArchivos(nombrecompleto)
    else:
      print nombrecompleto

def esDirectorio(nombrearchivo):
  estadoarchivo = File(nombrearchivo)
  return estadoarchivo.isDirectory()
```

■

## Cómo funciona

Necesitaremos el módulo `os` de Python y la clase `java.io.File` de Java. Para `imprimirTodosLosArchivos` en un `directorio` especificado, obtenemos una lista de todos los archivos en el directorio usando `os.listdir`. Para cada uno de esos archivos, agregamos el directorio y el nombre de archivo para obtener una ruta completa. Luego preguntamos si es un directorio mediante la función `esDirectorio`. Usamos el objeto `File` de Java para preguntar si es un directorio y luego devolvemos esa respuesta.

Por lo general en este libro, cuando tenemos que evaluar dos cosas, usamos dos instrucciones `if`. Por ende, escribiríamos la primera función anterior de la siguiente forma:

```
def imprimirTodosLosArchivos(directorio):
  archivos = os.listdir(directorio)
  for archivo in archivos:
    nombrecompleto = directorio+"/"+archivo
    if esDirectorio(nombrecompleto):
      imprimirTodosLosArchivos(nombrecompleto)
    if not esDirectorio(nombrecompleto):
      print nombrecompleto
```

Sin embargo, cada evaluación de `esDirectorio` requiere una operación de archivos bastante compleja. Esto implica una cantidad considerable de tiempo del procesador. En vez de hacerlo de manera repetitiva, decimos: "Si *no* es un directorio, o de lo contrario (`else`), hacer esto". De esta forma usamos la instrucción `else`, que no es tan legible como las instrucciones `if` dobles, pero evita que repitamos una evaluación y es, por consecuencia, más eficiente.

Vamos a probar esta función en la carpeta que se muestra en la figura 15.3.

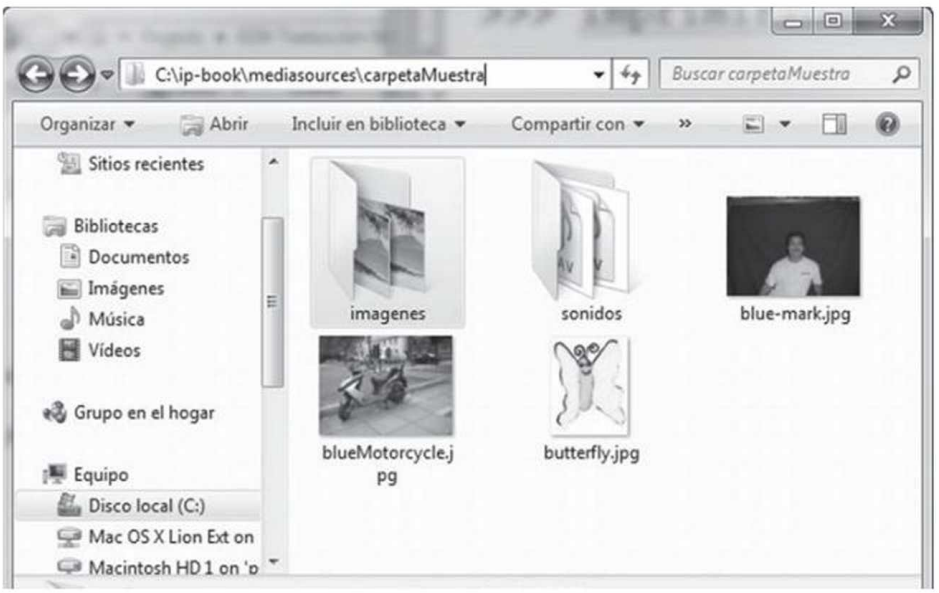



**FIGURA 15.3**
Contenido de la carpeta Muestra.

```
>>> imprimirTodosLosArchivos("C:/ip-book/mediasources/carpetaMuestra")
C:/ip-book/mediasources/carpetaMuestra/blue-mark.jpg
C:/ip-book/mediasources/carpetaMuestra/blueMotorcycle.jpg
C:/ip-book/mediasources/carpetaMuestra/butterfly.jpg
C:/ip-book/mediasources/carpetaMuestra/imagenes/beach-smaller.jpg
C:/ip-book/mediasources/carpetaMuestra/imagenes/beach.jpg
C:/ip-book/mediasources/carpetaMuestra/imagenes/bgframe.jpg
C:/ip-book/mediasources/carpetaMuestra/imagenes/bigben.jpg
C:/ip-book/mediasources/carpetaMuestra/imagenes/blueShrub.jpg
C:/ip-book/mediasources/carpetaMuestra/imagenes/bridge.jpg
C:/ip-book/mediasources/carpetaMuestra/sonidos/bassoon-c4.wav
C:/ip-book/mediasources/carpetaMuestra/sonidos/bassoon-e4.wav
C:/ip-book/mediasources/carpetaMuestra/sonidos/bassoon-g4.wav
```

### 15.4.2 Funciones multimedia recursivas

Podemos pensar en escribir funciones multimedia como `reducirRojo()` en forma recursiva, de la siguiente manera.

**Programa 145: reducción de color rojo en forma recursiva**

```
def reducirRojoR(unaLista):
  if unaLista == []:  # vacía
    return
  setRed(unaLista[0],getRed(unaLista[0])*0.8)
  reducirRojoR(unaLista[1:])
```

■

## Cómo funciona

Si la lista de entrada de píxeles está vacía, entonces nos detenemos (`return`). De lo contrario, tomamos el primer pixel en la lista (`unaLista[0]`) y reducimos su nivel de rojo en un 20% (lo multiplicamos por 0.8). Después llamamos a `reducirRojoR` con el *resto* de la lista (`unaLista[1:]`).

Esta versión se invoca de la siguiente manera: `reducirRojoR(getPixels(imagen))`. *Advertencia:* esto no funciona ni siquiera para imágenes de un tamaño razonable. Python (y el lenguaje Java detrás de Jython) no espera una recursividad de este nivel de profundidad, por lo que se quedará sin memoria. No obstante, sí funciona con imágenes *verdaderamente* pequeñas. En realidad esta versión de `reducirRojo` tiene dos problemas:

- En primer lugar, se ejecuta de manera recursiva para cada pixel. Esto se traduce en cientos de miles de veces.
- En segundo lugar, pasa toda la lista de píxeles para cada llamada. Esto significa, para cada pixel procesado, una copia de todos los píxeles que también están almacenados en memoria y representa una enorme cantidad de esta.

Podemos escribir esta función de una manera distinta, de modo que podamos corregir el segundo problema. Es posible evitar pasar toda la lista de píxeles si declaramos la variable que contiene la lista de píxeles como `global`. Así la variable se comparte entre las funciones.

**Programa 146: función `reducirRojo` recursiva con una variable global**

```
pixelesDeUnaImagen=[]

def reducirRojoR(unaImagen):
  global pixelesDeUnaImagen
  pixelesDeUnaImagen=getPixels(unaImagen)
  reducirRojoPorIndice(len(pixelesDeUnaImagen)-1)

def reducirRojoPorIndice(indice):
  global pixelesDeUnaImagen
  pixel = pixelesDeUnaImagen[indice]
  setRed(pixel, 0.8 * getRed(pixel))
  if indice == 0:  # vacía
    return
  reducirRojoPorIndice(indice - 1)
```

■

## Cómo funciona

Primero creamos la variable `pixelesDeUnaImagen` fuera de cualquier función. Las instrucciones `global` indican a Python que `pixelesDeUnaImagen` debe definirse al nivel de

archivo (módulo), no sólo a nivel local de esta función. La función `reducirRojoR` coloca todos los píxeles en `pixelesDeUnaImagen` y después llama a la función ayudante `reducirRojoPorIndice` con el último índice en la lista de píxeles de la imagen, la longitud (`len`) de la lista menos uno. La función `reducirRojoPorIndice` obtiene el pixel en el índice especificado y reduce su nivel de rojo al 80%. Si el índice es cero, el primero de todos los píxeles, sólo ejecutamos `return` y nos detenemos. Si no es cero, seguimos reduciendo el nivel de rojo por uno menos que el índice actual.

El problema es que esta versión aún se ejecuta en forma recursiva para cada pixel. Incluso con imágenes relativamente pequeñas (640 × 480), la pila se desborda antes de completar el procesamiento de la imagen. En general es difícil realizar el procesamiento pixel por pixel, tal vez hasta sea algo imposible de hacer en Jython.

## RESUMEN DE PROGRAMACIÓN

A continuación mostramos algunas de las funciones y piezas de programación que vimos en este capítulo.

## PROGRAMACIÓN FUNCIONAL

| | |
|---|---|
| `apply` | Recibe como entrada una función y una lista como entrada para esa función, donde la lista tiene tantos elementos como los que la función reciba como entrada. Llama a la función con base en la entrada. |
| `map` | Recibe como entrada una función y una lista de varias entradas a esa función. Llama a la función sobre cada una de las entradas y devuelve una lista de las salidas (valores de retorno). |
| `filter` | Recibe como entrada una función y una lista de varias entradas a esa función. Llama a la función sobre cada uno de los elementos de la lista y devuelve el *elemento de entrada* si la función devuelve verdadero (un valor distinto de cero) para ese elemento. |
| `reduce` | Recibe como entrada una función, la cual recibe dos entradas y una lista de varias entradas a esa función. La función se aplica a los primeros dos elementos de la lista; después el resultado de eso se usa como entrada con el siguiente elemento de la lista, luego el resultado de eso se usa como entrada con el *siguiente* elemento de la lista, y así en lo sucesivo. Al final se devuelve el resultado total. |
| `else:` | Después de una instrucción `if`, ejecuta el siguiente bloque de código sólo si la prueba en la instrucción `if` es *falsa*. |
| `global` | Las variables listadas después de la instrucción `global` hacen referencia a una variable creada antes de esta función, a nivel de archivo (o módulo). Esto nos permite compartir una referencia a un objeto, en vez de duplicarlo. |

## PROBLEMAS

15.1 Ahora veamos un acertijo. Suponga que tiene seis bloques. Uno de ellos pesa más que el otro. Tiene una pesa a su disposición, pero sólo puede usarla dos veces. Encuentre el bloque más pesado. (a) Escriba su proceso como un algoritmo. (b) ¿Qué tipo de búsqueda es esta?

15.2 Escriba una función para calcular el valor de Fibonacci de un número. El número Fibonacci de 0 es 0 y el de 1 es 1. El número Fibonacci de n es $Fib(n) = Fib(n-2) + Fib(n-1)$.

15.3 Escriba una función para convertir grados Celsius a Fahrenheit.

15.4 Escriba una función para calcular el índice de masa corporal de una persona, dados su peso y altura.

15.5 Escriba una función para calcular una propina del 20 por ciento.

15.6 Busque el triángulo de Sierpinski. Escriba una función recursiva para crear el triángulo de Sierpinski.

15.7 Busque el copo de nieve de Koch. Escriba una función recursiva para crear el copo de nieve de Koch.

15.8 Cambie la función `convertirCabelloRojo` de programación funcional en el programa 141 (página 362) de modo que esté escrita en *sólo* una línea, registrando las funciones utilitarias como funciones `lambda`.

15.9 Describa lo que hace esta función. Pruebe con distintos números como entrada.

```
def prueba(num):
  if num > 0:
    return prueba(num-1)
  else:
    return 0
```

15.10 Describa lo que hace esta función. Pruebe con distintos números como entrada.

```
def prueba(num):
  if num > 0:
    return prueba(num-1) + num
  else:
    return 0
```

15.11 Describa lo que hace esta función. Pruebe con distintos números como entrada.

```
def prueba(num):
  if num > 0:
    return prueba(num-1) * num
  else:
    return 0
```

15.12 Describa lo que hace esta función. Pruebe con distintos números como entrada.

```
def prueba(num):
  if num > 0:
    return num - prueba(num-1)
  else:
    return 0
```

15.13 Describa lo que hace esta función. Pruebe con distintos números como entrada.

```
def prueba(num):
  if num > 0:
    return prueba(num-2) * prueba(num-1)
  else:
    return 0
```

15.14 Escriba un método recursivo para listar todos los archivos en un directorio y en todos los subdirectorios.

15.15 Pruebe a escribir `arribaAbajo()`:

```
>>> arribaAbajo("Hola")
Hola
Hol
Ho
H
Ho
Hol
Hola
```

Intente escribir la función *tanto* recursiva como no recursiva. ¿Cuál es más fácil? ¿Por qué?

15.16 Pruebe a escribir algunos de nuestros ejemplos de sonidos e imágenes de los primeros capítulos mediante *programación funcional*, usando estructuras como `filter` y `map`.

CAPÍTULO 16

# Programación orientada a objetos

16.1 HISTORIA DE LOS OBJETOS
16.2 TRABAJO CON TORTUGAS
16.3 ENSEÑAR NUEVOS TRUCOS A LAS TORTUGAS
16.4 UNA PRESENTACIÓN ORIENTADA A OBJETOS
16.5 MEDIOS ORIENTADOS A OBJETOS
16.6 LA CAJA JOE
16.7 ¿POR QUÉ OBJETOS?

**Objetivos de aprendizaje del capítulo**

- Usar la programación orientada a objetos para que los programas sean más fáciles de desarrollar en equipos, más robustos y fáciles de depurar.
- Comprender las características de los programas orientados a objetos, como polimorfismo, encapsulamiento, herencia y agregación.
- Poder elegir entre distintos estilos de programación para distintos fines.

## 16.1 HISTORIA DE LOS OBJETOS

El estilo más común de programación en la actualidad es la **programación orientada a objetos**. Vamos a definirla en contraste con la programación por procedimientos que hemos realizado hasta ahora.

En las décadas de 1960 y 1970, la programación por procedimientos era la forma dominante de programación. Las personas usaban la *abstracción procedural* y definían muchas funciones en niveles altos y bajos. Además reutilizaban sus funciones en donde fuera posible. Esto funcionaba bastante bien, hasta cierto punto. A medida que los programas se volvían grandes y complejos, con muchos programadores trabajando en ellos al mismo tiempo, la programación por procedimientos empezó a sufrir conflictos.

Los programadores tuvieron problemas con los conflictos de los procedimientos. Las personas escribían programas que modificaban datos en formas que otras personas no esperaban. Usaban los mismos nombres para las funciones y descubrían que su código no podía integrarse en un programa extenso.

También hubo problemas al *pensar* sobre los programas y las tareas que se suponía debían realizar. Los procedimientos tratan sobre los *verbos*: dicen a la computadora que haga esto

o aquello. Pero no está claro si esa es la forma en que las personas piensan mejor sobre los problemas.

La programación orientada a objetos es *programación orientada a sustantivos*. Para crear un programa orientado a objetos, primero hay que pensar qué son los sustantivos en el *dominio* del problema: ¿qué son las personas y cosas que forman parte de este problema y su solución? Al proceso de identificar los objetos, lo que cada uno de ellos sabe (respecto al problema) y lo que tiene que hacer se le denomina **análisis orientado a objetos**.

La programación de una manera orientada a objetos significa que se definen variables (**variables de instancia**) y funciones (**métodos**) *para los objetos*. En la mayoría de los lenguajes orientados a objetos, los programas tienen muy pocas (o incluso *ninguna*) funciones o variables globales: cosas que pueden utilizarse en cualquier parte. En Smalltalk, el lenguaje de programación orientada a objetos original, los objetos *sólo* pueden realizar cosas si piden a otro objeto que realice las cosas por medio de sus métodos. Adele Goldberg, una pionera de la programación orientada a objetos, nombró esto como "Preguntar sin tocar". No podemos "tocar" los datos y hacer todo lo que queramos con ellos; lo que hacemos es "pedir" a los objetos que manipulen sus datos por medio de sus métodos. Esto es un buen objetivo, incluso en lenguajes como Python o Java, en donde los objetos *pueden* manipular los datos de otros objetos en forma directa.

El término *programación orientada a objetos* lo inventó Alan Kay, un brillante personaje multidisciplinario: posee títulos universitarios en matemáticas y biología, un doctorado en ciencias computacionales y ha sido guitarrista de jazz profesional. En 2004 recibió el Premio Turing de la ACM, algo así como el Premio Nobel de la computación. Kay vio la programación orientada a objetos como una forma de desarrollar software que en verdad pudiera escalar a sistemas grandes. Describió los objetos como algo parecido a las *células* biológicas que trabajan juntas en formas bien definidas para hacer que todo el organismo trabaje. Al igual que las células, los objetos podrían:

- Ayudar a lidiar con la *complejidad* al distribuir la responsabilidad de las tareas entre muchos objetos, en vez de un programa extenso.
- Apoyar la *robustez* al hacer que los objetos trabajen relativamente en forma independiente.
- Apoyar la *reutilización* ya que cada objeto provee *servicios* a otros objetos (tareas que el objeto realiza para otros objetos, accesibles por medio de sus métodos), al igual que los objetos en el mundo real.

La noción de partir de sustantivos es parte de la visión de Kay. El **software**, dijo él, es en realidad una *simulación* del mundo. Al hacer que el software *modele* el mundo, nos queda más claro cómo crear software. Analizamos el mundo y la forma en que funciona, para luego copiar eso en el software. Las cosas en el mundo *conocen* cosas; éstas se convierten en **variables de instancia**. Las cosas en el mundo pueden *hacer* cosas; éstas se convierten en **métodos**.

Cabe mencionar que ya hemos estado usando objetos. Las imágenes, los sonidos, las muestras y los colores son todos objetos. Nuestras listas de píxeles y muestras son ejemplos de *agregación*, que significa crear colecciones de objetos. Las funciones que hemos utilizado en realidad sólo cubren los métodos subyacentes. Podemos llamar a los métodos de los objetos en forma directa, lo cual haremos más adelante en el capítulo.

## 16.2 TRABAJO CON TORTUGAS

Seymour Papert en el MIT utilizó tortugas robot para ayudar a los niños a pensar cómo especificar procedimientos a finales de la década de 1960. La tortuga tenía una pluma en su parte media, que podía levantar y bajar para dejar un rastro de sus movimientos. A medida que empezaron a surgir las pantallas de gráficos, utilizó una tortuga virtual en una pantalla de computadora en vez de una computadora robótica.

Parte del soporte de medios en JES ofrece objetos de tortugas de gráficos. Las tortugas son una excelente introducción a las ideas de los objetos. Manipulamos objetos tortuga que se desplazan por un mundo. Las tortugas saben cómo moverse y girar. Tienen una pluma en su parte media que deja un rastro para mostrar sus movimientos. El mundo lleva la cuenta de las tortugas que viven en él.

### 16.2.1 Clases y objetos

¿Cómo sabe la computadora lo que quiere decir una tortuga y un mundo? Tenemos que definir qué es una tortuga, qué es lo que conoce y lo que puede hacer. Tenemos que definir qué es un mundo, lo que conoce y lo que puede hacer. Para hacer esto en Python definimos clases. Una clase define lo que los tipos u objetos (instancias) de esa clase saben y pueden hacer. El paquete de medios para JES define clases que a su vez definen lo que queremos decir con una tortuga y un mundo.

Los programas orientados a objetos consisten de objetos. Creamos objetos a partir de clases. La clase sabe lo que cada objeto de esa clase necesita vigilar y lo que debe ser capaz de hacer. Podemos considerar una clase como una fábrica de objetos. La fábrica puede crear muchos objetos. Una clase es también como un molde para galletas. Podemos hacer muchas galletas a partir de un molde y todas tendrán la misma forma. O podemos considerar la clase como un plano de construcción y los objetos como las casas que podemos crear con base en ese plano.

Para crear e inicializar un mundo, utilizamos `makeWorld()`. Para crear un objeto tortuga, utilizamos `makeTurtle(world)`. Esta función es muy similar a `makePicture` y `makeSound`: hay cierto patrón aquí, pero introduciremos uno nuevo, una sintaxis más estándar de Python en unos minutos. Ahora vamos a crear un nuevo objeto mundo.

```
>>> makeWorld()
```

Esta instrucción crea un objeto mundo y visualiza una ventana que muestra ese mundo. Comienza sólo como una imagen blanca en un marco titulado "World". Pero no podemos hacer referencia a él, puesto que no le hemos asignado un nombre.

En este caso nombraremos `tierra` al objeto mundo que acabamos de crear, y luego vamos a crear un objeto tortuga en el mundo llamado `tierra`. A ese objeto tortuga lo llamaremos `tina`.

```
>>> tierra = makeWorld()
>>> tina = makeTurtle(tierra)
>>> print tina
No name turtle at 320, 240 heading 0.0.
```

El objeto tortuga aparece en el centro del mundo (320 240) y con dirección hacia el norte (*una orientación de* 0) (figura 16.1). Todavía no hemos asignado un nombre a la tortuga.

**FIGURA 16.1**
Creación de una tortuga en el mundo.

JES nos permite crear muchas tortugas. Cada nueva tortuga aparecerá en el centro del mundo.

```
>>> sue = makeTurtle(tierra)
```

### 16.2.2 Envío de mensajes a los objetos

Para pedir a la tortuga que haga cosas enviamos un mensaje al objeto tortuga, lo cual también consideramos como llamar a un método de un objeto. Para ello usamos la *notación punto*. En esta notación, para pedir a un objeto que haga algo especificamos el nombre de ese objeto y luego un '.', seguido de la función a ejecutar (`nombre.función(listaParámetros)`). En la sección 10.3.1 vimos la notación punto con las cadenas.

```
>>> tina.forward()
>>> tina.turnRight()
>>> tina.forward()
```

Observe que sólo se mueve la tortuga a la que pedimos realizar las acciones (figura 16.2). Podemos hacer que la otra se mueva si también le pedimos que haga cosas.

```
>>> sue.turnLeft()
>>> sue.forward(50)
```

Cabe mencionar que cada tortuga tiene un color distinto (figura 16.3). Como podemos ver, las tortugas saben cómo girar a la izquierda y a la derecha, mediante el uso de `turnLeft()` y `turnRight()`. También pueden avanzar en la dirección en la que se encuentran

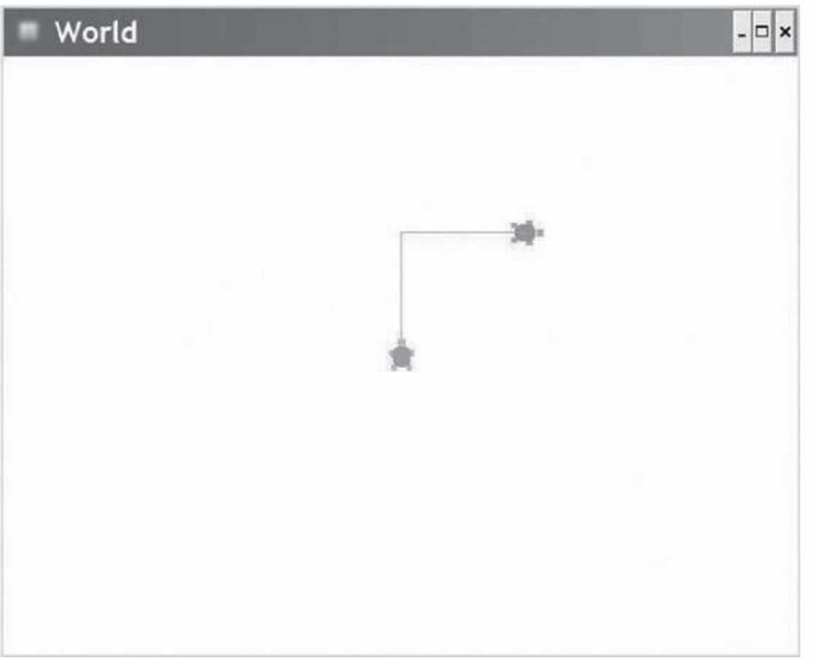



**FIGURA 16.2**
Se le pide a una tortuga que se mueva y gire mientras la otra se queda quieta.

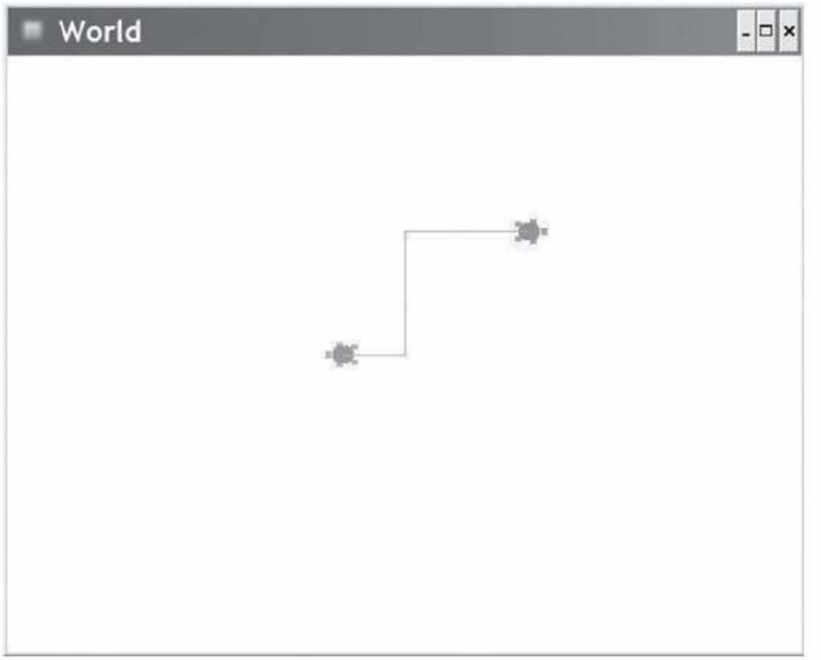



**FIGURA 16.3**
Después de que se mueve la segunda tortuga.

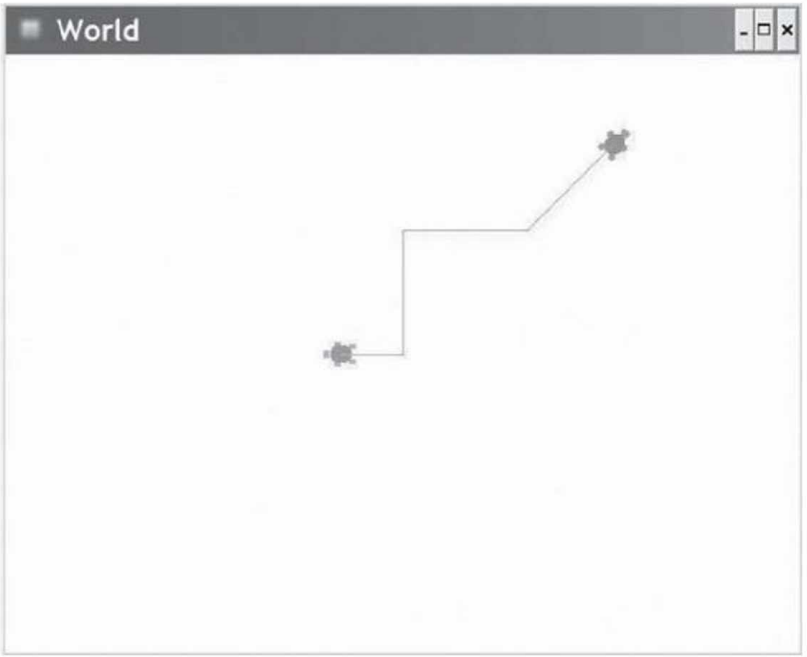



**FIGURA 16.4**
Girar cierta cantidad especificada (–45).

orientadas mediante `forward()`. De manera predeterminada avanzan 100 píxeles, pero también podemos especificar cuántos píxeles deben avanzar: `forward(50)`. Las tortugas saben cómo girar cierto número de grados también. Los montos positivos giran a la tortuga a la derecha y los negativos a la izquierda (figura 16.4).

```
>>> tina.turn(-45)
>>> tina.forward()
```

### 16.2.3 Los objetos controlan su estado

En la programación orientada a objetos, enviamos mensajes para pedir a los objetos que hagan cosas. Los objetos pueden rehusarse a hacer lo que usted les pide. Un objeto *debe* rehusarse si usted le pide algo que haga que sus datos sean incorrectos. El mundo en el que se encuentran las tortugas es de 640 píxeles de ancho por 480 píxeles de alto. ¿Qué ocurre si usted trata de decirle a la tortuga que vaya más allá del fin del mundo?

```
>>> mundo1 = makeWorld()
>>> tortuga1 = makeTurtle(mundo1)
>>> tortuga1.forward (400)
>>> print tortuga1
No name turtle at 320, 0 heading 0.0.
```

Las tortugas se posicionan por primera vez en (320, 240) con orientación hacia el norte (arriba). En el mundo la posición superior izquierda es (0, 0). El eje x aumenta a la derecha y el eje y aumenta hacia abajo. Al pedir a la tortuga que avance 400, le estamos pidiendo que vaya a (320, 240 –400), lo que produciría una posición de (320, –160). Pero la tortuga

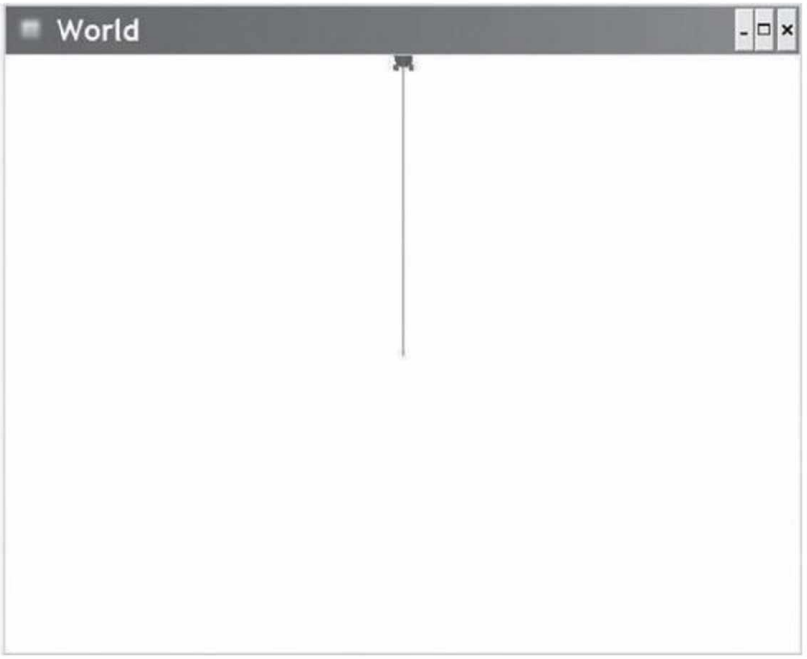



**FIGURA 16.5**
Una tortuga atascada en la orilla del mundo.

se rehúsa a salir del mundo y mejor se detiene cuando el centro de la tortuga está en (320, 0) (figura 16.5). Esto significa que no perderemos de vista a ninguna de nuestras tortugas.

El objetivo de este ejercicio es mostrar al lector cómo es que los métodos controlan el acceso a los datos del objeto. Si no desea que las variables tengan ciertos valores en sus datos, puede controlar esta opción por medio de los métodos. Estos métodos sirven como la puerta de enlace y como guardián para los datos del objeto.

Las tortugas pueden hacer muchas otras cosas además de avanzar y girar. Como tal vez haya notado, cuando las tortugas se mueven dibujan una línea que es del mismo color que la tortuga. Puede pedir a la tortuga que levante la pluma mediante el uso de `penUp()`. Puede pedir a la tortuga que baje la pluma usando `penDown()`. Puede pedir a la tortuga que se mueva a una posición específica si utiliza `moveTo(x,y)`. Si la pluma está abajo cuando usted le pida que se mueva a una nueva posición, la tortuga dibujará una línea desde la posición anterior hasta la nueva posición (figura 16.6).

```
>>> mundoX = makeWorld()
>>> tortugaX = makeTurtle(mundoX)
>>> tortugaX.penUp()
>>> tortugaX.moveTo(0,0)
>>> tortugaX.penDown()
>>> tortugaX.moveTo(639,479)
```

Para cambiar el color de una tortuga use `setColor(color)`. Para que deje de dibujar use `setVisible(false)`. Puede cambiar el grosor de la pluma mediante `setPenWidth(anchura)`.

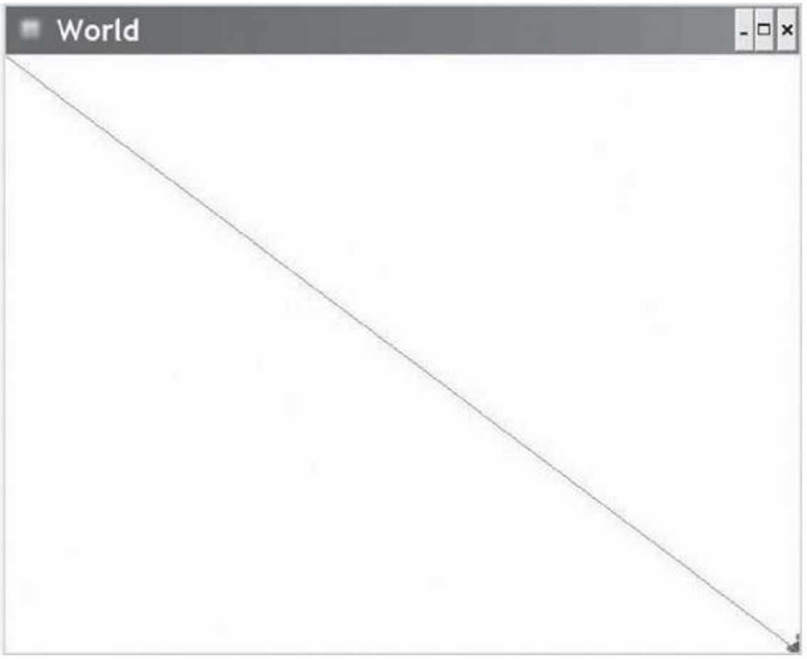



**FIGURA 16.6**
Uso de la tortuga para dibujar una línea diagonal.

## 16.3 ENSEÑAR NUEVOS TRUCOS A LAS TORTUGAS

Ya definimos una clase tortuga (`Turtle`) por usted. Pero ¿qué tal si desea crear su propio tipo de tortuga y enseñarle nuevos trucos? Podemos crear un nuevo tipo de tortuga que entienda cómo hacer todas las cosas que una tortuga sabe hacer, y tenemos la posibilidad de agregar nueva funcionalidad. A esto se le conoce como crear una *subclase*. Así como los hijos heredan el color de los ojos de sus padres, nuestra subclase *heredará* todas las cosas que las tortugas conocen y pueden hacer. A una subclase también se le conoce como clase *hija* y la clase de la que hereda se llama clase `padre` o `superclase`.

Vamos a llamar a nuestra subclase `TortugaChica`. Agregaremos un *método* que permita a nuestra tortuga dibujar un cuadrado. Los métodos se definen justo igual que las funciones, sólo que están dentro de la clase. En Python los métodos *siempre* reciben como entrada una referencia al objeto de la clase en la que se invocó a ese método (por lo general se llama `self`). Para dibujar un cuadrado nuestra tortuga girará a la derecha y avanzará 4 veces. Cabe mencionar que heredamos la habilidad de girar a la derecha y avanzar de la clase `Turtle`.

**Programa 147: definición de una subclase**

```
class TortugaChica(Turtle):

  def dibujarCuadrado(self):
    for i in range(0,4):
      self.turnRight()
      self.forward()
```

■

Como `TortugaChica`es un tipo de tortuga (`Turtle`), podemos usarla en forma muy parecida. Sólo que necesitaremos crear esa `TortugaChica` de una nueva forma. Hemos estado usando `makePicture`, `makeSound`, `makeWorld` y `makeTurtle` para crear nuestros objetos. Éstas son funciones que creamos para facilitar la creación de esos objetos. Pero, la verdadera forma de crear un nuevo objeto en Python es mediante el uso de `NombreClase(listaParámetros)`. Para crear un mundo podemos usar `objMundo = World()` y para crear una `TortugaChica` podemos usar `objTortuga = TortugaChica(objMundo)`.

```
>>> tierra = World()
>>> audaz = TortugaChica(tierra)
>>> audaz.dibujarCuadrado()
```

Ahora nuestra `TortugaChica` sabe cómo dibujar un cuadrado (figura 16.7). Pero sólo puede dibujar cuadrados con un tamaño de 100. Sería agradable poder dibujar cuadrados de distintos tamaños. Podemos agregar otra función que reciba un parámetro y especifique la anchura del cuadrado.

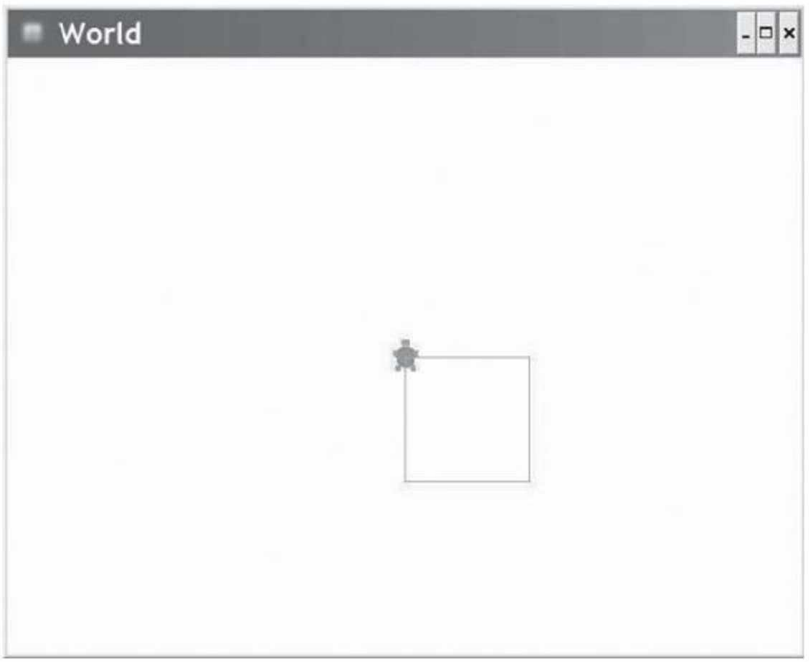



**FIGURA 16.7**
Dibujo de un cuadrado con nuestra TortugaChica.

**Programa 148: definición de una subclase**

```
class TortugaChica(Turtle):

  def dibujarCuadrado(self):
    for i in range(0,4):
```

```
    self.turnRight()
    self.forward()

def dibujarCuadrado(self, ancho):
  for i in range(0,4):
    self.turnRight()
    self.forward(ancho)
```

■

Puede usar esto para dibujar cuadrados de distintos tamaños (figura 16.8).

```
>>> marte = World()
>>> tina = TortugaChica(marte)
>>> tina.dibujarCuadrado(30)
>>> tina.dibujarCuadrado(150)
>>> tina.dibujarCuadrado(100)
```

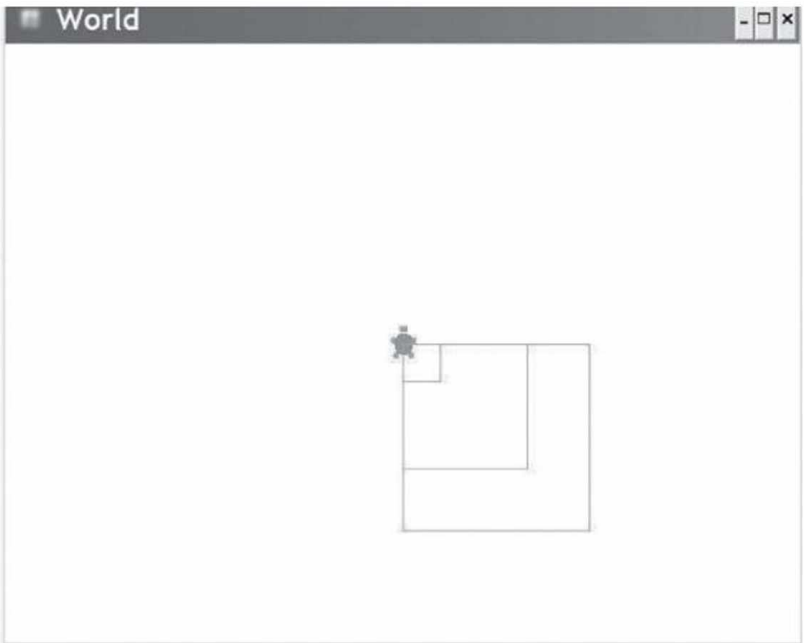



**FIGURA 16.8**
Cómo dibujar cuadrados de distintos tamaños.

### 16.3.1 Redefinición de un método existente de la clase tortuga

Una subclase puede redefinir un método que ya exista en la superclase. Podríamos hacer esto para crear una forma especializada del método existente.

A continuación veremos la clase `TortugaConfundida`, que redefine los métodos `forward` y `turn` de modo que ejecute los métodos `forward` y `turn` de la clase `Turtle`, pero

con una cantidad aleatoria. Podemos usarla como una tortuga normal, sólo que no avanzará ni girará tanto como se le solicite. El siguiente ejemplo hará que `tontita` avance alrededor de 100 y gire cerca de 90 grados.

```
>>> pluton = World()
>>> tontita = TortugaConfundida(pluton)
>>> tontita.forward(100)
>>> tontita.turn(90)
```

**Programa 149: `TortugaConfundida`, que avanza y gira una cantidad aleatoria**

```
import random
class TortugaConfundida(Turtle):
   def forward(self,num):
     Turtle.forward(self,int(num*random.random()))
   def turn(self,num):
     Turtle.turn(self,int(num*random.random()))
```

■

### Cómo funciona

Declaramos la clase `TortugaConfundida` como una subclase de `Turtle`. Definimos nuevos métodos en `TortugaConfundida`: `forward` y `turn`. Al igual que cualquier otro método, reciben `self` y cualquiera que sea la entrada para el método. En estos casos, la entrada para ambos métodos es un número `num`.

Lo que queremos hacer es llamar a la *superclase* (en este caso, `Turtle`) y pedirle que ejecute las funciones `forward` y `turn` normales, pero con la entrada multiplicada por un número aleatorio. El cuerpo de cada método es una sola línea, pero bastante complicada.

- Tenemos que decirle a Python que invoque de manera explícita el método `forward` de `Turtle`.
- Tenemos que pasar `self`, de modo que se utilicen y actualicen los datos del objeto correcto.
- Multiplicamos el `num` de entrada por `random.random()`, pero necesitamos convertirlo en un entero (mediante `int`). El número aleatorio devuelto estará entre 0 y 1 (un número de punto flotante), pero necesitamos un entero para `forward` y `turn`.

## 16.3.2 Uso de tortugas para realizar más acciones

Las tortugas tienen varios métodos que permiten realizar efectos gráficos interesantes. Por ejemplo, las tortugas están conscientes de las otras tortugas a su alrededor. Pueden voltear hacia otra tortuga mediante `turnToFace(otraTortuga)` para cambiar su orientación de modo que la tortuga esté "viendo" a otra tortuga (de modo que, si sigue avanzando, alcanzará a la otra tortuga). En el siguiente ejemplo establecemos cuatro tortugas (`al`, `bo`, `cy` y `di`) en cuatro esquinas de un cuadrado para después hacer que cada una se mueva en forma repetida hacia la tortuga que está a su izquierda. El resultado se muestra en la figura 16.9.

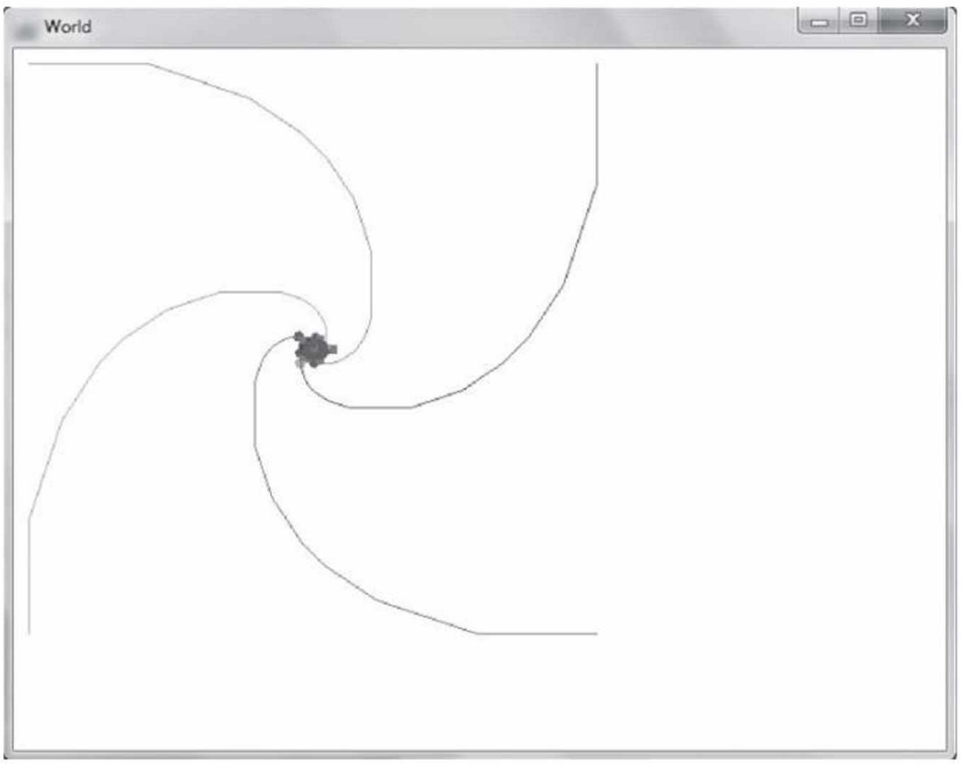



**FIGURA 16.9**
Cuatro tortugas persiguiéndose entre sí.

**Programa 150: persecución de tortugas**

```
def perseguir():
  # Establecer las cuatro tortugas
  tierra = World()
  al = Turtle(tierra)
  bo = Turtle(tierra)
  cy = Turtle(tierra)
  di = Turtle(tierra)
  al.penUp()
  al.moveTo(10,10)
  al.penDown()
  bo.penUp()
  bo.moveTo(10,400)
  bo.penDown()
  cy.penUp()
  cy.moveTo(400,10)
  cy.penDown()
  di.penUp()
  di.moveTo(400,400)
  di.penDown()
  # Ahora, perseguirse durante 300 pasos
  for i in range(0,300):
```

```
    perseguirTortuga(al,cy)
    perseguirTortuga(cy,di)
    perseguirTortuga(di,bo)
    perseguirTortuga(bo,al)

def perseguirTortuga(t1,t2):
  t1.turnToFace(t2)
  t1.forward(4)
```

■

### Cómo funciona

Aquí la función principal es `perseguir()`. Las primeras líneas crean un mundo y a las cuatro tortugas. Después colocan a cada una de ellas en las esquinas (10, 10), (10, 400), (400, 400) y (400, 10). Durante 300 pasos (un número relativamente arbitrario), se instruye a cada tortuga para que "persiga" (`perseguirTortuga`) a la que está a su lado, en sentido de las manecillas del reloj. Así, la tortuga que empieza en (10, 10) (`al`) recibe instrucciones para perseguir a la tortuga que empieza en (400, 10) (`cy`). Perseguir significa que la primera tortuga debe voltear a ver a la segunda tortuga y luego avanzar cuatro pasos (pruebe con distintos valores; a nosotros nos gustó más el efecto de 4). En un momento dado, las tortugas realizarán una espiral hacia el centro.

Estas funciones son valiosas para crear *simulaciones*. Imagine que tenemos tortugas color café actuando como ciervos y tortugas grises como lobos. Los lobos voltearían hacia los ciervos (`turnToFace`) al momento de verlos y empezarían a perseguirlos. Para escapar, los ciervos podrían voltear a ver (`turnToFace`) a un lobo acechante, para luego girar 180 grados y correr. Las simulaciones son uno de los usos más poderosos e intuitivos de las computadoras.

**Idea de ciencias computacionales: los parámetros funcionan un poco distinto con los objetos**

Básicamente, al invocar una función y pasarle un número como entrada, la variable parámetro (la variable local que acepta la entrada) recibe una *copia* de ese número. Al modificar la variable local no se modifica la variable de entrada.

Analicemos la función `perseguirTortuga`. Al invocar la función con `perseguirTortuga(al,cy)`, *cambiamos* la posición y la orientación de la tortuga llamada `al`. ¿Por qué es tan diferente? En realidad no lo es. En sí, la variable `al` no contiene una tortuga, sino una *referencia* a ésta. Considérelo como una dirección (en memoria) de la ubicación del objeto tortuga. Si realiza una copia de una dirección, esa dirección aún hace referencia al mismo lugar. Se está manipulando la misma tortuga dentro y fuera de la función. Aún no podemos lograr que `al` haga referencia a un nuevo objeto desde el interior de una función como `perseguirTortuga`. Sólo podemos modificar el objeto al que al hace referencia.

■

Las tortugas también saben cómo soltar (`drop`) imágenes. Cuando una tortuga suelta una imagen, permanece en la esquina superior izquierda de la imagen, sin importar la orientación de la tortuga (vea la figura 16.10).

```
>>> # Seleccioné la imagen Barbara.jpg para este ejemplo
>>> p=makePicture(pickAFile())
>>> # Observe que aquí creamos los objetos mundo y tortuga
>>> tierra=World()
>>> tortuga=Turtle(tierra)
>>> tortuga.drop(p)
```

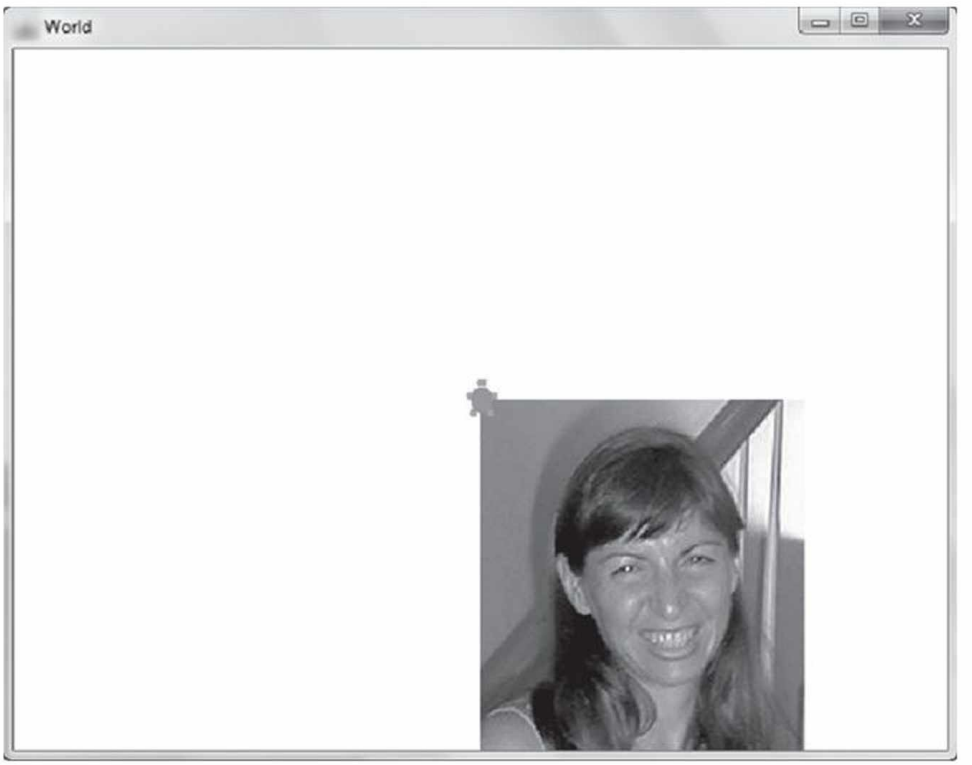



**FIGURA 16.10**
Cómo soltar una imagen en un mundo.

También podemos colocar tortugas en imágenes, al igual que en instancias de mundos. Al colocar una tortuga en una imagen, su cuerpo no aparece de manera predeterminada (aunque podemos hacerla visible) de modo que no se confunda con la imagen. Pero la pluma está abajo y podemos dibujar. Colocar una tortuga en una imagen significa que podemos crear gráficos interesantes encima de imágenes existentes, o usar imágenes existentes en nuestras manipulaciones de tortugas.

Una de nuestras técnicas favoritas es girar una imagen: hacer que la tortuga se mueva un poco, gire un poco, suelte una copia de la imagen, y luego repita estos pasos. En la figura 16.11 podrá ver un ejemplo. A continuación le mostramos el código con el que se creó esa figura. Invocamos la función con la misma imagen de Barb del ejemplo anterior, `show(girarUnaImagen(p))`.

**Programa 151: cómo girar una imagen al soltarla de una tortuga girando**

```
def girarUnaImagen(unaimag):
  lienzo = makeEmptyPicture(640,480)
  ted = Turtle(lienzo)
  for i in range(0,360):
    ted.drop(unaimag)
    ted.forward(10)
    ted.turn(20)
  return lienzo
```

■

**FIGURA 16.11**
Cómo soltar una imagen sobre una imagen, mientras se mueve y gira.

## 16.4 UNA PRESENTACIÓN ORIENTADA A OBJETOS

Vamos a usar técnicas orientadas a objetos para crear una presentación con diapositivas. Digamos que queremos mostrar una imagen, después reproducir cierto sonido y luego esperar hasta que termine el sonido antes de avanzar a la siguiente imagen. Usaremos la función (que mencionamos en uno de los primeros capítulos del libro) *blockingPlay()*, que reproduce un sonido y espera a que termine antes de ejecutar la siguiente instrucción.

**Programa 152: presentación con diapositivas como una sola función grande**

```
def reproducirPresentacion():
  imag = makePicture(getMediaPath("barbara.jpg"))
  sonido = makeSound(getMediaPath("bassoon-c4.wav"))
  show(imag)
  blockingPlay(sonido)
  imag = makePicture(getMediaPath("beach.jpg"))
  sonido = makeSound(getMediaPath("bassoon-e4.wav"))
  show(imag)
  blockingPlay(sonido)
  imag = makePicture(getMediaPath("church.jpg"))
  sonido = makeSound(getMediaPath("bassoon-g4.wav"))
```

```
show(imag)
blockingPlay(sonido)
imag = makePicture(getMediaPath("jungle2.jpg"))
sonido = makeSound(getMediaPath("bassoon-c4.wav"))
show(imag)
blockingPlay(sonido)
```

■

Éste no es un programa muy bueno desde ninguna perspectiva. Desde la perspectiva de programación por procedimientos, aquí hay una cantidad exagerada de código duplicado. Sería agradable deshacernos de él. Desde una perspectiva orientada a la programación, deberíamos tener objetos diapositivas.

Como dijimos antes, los objetos constan de dos partes. Los objetos *conocen* cosas: éstas se convierten en *variables de instancia*. Los objetos pueden *hacer* cosas: éstas se convierten en *métodos*. Vamos a acceder a ambos elementos mediante el uso de la notación punto.

Entonces ¿qué es lo que conoce una diapositiva? Conoce tanto su *imagen* como su *sonido*. ¿Qué puede hacer una diapositiva? Puede *mostrarse* a sí misma, al mostrar su imagen y reproducir su sonido.

Para definir un objeto diapositiva en Python (y en muchos otros lenguajes de programación orientados a objetos, incluyendo Java y C++), debemos definir una **clase** llamada `Diapositiva`. Ya vimos algunas definiciones de clases. Ahora lo repasaremos de nuevo, con lentitud: vamos a crear una clase desde cero.

Como ya vimos antes, una clase define las variables de instancia y los métodos para un conjunto de objetos; es decir, lo que cada objeto de esa clase conoce y puede hacer. Cada objeto de la clase es una *instancia* de ella. Para crear varias diapositivas vamos a crear múltiples instancias de la clase `Diapositiva`. Esto es lo que se conoce como agregación: colecciones de objetos, así como nuestros cuerpos podrían crear varias células del riñón o del corazón, cada una de las cuales sabe cómo hacer ciertas tareas.

Para crear una clase en Python, empezamos con:

```
class Diapositiva:
```

Después de esta instrucción se colocan con sangría los métodos para crear nuevas diapositivas y reproducirlas. Vamos a agregar un método llamado *mostrar()* a nuestra clase `Diapositiva`:

```
class Diapositiva:
  def mostrar(self):
      show(self.imagen)
      blockingPlay(self.sonido)
```

Para crear nuevas instancias, invocamos el nombre de la clase como una función. Para definir nuevas variables de instancia sólo tenemos que asignarlas. Ahora veamos cómo crear una diapositiva y proporcionarle tanto una imagen como un sonido.

```
>>> diapositiva1=Diapositiva()
>>> diapositiva1.imagen = makePicture(getMediaPath("barbara.jpg"))
>>> diapositiva1.sonido = makeSound(getMediaPath("bassoon-c4.wav"))
>>> diapositiva1.mostrar()
```

La función `diapositiva1.mostrar()` muestra la imagen y reproduce el sonido. ¿Qué es esto de `self`? Al ejecutar `objeto.método()`, Python busca el método en la clase del objeto y después lo invoca, usando el objeto de instancia como *entrada*. El estilo de Python es

llamar a esta variable de entrada `self` (puesto que es el objeto en sí mismo). Como tenemos el objeto en la variable `self`, podemos entonces acceder a su imagen y a su sonido al decir `self.imagen` y `self.sonido`.

Pero esto es aún bastante difícil de usar si tenemos que establecer todas las variables desde el área de comandos. ¿Cómo podríamos hacerlo más sencillo? ¿Qué tal si pudiéramos pasar el sonido y la imagen de las diapositivas como *entradas* para la clase `Diapositiva`, como si la clase fuera una función verdadera? Podemos hacerlo si definimos algo que se conoce como **constructor**.

Para crear nuevas instancias con algunas entradas, debemos definir una función llamada `__init__`. Se escribe "guión bajo-guión bajo-i-n-i-t-guión bajo-guión bajo". Éste es el nombre predefinido en Python para un método que *inicializa* nuevos objetos. Nuestro método `__init__` necesita tres entradas: la instancia en sí (porque todos los métodos la reciben), una imagen y un sonido.

**Programa 153: Una clase `Diapositiva`**

```
class Diapositiva:
  def __init__(self,archivoImagen,archivoSonido):
      self.imagen = makePicture(archivoImagen)
      self.sonido = makeSound(archivoSonido)

  def mostrar(self):
      show(self.imagen)
      blockingPlay(self.sonido)
```

■

Podemos usar nuestra clase `Diapositiva` para definir una diapositiva de la siguiente forma.

**Programa 154: Reproducción de una presentación con diapositivas, usando nuestra clase `Diapositiva`**

```
def reproducirPresentacion2():
    archImag = getMediaPath("barbara.jpg")
    archSon = getMediaPath("bassoon-c4.wav")
    diapositiva1 = Diapositiva(archImag,archSon)
    archImag = getMediaPath("beach.jpg")
    archSon = getMediaPath("bassoon-e4.wav")
    diapositiva2 = Diapositiva(archImag,archSon)
    archImag = getMediaPath("church.jpg")
    archSon = getMediaPath("bassoon-g4.wav")
    diapositiva3 = Diapositiva(archImag,archSon)
    archImag = getMediaPath("jungle2.jpg")
    archSon = getMediaPath("bassoon-c4.wav")
    diapositiva4 = Diapositiva(archImag,archSon)
    diapositiva1.mostrar()
    diapositiva2.mostrar()
    diapositiva3.mostrar()
    diapositiva4.mostrar()
```

■

Una de las características de Python que lo hacen tan poderoso es que podemos mezclar los estilos de programación orientado a objetos y funcional. Ahora las diapositivas son objetos

que pueden almacenarse en listas con facilidad, al igual que cualquier otro tipo de objeto de Python. Ahora veamos un ejemplo de la misma presentación en donde usamos `map` para mostrar las diapositivas.

**Programa 155: presentación, en objetos y funciones**

```
def mostrarDiapositiva(unaDiapositiva):
  unaDiapositiva.mostrar()

def reproducirPresentacion3():
  archImag = getMediaPath("barbara.jpg")
  archSon = getMediaPath("bassoon-c4.wav")
  diapositiva1 = Diapositiva(archImag,archSon)
  archImag = getMediaPath("beach.jpg")
  archSon = getMediaPath("bassoon-e4.wav")
  diapositiva2 = Diapositiva(archImag,archSon)
  archImag = getMediaPath("church.jpg")
  archSon = getMediaPath("bassoon-g4.wav")
  diapositiva3 = Diapositiva(archImag,archSon)
  archImag = getMediaPath("jungle2.jpg")
  archSon = getMediaPath("bassoon-c4.wav")
  diapositiva4 = Diapositiva(archImag,archSon)

  map(mostrarDiapositiva, [diapositiva1,diapositiva2,diapositiva3,diapositiva4])
```

■

¿Acaso es más fácil escribir la versión orientada a objetos de la presentación? Sin duda tiene menos código duplicado. Presenta la **encapsulación** en cuanto a que los datos y el comportamiento del objeto se definen en un solo lugar, de modo que cualquier modificación en uno sea fácil de realizar en el otro. A la capacidad de usar muchos objetos (como listas de ellos) se le conoce como **agregación**. Ésta es una idea poderosa. No siempre tenemos que definir nuevas clases: a menudo es posible usar las estructuras poderosas que conocemos, como las listas con objetos existentes, para generar un impacto considerable.

### 16.4.1 Cómo hacer la clase `Diapositiva` más orientada a objetos

¿Qué ocurre si necesitamos cambiar la imagen o el sonido de alguna clase? Podemos hacerlo. Sólo tenemos que cambiar las variables de instancia `imagen` o `sonido`. Pero si piensa en ello, se dará cuenta que no es algo muy seguro. ¿Qué tal si alguien más usa la presentación y decide guardar películas en la variable `imagen`? Podría ser fácil ponerla en funcionamiento, pero ahora tenemos dos usos distintos para la misma variable.

Lo que nos conviene en realidad es tener un método que maneje la acción de obtener o establecer una variable. Y si existe la posibilidad de que se guarden los datos incorrectos en la variable, podemos cambiar el método para establecer esa variable de modo que verifique el valor, para asegurarnos de que sea del tipo correcto y válido, antes de establecer esa variable. Para que esto funcione, *todos* los que usen la clase tienen que estar de acuerdo en usar los métodos para obtener y establecer las variables de instancia y *no* manipularlas de manera directa. En lenguajes como Java, podemos pedir al compilador que mantenga las variables de instancia privadas (mediante `private`) y no permita usos que traten de manipularlas en forma directa. En Python, lo mejor que podemos hacer es crear los métodos para establecer y obtener y tan sólo fomentar su uso.

A estos métodos los llamamos *setters* y *getters*. He aquí una versión de la clase en donde definimos setters y getters para las dos variables de instancia; como puede ver, son bastante simples. Observe cómo cambiamos los métodos `show` e incluso `__init__` de modo que podamos usar lo más que sea posible los métodos setter y getter en vez del acceso directo a las variables de instancia. Éste es el estilo de programación al que se refería Adele Goldberg cuando hablaba sobre "Preguntar sin tocar".

**Programa 156: clase `Diapositiva` con getters y setters**

```
class Diapositiva:
  def __init__(self, archivoImagen, archivoSonido):
      self.setImagen(makePicture(archivoImagen))
      self.setSonido(makeSound(archivoSonido))

  def getImagen(self):
    return self.imagen
  def getSonido(self):
    return self.sonido

  def setImagen(self,nuevaImagen):
    self.imagen = nuevaImagen
  def setSonido(self,nuevoSonido):
    self.sonido = nuevoSonido

  def mostrar(self):
      show(self.getImagen())
      blockingPlay(self.getSonido())
```

■

Una de las ventajas de esta clase modificada es que no tenemos que cambiar *nada* en nuestra función `reproducirPresentacion3`. Funciona así de simple, aun y cuando realizamos varias modificaciones en cuanto a la forma en que funciona la clase `Diapositiva`. Decimos que la función `reproducirDiapositiva3` y la clase `Diapositiva` están *débilmente acopladas*. Funcionan en conjunto y de maneras bien definidas, pero es posible cambiar el funcionamiento interno de una sin afectar a la otra.

## 16.5 MEDIOS ORIENTADOS A OBJETOS

Como dijimos antes, hemos estado usando objetos en todo el libro. Creamos objetos `Picture` con la función `makePicture`. También podemos crear una imagen mediante el constructor normal de Python.

```
>>> imag=Picture(getMediaPath("barbara.jpg"))
>>> imag.show()
```

He aquí cómo se define la *función* `show()`. Puede ignorar a `raise` y `__class__`. El punto clave es que lo único que hace la función es ejecutar el método `show` de la imagen existente.

```
def show(picture):
    if not picture.__class__ == Picture:
        print "show(picture): Input is not a picture"
        raise ValueError
    picture.show()
```

Podríamos tener otras clases que también sepan cómo usar la función `show`. Los objetos pueden tener sus propios métodos con nombres que también utilicen otros objetos. Es mucho más poderoso que cada uno de estos métodos con el mismo nombre puedan lograr el mismo *objetivo,* sólo que en distintas formas. Definimos una clase para las diapositivas y éstas sabían cómo usar la función `mostrar`. Tanto para las diapositivas como para las imágenes, el método `mostrar()` dice: "Muestra el objeto". Pero lo que ocurre en realidad es distinto en cada caso: las imágenes sólo se muestran a sí mismas, pero las diapositivas muestran sus imágenes y reproducen sus sonidos.

**Idea de ciencias computacionales: polimorfismo**

A la técnica de usar el mismo nombre para invocar distintos métodos que logran el mismo objetivo se le conoce como **polimorfismo.** Es algo muy poderoso para el programador. Sólo tenemos que decir a un objeto que use la función `mostrar()`: no tenemos que preocuparnos por saber cuál es el método específico que se va a ejecutar y ni siquiera tenemos que saber con exactitud qué objeto es al que le decimos que se muestre. Usted el programador tan sólo tiene que especificar su *objetivo,* que es mostrar el objeto. El programa orientado a objetos se encarga del resto.

■

Hay varios ejemplos de polimorfismo integrado en los métodos que usamos en JES.[1] Por ejemplo, tanto los píxeles como los colores saben acerca de los métodos `setRed`, `getRed`, `setBlue`, `getBlue`, `setGreen` y `getGreen`. Esto nos permite manipular los colores de los píxeles sin tener que extraer los objetos color por separado. Podríamos haber definido las funciones para recibir ambos tipos de entrada o proveer distintas funciones para cada tipo de entrada, pero ambas opciones son confusas. Esto es fácil de hacer con métodos.

```
>>> imag=Picture(getMediaPath("barbara.jpg"))
>>> imag.show()
>>> pixel = imag.getPixel(100,200)
>>> print pixel.getRed()
73
>>> color = pixel.getColor()
>>> print color.getRed()
73
```

El método `write()` es otro ejemplo. Se define el método `write(nombrearchivo)` para imágenes y para sonidos. ¿Alguna vez confundió `writePictureTo()` con `writeSoundTo()`? ¿No sería más fácil escribir siempre `write(nombrearchivo)`? Esto explica por qué el método se llama igual en ambas clases y por qué el polimorfismo es tan poderoso (tal vez se pregunte por qué no introdujimos esta técnica desde un principio. Pero ¿estaba usted listo en el capítulo 2 para hablar sobre la notación punto y los métodos polimórficos?).

En general, hay en realidad mucho más métodos definidos en JES que funciones. O dicho de manera más específica, hay varios métodos para dibujar sobre imágenes que no están disponibles como funciones.

[1] Recuerde que JES es un entorno para programar en Jython, el cual es un tipo específico de Python. Los soportes de medios forman parte de lo que ofrece JES: no pertenecen al núcleo de Python.

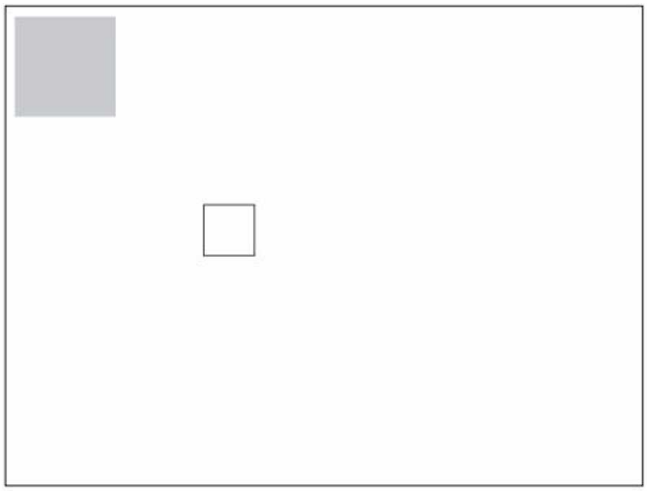


**FIGURA 16.12**
Ejemplos de métodos de rectángulos.

- Como es de esperar, las imágenes conocen `imag.addRect(color,x,y,anchura, altura)`, `imag.addRectFilled(color,x,y,anchura,altura)`, `imag.addOval (color,x,y,anchura,altura)` e `imag.addOvalFilled(color,x,y,anchura, altura)`.

  En la figura 16.12 podrá ver ejemplos de métodos de rectángulos dibujados a partir del siguiente ejemplo.

```
>>> imag=Picture(getMediaPath("640x480.jpg"))
>>> imag.addRectFilled(orange,10,10,100,100)
>>> imag.addRect(blue,200,200,50,50)
>>> imag.show()
>>> imag.write("nuevosrects.jpg")
```

  En la figura 16.13 podrá ver ejemplos de óvalos dibujados a partir del siguiente ejemplo.

```
>>> imag=Picture(getMediaPath("640x480.jpg"))
>>> imag.addOval(green,200,200,50,50)
>>> imag.addOvalFilled(magenta,10,10,100,100)
>>> imag.show()
>>> imag.write("ovalos.jpg")
```

- Las imágenes también conocen los *arcos*. Éstos son, en sentido literal, partes de un círculo. Los dos métodos son `imag.addArc(color,x,y,anchura,altura,anguloInic, anguloArco)` e `imag.addArcFilled(color,x,y,anchura,altura,anguloInic, anguloArco)`. Dibujan arcos con ángulos basados en el valor de `anguloArco`, en donde `anguloInic` es el punto de partida. Cero grados son las 3 en punto en la carátula del reloj. Un arco positivo se cuenta en sentido contrario a las manecillas del reloj y un arco negativo es en sentido a favor de las manecillas del reloj. El centro del círculo es la parte media del rectángulo definido por $(x, y)$ con la `anchura` y `altura` dadas.
- Ahora también podemos dibujar líneas coloreadas, mediante el uso de `imag.addLine-(color,x1,y1,x2,y2)`.

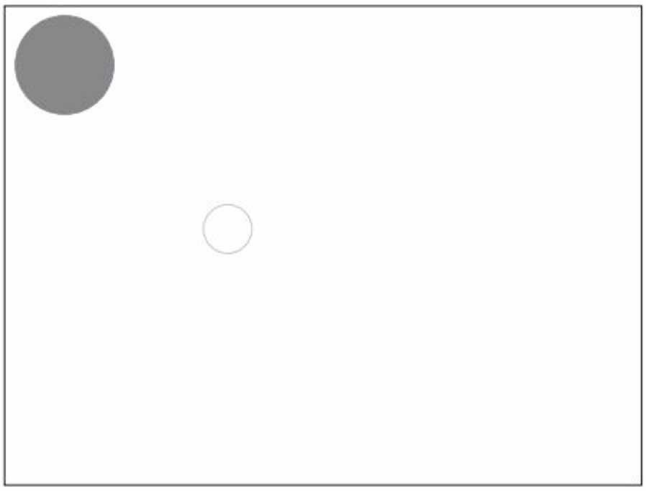


**FIGURA 16.13**
Ejemplos de métodos de óvalos.

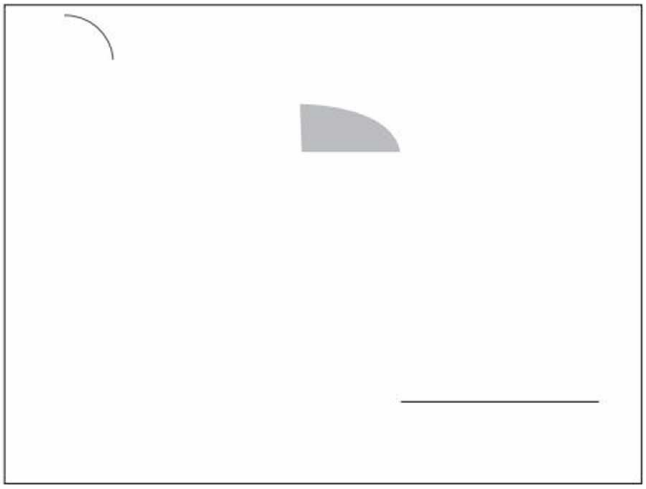


**FIGURA 16.14**
Ejemplos de métodos de arcos.

En la figura 16.14 podrá ver ejemplos de arcos y líneas dibujadas a partir del siguiente ejemplo.

```
>>> imag=Picture(getMediaPath("640X480.jpg"))
>>> imag.addArc(red,10,10,100,100,5,45)
>>> imag.show()
>>> imag.addArcFilled(green,200,100,200,100,1,90)
>>> imag.repaint()
>>> imag.addLine(blue,400,400,600,400)
>>> imag.repaint()
>>> imag.write("arcos-lineas.jpg")
```

- El texto en Java puede tener estilos, pero éstos se limitan a asegurar que todas las plataformas puedan duplicarlos. Uno de los que podríamos esperar ver es `imag.addText(color,x,y,cadena)`. También está `imag.addTextWithStyle(color,x,y,cadena,estilo)`, el cual obtiene un estilo creado de `makeStyle(fuente,énfasis,tamaño)`. El parámetro `fuente` es `sansSerif`, `serif` o `mono`. El parámetro énfasis es `italic`, `bold` o `plain`, o la suma de éstos para obtener combinaciones (por ejemplo, `italic+bold`). El parámetro `tamaño` es un tamaño de punto.

  En la figura 16.15 podrá ver ejemplos de texto dibujado a partir del siguiente ejemplo.

```
>>> imag=Picture(getMediaPath("640x480.jpg"))
>>> imag.addText(red,10,100,"Esta es una cadena roja")
>>> imag.addTextWithStyle(green,10,200,"Esta es una cadena en negrita,
    cursiva, verde y grande",makeStyle(sansSerif, bold+italic,18))
>>> imag.addTextWithStyle(blue,10,300,"Esta es una cadena azul, más
    grande, sólo cursiva, serif",makeStyle(serif,italic,24))
>>> imag.write("texto.jpg")
```

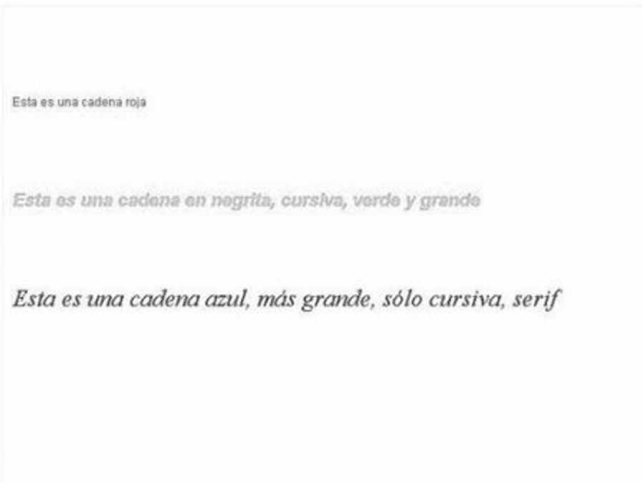



**FIGURA 16.15**
Ejemplos de métodos de texto.

Las funciones de medios que escribimos en capítulos anteriores pueden volver a escribirse en forma de métodos. Tendremos que crear una subclase de la clase `Picture` y agregar el método a esa clase.

**Programa 157: creación de un atardecer usando un método**

```
class MiImagen(Picture):
  def crearAtardecer(self):
    for p in getPixels(self):
      p.setBlue(int(p.getBlue()*0.7))
      p.setGreen(int (p.getGreen()*0.7))
```

■

Este programa puede usarse así:

```
>>> imag = MiImagen(getMediaPath("beach.jpg"))
>>> imag.explore()
>>> imag.crearAtardecer()
>>> imag.explore()
```

También podemos crear nuevas subclases de la clase Sound y nuevos métodos para trabajar con objetos sonido. Los métodos para acceder a los valores de una muestra de sonido son `getSampleValue()` y `getSampleValueAt(índice)`.

**Programa 158: invertir un sonido con un método**

```
class MiSonido(Sound):
  def invertir(self):
    destino = Sound(self.getLength())
    indiceOrigen = self.getLength() - 1
    for indiceDestino in range(0,destino.getLength()):
      valorOrigen = self.getSampleValueAt(indiceOrigen)
      destino.setSampleValueAt(indiceDestino,valorOrigen)
      indiceOrigen = indiceOrigen - 1
    return destino
```

■

Podemos usar el programa anterior de la siguiente forma:

```
>>> sonido = MiSonido(getMediaPath("always.wav"))
>>> sonido.explore()
>>> destino = sonido.invertir()
>>> destino.explore()
```

## 16.6 LA CAJA JOE

Adele Goldberg y Alan Kay desarrollaron el ejemplo más antiguo utilizado para enseñar programación orientada a objetos. A este método se le conoce como *La caja Joe*. No hay nada nuevo en este ejemplo, aunque ofrece un ejemplo *diferente* desde otra perspectiva, por lo que vale la pena revisarlo.

Imagine que tiene una clase llamada `Caja`, como la que se muestra a continuación:

```
class Caja:
  def __init__(self):
    self.establecerColorPredeterminado()
    self.tam=10
    self.posic=(10,10)
  def establecerColorPredeterminado(self):
    self.color = red
  def dibujar(self,lienzo):
    addRectFilled(lienzo, self.posic[0],self.
    posic[1],self.tam, self. tam,self.color)
```

¿Qué verá si ejecuta el siguiente código?

```
>>> lienzo = makeEmptyPicture(400,200)
>>> joe = Caja()
>>> joe.dibujar(lienzo)
>>> show(lienzo)
```

Vamos a rastrearlo.

- Es obvio que la primera línea sólo crea un `lienzo` blanco con 400 píxeles de ancho y 200 píxeles de alto.
- Al crear a `joe`, se invoca el método `__init__`. Luego se invoca el método `establecerColorPredeterminado` sobre `joe`, por lo que recibe el color predeterminado rojo. Al ejecutarse `self.color=red`, se crea la *variable de instancia* `color` para `joe` y obtiene un valor de rojo. Regresamos a `__init__`, en donde `joe` recibe un tamaño de 10 y una posición de (10,10) (`tam` y `posic` se convierten en nuevas variables de instancia).
- Cuando se le pide a `joe` que se dibuje a sí mismo en el `lienzo`, se dibuja como un rectángulo rojo relleno (`addRectFilled`) en la posición $x$ 10 y la posición $y$ 10, con un tamaño de 10 píxeles de cada lado.

Podríamos agregar un método a `Caja` que nos permita hacer que `joe` cambie su tamaño.

```
class Caja:
  def __init__(self):
    self.establecerColorPredeterminado()
    self.tam=10
    self.posic=(10,10)
  def establecerColorPredeterminado(self):
    self.color = red
  def dibujar(self,lienzo):
    addRectFilled(lienzo, self.posic[0],self.
    posic[1],self.tam, self.tam,self.color)
  def crecer(self,tam):
    self.tam=self.tam+tam
```

Ahora podemos pedir a Joe que crezca mediante la función `crecer`. Un número negativo como –2 provocará que `joe` se encoja. Un número positivo hará que `joe` crezca; aunque tendríamos que agregar un método `mover` si queremos que crezca mucho y pueda caber todavía en el lienzo.

Considere ahora el siguiente código que se agrega en la misma área del programa.

```
class CajaTriste(Caja):
  def establecerColorPredeterminado(self):
    self.color=blue
```

Observe que `CajaTriste` lista a `Caja` como una superclase (clase padre). Esto significa que `CajaTriste` *hereda* todos los métodos de `Caja`. ¿Qué veremos al ejecutar el siguiente código?

```
>>> jane = CajaTriste()
>>> jane.dibujar(lienzo)
>>> repaint(lienzo)
```

Vamos a rastrearlo:

- Al crear a `jane` como una `CajaTriste`, se ejecuta el método `__init__` en la clase `Caja`.
- Lo primero que ocurre en `__init__` es que llamamos a `establecerColorPredeterminado` *en el objeto de entrada* `self`. Ahora ese objeto es `jane`. Por ende, invocamos a `establecerColorPredeterminado` de `jane`. Decimos que `establecerColorPredeterminado` de `CajaTriste` *redefine* al método de `Caja` con el mismo nombre.
- El método `establecerColorPredeterminado` de `jane` establece el color en azul.
- Después regresamos a ejecutar el resto del método `__init__` de `Caja`. Establecemos el tamaño de `jane` en 10 y su posición en (10,10).
- Cuando pedimos a `jane` que se dibuje, aparece como un cuadrado azul de 10 × 10 en la posición (10,10). Si no movemos o aumentamos el tamaño de `joe`, éste desaparecerá al momento que `jane` se dibuje encima de él.

Cabe mencionar que `joe` y `jane` son distintos *tipos* de `Caja`. Tienen las mismas variables de instancia (pero distintos *valores* para las mismas variables) y en su mayoría conocen las mismas cosas. Por ejemplo, como ambas conocen el método `dibujar`, decimos que `dibujar` es *polimórfico*. La palabra *polimórfico* significa muchas formas.

Una `CajaTriste (jane)` es un poco distinta en cuanto a la forma en que se comporta al momento de su creación, de modo que conoce algunas cosas de manera distinta. Joe y Jane resaltan algunas de las ideas básicas de la programación orientada a objetos: herencia, especialización en subclases y variables de instancia compartidas, al tiempo que tienen distintos valores en sus variables de instancia.

## 16.7 ¿POR QUÉ OBJETOS?

Un rol para los objetos es reducir el número de nombres que tenemos que recordar. Por medio del polimorfismo, sólo tenemos que recordar el nombre y el objetivo, no todas las diversas funciones globales.

Pero lo más importante es que los objetos encapsulan los datos y el comportamiento. Imagine que desea cambiar el nombre de una variable de instancia y después todos los métodos que usan esa variable. Es mucho por cambiar. ¿Qué tal si olvida uno? Sería conveniente cambiarlos todos en un solo lugar.

Los objetos reducen el *acoplamiento* entre los componentes del programa; esto es, qué tanto dependen unos de otros. Imagine que tiene varias funciones y todas usan la misma variable global. Si cambia una función de modo que almacene algo ligeramente diferente en esa variable, todas las demás funciones también deberán actualizarse o de lo contrario no funcionarán. A esto se le conoce como acoplamiento *fuerte*. Los objetos que sólo usan métodos sobre otros objetos (sin acceso directo a las variables de instancia) tienen un acoplamiento más *débil*. El acceso está bien definido y se puede modificar con facilidad en un solo lugar. Los cambios en un objeto no exigen cambios en otros objetos.

Una ventaja del acoplamiento débil es la facilidad para desarrollar en contextos de equipos. Es posible tener a distintas personas trabajando en diferentes clases. Mientras que todos estén de acuerdo en la forma en que funcionará el acceso a través de los métodos, nadie tendrá qué saber cómo funcionan los métodos de los demás. La programación orientada a objetos puede ser especialmente útil al trabajar en equipos.

La agregación es también un beneficio considerable de los sistemas de objetos. Es posible tener muchos objetos que realicen cosas útiles. ¿Desea más? ¡Sólo hay que crearlos!

Los objetos de Python son similares a los objetos de muchos lenguajes. No obstante, una de las diferencias considerables está en el acceso a las variables de instancia. En Python, cualquier objeto puede acceder a las variables de instancia de cualquier otro objeto y manipularlas. Esto no es cierto en lenguajes como Java, C++ o Smalltalk. En estos otros lenguajes está limitado el acceso a las variables de instancia desde otros objetos, e incluso puede eliminarse por completo: así sólo es posible acceder a las variables de instancia de los objetos por medio de métodos getter y setter.

La **herencia** es otra gran parte de los sistemas de objetos. Como vimos con los ejemplos de la tortuga y la caja, podemos declarar que una clase (*clase padre*) será *heredada* por otra clase (*clase hija*) (lo que también se conoce como superclase y subclase). La herencia ofrece un polimorfismo al instante: las instancias de la hija obtienen de manera automática todos los datos y el comportamiento de la clase padre. Así, la hija puede agregar más comportamiento y datos a lo que la clase padre tenía. A esto se le conoce como hacer que la hija sea una *especialización* de la clase padre. Por ejemplo, la instancia de un rectángulo en tercera dimensión podría conocer y hacer todo lo que hace una instancia de rectángulo mediante la instrucción `class Rectangulo3D(Rectangulo)`.

La herencia es muy popular en el mundo orientado a objetos, aunque tiene sus ventajas y desventajas. Reduce aún más la duplicación de código, lo cual es algo bueno. En la práctica real, la herencia no se utiliza tanto como muchas otras de las ventajas de la programación orientada a objetos (como la agregación y la encapsulación), además de que puede ser confusa. ¿Cuál método se está ejecutando cuando escribimos el código siguiente? Es invisible desde aquí, y si está *equivocado*, puede ser difícil averiguar en dónde está equivocado.

```
miCaja = Rectangulo3D()
miCaja.dibujar()
```

Entonces, ¿cuándo hay que usar objetos? Debemos definir nuestras propias clases cuando tenemos datos y comportamiento que deseamos definir para todas las instancias del grupo (como imágenes y sonidos). Debemos usar objetos existentes *todo el tiempo*. Son muy poderosos. Si usted no se siente cómodo con la notación punto y las ideas de los objetos, puede seguir usando funciones: trabajan muy bien. Los objetos son más convenientes en sistemas más complejos.

## RESUMEN DE PROGRAMACIÓN

Algunas de las piezas de programación que vimos en este capítulo.

## PROGRAMACIÓN ORIENTADA A OBJETOS

| | |
|---|---|
| `class` | Nos permite definir una clase. La palabra clave `class` recibe el nombre de una clase y una superclase opcional entre paréntesis, y termina con un signo de dos puntos. Después le siguen los métodos de la clase, con sangría dentro del bloque de la clase. |
| `__init__` | El nombre del método invocado sobre un objeto al momento de su creación. No se requiere tener uno. |

## MÉTODOS DE GRÁFICOS

| | |
|---|---|
| `addRect`, `addRectFilled` | Los métodos en la clase `Picture` para dibujar rectángulos y rectángulos rellenos. |
| `addOval`, `addOvalFilled` | Los métodos en la clase `Picture` para dibujar óvalos y óvalos rellenos. |
| `addArc, addArcFilled` | Los métodos en la clase `Picture` para dibujar arcos y arcos rellenos. |
| `addText`, `addTextWithStyle` | Los métodos en la clase `Picture` para dibujar texto y texto con elementos de estilo (como negrita o sans serif). |

| | |
|---|---|
| `addLine` | El método en la clase `Picture` para dibujar una línea. |
| `getRed, getGreen, getBlue` | Los métodos de los objetos `Pixel` y `Color` para obtener los componentes de color rojo, verde y azul. |
| `setRed, setGreen, setBlue` | Los métodos de los objetos `Pixel` y `Color` para establecer los componentes de color rojo, verde y azul. |

## PROBLEMAS

16.1 Responda a las siguientes preguntas.

- ¿Cuál es la diferencia entre una instancia y una clase?
- ¿En qué difieren las funciones y los métodos?
- ¿En qué difiere la programación orientada a objetos de la programación por procedimientos?
- ¿Qué es el polimorfismo?
- ¿Qué es la encapsulación?
- ¿Qué es la agregación?
- ¿Qué es un constructor?
- ¿Cómo influyen las células biológicas en el desarrollo de la idea de los objetos?

16.2 Responda a las siguientes preguntas.

- ¿Qué es la herencia?
- ¿Qué es una superclase?
- ¿Qué es una subclase?
- ¿Qué métodos hereda una clase hija?
- ¿Qué variables de instancia (campos) hereda una clase hija?

16.3 Agregue un método a la clase `Tortuga` para dibujar un triangulo equilátero.

16.4 Agregue un método a la clase `Tortuga` para dibujar un rectángulo con base en una anchura y altura específicas.

16.5 Agregue un método a la clase `Tortuga` para dibujar una casa simple. Puede tener un rectángulo para la casa y un triángulo equilátero para el techo.

16.6 Agregue un método a la clase `Tortuga` para dibujar una calle de casas.

16.7 Agregue un método a la clase `Tortuga` para dibujar una letra.

16.8 Agregue un método a la clase `Tortuga` para dibujar sus iniciales.

16.9 Cree una película con varias tortugas moviéndose en cada cuadro.

16.10 Agregue otro constructor a la clase `Diapositiva` que reciba sólo el nombre de archivo de una imagen.

16.11 Cree una clase llamada `Presentacion` que contenga una lista de diapositivas y muestre una diapositiva a la vez.

16.12 Cree una clase llamada `PanelCaricaturas` que reciba un arreglo de objetos `Picture` y muestre las imágenes de izquierda a derecha. También debe tener un título y un autor, además de mostrar el título en el borde superior izquierdo y el autor en el borde superior derecho.

16.13 Cree una clase llamada `Estudiante`. Cada estudiante debe tener un nombre y una imagen. Agregue un método llamado `mostrar` que muestre la imagen del estudiante.

16.14 Agregue un campo a la clase `Presentacion` que contenga el título y modifique el método `mostrar`, para que primero muestre una imagen en blanco con el título en ella.

16.15 Cree una clase llamada `ListaReproduccion` que reciba una lista de sonidos y los reproduzca uno a la vez.

16.16 Use los métodos en la clase `Picture` para dibujar una cara sonriente.

16.17 Use los métodos en la clase `Picture` para dibujar un arcoíris.

16.18 Vuelva a escribir las funciones espejo como métodos en la clase `MiImagen`.

16.19 Realice algunas modificaciones a la caja Joe.

- Agregue un método a `Caja` llamado `setColor` que reciba un color como entrada y luego haga que el color de entrada sea el nuevo color de la caja (¿tal vez `establecerColorPredeterminado` debería llamar a `setColor`?)
- Agregue un método a `Caja` llamado `setTamanio` que reciba un número como entrada y luego lo asigne como el nuevo tamaño para la caja.
- Agregue un método a `Caja` llamado `setPosicion` que reciba una lista o tupla como parámetro y después la asigne como la nueva posición para la caja.
- Modifique `__init__` de modo que use `setTamanio` y `setPosicion` en vez de establecer de manera directa las variables de instancia.

*16.20 Termine el ejemplo de la caja Joe.

*a*) Implemente `crecer` y `mover`. El método `mover` debe recibir como entrada una distancia relativa como (–10,15) para moverse 10 píxeles a la izquierda (posición *x*) y 15 píxeles hacia abajo (posición *y*).

*b*) Dibuje patrones al crear a `joe` y `jane`; después mueva un poco y dibuje, aumente un poco el tamaño y dibuje, luego vuelva a pintar el nuevo lienzo.

16.21 Cree una película con cajas que aumenten y reduzcan su tamaño.

*Problema más desafiante.

## PARA PROFUNDIZAR

Queda mucho más qué hacer con Python en cuanto a explorar los estilos de programación por procedimientos, funcional y orientada a objetos. Mark recomienda los libros de Mark Lutz (en especial [30]) y Richard Hightower [24] como excelentes introducciones a los ámbitos más extensos de Python. También podría explorar algunos de los tutoriales en el sitio Web de Python (`http://www.python.org`).

APÉNDICE

# A Referencia rápida de Python


A.1 VARIABLES
A.2 CREACIÓN DE FUNCIONES
A.3 CICLOS Y CONDICIONALES
A.4 OPERADORES Y FUNCIONES DE REPRESENTACIÓN
A.5 FUNCIONES NUMÉRICAS
A.6 OPERACIONES DE SECUENCIA
A.7 CADENAS DE ESCAPE
A.8 MÉTODOS DE CADENA ÚTILES
A.9 ARCHIVOS
A.10 LISTAS
A.11 DICCIONARIOS, TABLAS HASH O ARREGLOS ASOCIATIVOS
A.12 MÓDULOS EXTERNOS
A.13 CLASES
A.14 MÉTODOS FUNCIONALES


## A.1 VARIABLES

Las variables empiezan con una letra y pueden ser cualquier palabra, *excepto* una de las *palabras reservadas*, las cuales son: `and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `yield`.

Podemos usar `print` para mostrar el valor de una expresión (por ejemplo, una variable). Si sólo escribimos la variable sin `print`, obtendremos la representación interna: las funciones y objetos nos dicen en qué parte de la memoria se encuentran y las cadenas aparecen con sus comillas.

```
>>> x = 10
>>> print x
10
```

```
>>> x
10
>>> y='cadena'
>>> print y
cadena
>>> y
'cadena'
>>> p=makePicture(pickAFile())
>>> print p
Picture, filename C:\ip-book\mediasources\
   7inx95in.jpg height 684 width 504
>>> p
<media, Picture instance at 6436242>
>>> print sin(12)
-0.5365729180004349
>>> sin
<java function sin at 26510058>
```

## A.2 CREACIÓN DE FUNCIONES

Para definir funciones usamos `def`. Por ejemplo `def x(a,b):` define una función llamada "x" que recibe dos valores de entrada, los cuales están enlazados a las variables "a" y "b". El cuerpo de la función sigue después de la instrucción `def` y lleva sangría.

La función puede devolver valores mediante el uso de la instrucción `return`.

## A.3 CICLOS Y CONDICIONALES

Creamos la mayoría de los ciclos mediante el uso de `for`, que recibe una variable índice y una lista. El cuerpo del ciclo se ejecuta una vez por cada elemento de la lista.

```
>>> for p in [1,2,3]:
...     print p
...
1
2
3
```

Con frecuencia, la lista en un ciclo `for` se genera mediante el uso de una función `range`. Esta función puede recibir una, dos o tres entradas. Con una entrada, el rango es desde cero hasta la entrada menos uno. Con dos, el rango empieza en la primera entrada y se detiene *antes* de la segunda. Con tres, el rango empieza en la primera, avanza a intervalos según la tercera y termina *antes* de la segunda.

```
>>> range(4)
[0, 1, 2, 3]
>>> range(1,4)
[1, 2, 3]
```

```
>>> range(1,4,2)
[1, 3]
```

El ciclo `while` recibe una expresión lógica y ejecuta su bloque siempre y cuando la expresión lógica sea verdadera.

```
>>> x = 1
>>> while x < 5:
...         print x
...         x = x + 1
...
1
2
3
4
```

Una instrucción `break` termina de inmediato el ciclo actual.

Una instrucción `if` recibe una expresión lógica y la evalúa. Si es verdadera, se ejecuta el bloque de la instrucción `if`. Si es falsa, se ejecuta la cláusula `else`: , en caso de que exista una.

```
>>> if a < b:
...         print "a es menor"
... else:
...         print "b es menor"
```

## A.4 OPERADORES Y FUNCIONES DE REPRESENTACIÓN

| | |
|---|---|
| +, -, *, /, ** | Suma, resta, multiplicación, división y exponenciación. El orden de precedencia es algebraico. |
| <, >, ==, <=, >= | Los operadores lógicos menor que, mayor que, igual a, menor o igual a, mayor o igual a. |
| <>, != | Los operadores lógicos de desigualdad (ambos son equivalentes) |
| and, or, not | Los conjuntivos lógicos `and`, `or` y `not`. |
| int() | Devuelve la parte entera de la entrada (número de punto flotante o cadena). |
| float() | Devuelve una versión de punto flotante de la entrada. |
| str() | Devuelve una representación de cadena de la entrada. |
| ord() | Dado un carácter de entrada, devuelve la representación numérica en ASCII. |

## A.5 FUNCIONES NUMÉRICAS

| | |
|---|---|
| `abs()` | Valor absoluto. |
| `sin()` | Seno. |
| `cos()` | Coseno. |
| `max()` | Valor máximo de las entradas (incluyendo una lista). |
| `min()` | Valor mínimo de las entradas (incluyendo una lista). |
| `len()` | Devuelve la longitud de la secuencia de entrada. |

## A.6 OPERACIONES DE SECUENCIA

Las secuencias (cadenas, listas, tuplas) pueden sumarse o concatenarse (por ejemplo, `s1 + s2`).

Es posible acceder a los elementos de una secuencia mediante el uso de porciones:
`seq[n]` accede al *n*-ésimo elemento de la lista (el primer elemento es cero).
`seq[n:m]` accede a los elementos desde el *n*-ésimo hasta, *pero sin incluir*, el *m*-ésimo.
`seq[:m]` accede a los elementos desde el inicio hasta, pero sin incluir, el *m*-ésimo.
`seq[n:]` accede a los elementos desde el *n*-ésimo hasta el final de la secuencia.

## A.7 CADENAS DE ESCAPE

| | |
|---|---|
| `\t` | Carácter de tabulación |
| `\b` | Retroceso |
| `\n` | Nueva línea |
| `\r` | Retorno |
| `\uXXXX` | Carácter Unicode, XXXX hexadecimal |

Coloque una "r" antes de una cadena, como en `r"C:\mediasources"`, para manipular una cadena en modo puro, ignorando los escapes.

## A.8 MÉTODOS DE CADENA ÚTILES

- `count(sub)`: devuelve el número de veces que aparece `sub` en la cadena.
- `find(sub)`: devuelve el índice en donde aparece `sub` en la cadena, o devuelve –1 si no se encuentra. `find` puede recibir un punto inicial opcional y un punto final también opcional. `rfind` recibe las mismas entradas pero trabaja de derecha a izquierda.
- `upper()`, `lower()`: convierte la cadena a sólo mayúsculas o minúsculas.

- `isalpha()`, `isdigit()`: devuelven verdadero si todos los caracteres en la cadena son alfabéticos o todos son numéricos, respectivamente.
- `replace(s,r)`: sustituye todas las instancias de "s" con "r" en la cadena.
- `split(d)`: devuelve una lista de cadenas en donde el carácter `d` es el punto de división.

## A.9 ARCHIVOS

Los archivos se abren mediante `open` con dos entradas: el nombre de archivo y un modo de archivo. El modo de archivo es "r" para lectura, "w" para escritura y "a" para añadir, concatenado con una "t" para texto o una "b" para binario. Los métodos de archivos son:

- `read()`: devuelve todo el archivo como una cadena.
- `readlines()`: devuelve todo el archivo como una lista de cadenas delimitadas por salto de línea.
- `write(s)`: escribe la cadena `s` en el archivo.

## A.10 LISTAS

Las listas se indexan como las secuencias mediante el uso de "`[]`". Se concatenan mediante +. Los métodos de listas son:

- `append(b)`: añade el elemento `b` a una lista.
- `remove(b)`: retira el elemento `b` de la lista.
- `sort()`: ordena la lista.
- `reverse()`: invierte la lista.
- `count(s)`: devuelve el número de veces que aparece el elemento `s` en la lista.

## A.11 DICCIONARIOS, TABLAS HASH O ARREGLOS ASOCIATIVOS

Los diccionarios se crean con `{}`. Es posible acceder a ellos mediante una clave.

```
>>> d = {'gato':'Diana', 'perro':'Fido'}
>>> print d
['gato': 'Diana', 'perro': 'Fido'}
>>> print d.keys()
['gato', 'perro']
>>> print d['gato']
Diana
```

## A.12 MÓDULOS EXTERNOS

Para acceder a los módulos se utiliza `import`. También se pueden introducir como un alias; por ejemplo, `import javax.swing como swing`. Es posible importar piezas específicas sin necesidad de la notación punto para acceder a ellas, si usamos `from módulo import n1, n2`. Podemos importar y acceder a todas las piezas de un módulo sin la notación punto, si usamos `from módulo import *`.

## A.13 CLASES

Las clases se crean mediante la palabra clave `class` seguida del nombre de la clase y una superclase opcional entre paréntesis (una o más). Después se escriben los métodos con sangría. Los *constructores* (que se invocan al momento de crear una nueva instancia de la clase) deben llamarse `__init__`. Puede tener más de un constructor en una clase de Python siempre y cuando reciban distintos parámetros.

## A.14 MÉTODOS FUNCIONALES

| | |
|---|---|
| `apply` | Recibe una función y una lista como entrada para esa función, en donde la lista tiene tantos elementos como los que reciba la función de entrada. Invoca a la función con la entrada. |
| `map` | Recibe una función y una lista de varias entradas para esa función. Invoca a la función con cada una de las entradas y devuelve una lista de las salidas (valores return). |
| `filter` | Recibe una función y una lista de varias entradas para esa función. Invoca a la función con cada uno de los elementos de la lista y devuelve *el elemento de entrada* si la función devuelve verdadero (un valor distinto de cero) para ese elemento. |
| `reduce` | Recibe una función que tiene dos entradas y una lista de varias entradas para esa función. Ésta se aplica a los primeros dos elementos de la lista y el resultado se usa como entrada con el siguiente elemento de la lista; después el resultado de esto se utiliza como entrada con el *siguiente* elemento y así, en lo sucesivo. El resultado total se devuelve al final. |

# Bibliografía


1. AAUW, *Tech-Savvy: Educating Girls in the New Computer Age*, Asociación estadounidense de la Fundación de educación para mujeres universitarias, Nueva York, 2000.
2. HAROLD ABELSON, GERALD JAY SUSSMAN Y JULIE SUSSMAN, *Structure and Interpretation of Computer Programs*, 2ª edición, MIT Press, Cambridge, MA, 1996.
3. KEN ABERNETHY Y TOM ALLEN, *Exploring the Digital Domain: An Introduction to Computing with Multimedia and Networking*, PWS Publishing, Boston, 1998.
4. ACM/IEEE, *Computing Curriculum 2001*, `http://www.acm.org/sigcse/cc2001` (2001).
5. BETH ADELSON Y ELLIOT SOLOWAY, "The Role of Domain Experience in Software Design", *IEEE Transactions on Software Engineering* **SE-11** (1985), no. 11, 1351-1360.
6. JENS BENNEDSEN Y MICHAEL E. CASPERSEN, "Failure Rates in Introductory Programming", *SIGCSE Bulletin* **39** (2007), no. 2, 32-36.
7. RICHARD BOULANGER (ed.), *The Csound Book: Perspectives in Synthesis, Sound Design, Signal Processing, and Programming*, MIT Press, Cambridge, MA, 2000.
8. AMY BRUCKMAN, "Situated Support for Learning: Storm's Weekend with Rachael", *Journal of the Learning Sciences* **9** (2000), no. 3, 329-372.
9. JOHN T. BRUER, *Schools for Thought: A Science of Learning in the Classroom*, MIT Press, Cambridge, MA, 1993.
10. ALAN J. DIX, JANET E. FINLAY, GREGORY D. ABOWD Y RUSSELL BEALE, *Human-Computer Interaction*, 2a. edición, Prentice Hall, Upper Saddle River, NJ, 1998.
11. CHARLES DODGE Y THOMAS A. JERSE, *Computer Music: Synthesis, Composition, and Performance,* Schirmer-Thomson Learning, Nueva York, 1997.
12. MATTHIAS FELLEISEN, ROBERT BRUCE FINDLER, MATTHEW FLATT Y SHRIRAM KRISHNAMURTHI, *How to Design Programs: An Introduction to Programming and Computing*, MIT Press, Cambridge, MA, 2001.
13. ANN E. FLEURY, "Encapsulation and Reuse as Viewed by Java Students", *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education* (2001), pp. 189-193.
14. JAMES D. FOLEY, ANDRIES VAN DAM Y STEVEN K. FEINER, *Introduction to Computer Graphics*, Addison Wesley, Reading, MA, 1993.
15. ANDREA FORTE Y MARK GUZDIAL, *Computers for Communication, Not Calculation: Media as a Motivation and Context for Learning*, HICSS 2004, Big Island, HI, IEEE Computer Society 2004.
16. DANNY GOODMAN, *JavaScript & DHTML Cookbook*, O'Reilly & Associates, Sebastapol, CA, 2003.

17. Martin Greenberger, "Computers and the World of the Future", grabaciones transcritas de conferencias en la Sloan School of Business Administration, abril de 1961, MIT Press, Cambridge MA, 1962.
18. Rashi Gupta, *Making Use of Python*, Wiley, Nueva York, 2002.
19. Mark Guzdial, *Squeak: Object-Oriented Design with Multimedia Applications*, Prentice Hall, Englewood, NJ, 2001.
20. Mark Guzdial y Kim Rose (eds.), *Squeak, Open Personal Computing for Multimedia*, Prentice Hall, Englewood, NJ, 2001.
21. Mark Guzdial y Allison Elliot Tew, "Imagineering Inauthentic Legitimate Peripheral Participation: An Instructional Design Approach for Motivating Computing Education", artículo presentado en el Seminario internacional de investigación sobre educación computacional, Canterbury, Reino Unido, ACM, Nueva York, 2006.
22. Idit Harel y Seymour Papert, "Software Design as a Learning Environment", *Interactive Learning Environments* **1** (1990), no. 1, 1-32.
23. Brian Harvey, *Computer Science Logo Style*, 2a. edición, vol. 1: *Symbolic Computing*, MIT Press, Cambridge, MA, 1997.
24. Richard Hightower, *Python Programming with the Java Class Libraries*, Addison-Wesley, Reading, MA, 2003.
25. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace y Alan Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself", *OOPSLA'97 Conference Proceedings,* ACM, Atlanta, GA, 1997, pp. 318-326.
26. Janet Kolodner, *Case-Based Reasoning*, Morgan Kaufmann, San Mateo, CA, 1993.
27. Heather Perry, Lauren Rich y Mark Guzdial, "A CS1 Course Designed to Address Interests of Women", *ACM SIGCSE Conference 2004*, Norfolk, VA, ACM, Nueva York, 2004, pp. 190-194.
28. Margaret Livingstone, *Vision and Art: The Biology of Seeing*, Harry N. Abrams, Nueva York, 2002.
29. Frederik Lundh, *Python Standard Library*, O'Reilly and Associates, Sebastapol, CA, 2001.
30. Mark Lutz y David Ascher, *Learning Python*, O'Reilly & Associates, Sebastopol, CA, 2003.
31. Jane Margolis y Allan Fisher, *Unlocking the Clubhouse: Women in Computing,* MIT Press, Cambridge, MA, 2002.
32. Dan Olsen, *Developing User Interfaces*, Morgan Kaufmann Publishers, San Mateo, CA, 1998.
33. Mitchel Resnick, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press, Cambridge, MA, 1997.
34. Jeannette Wing, "Computational Thinking", *Communications of the ACM* **49** (2006), no. 3, 33-35.
35. Curtis Roads, *The Computer Music Tutorial*, MIT Press, Cambridge, MA, 1996.
36. Robert Sloan y Patrick Troy, "CS 0.5: A Better Approach to Introductory Computer Science for Majors", *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, Nueva York, 2008, pp. 271-275.

37. Allison Elliot Tew, Charles Fowler y Mark Guzdial, "Tracking an Innovation in Introductory CS Education from a Research University to a Two-Year College", *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, Nueva York, 2005, pp. 416-420.

# Índice


ÍNDICE
SÍMBOLOS
#, 87, 181
*, 53
\n, 30, 247
\t, 247
\u, 247
__class__, 394
__init__, 392, 402
=, 28
O(), 342

## A

abajoArriba, 365, 367
abs, 23, 37, 121
abstracción, 9, 34, 185
  procedural, 218, 295, 319, 358, 364, 376
  reutilización, 218
  subfunciones, 218
abstracto, 235
acoplada, 394
acoplamiento, 401
  débil, 401
  fuerte, 401
acorde musical, 198
acordes, 198
actual, 230
addArc, 403
addArcFilled, 403
addLine(imagen,x1,y1,x2,y2), 132, 139
addLine, 403
addOval, 403
addOvalFilled, 403
addRect(), 316
addRect(imagen,x1,y1,anchura, altura), 132, 139
addRect, 403
addRectFilled(imagen,x1,y1,anchura, altura,color), 132, 139
addText(imagen,x,y,cadena), 132, 139
addText, 316, 403
addTextWithStyle, 403
Adobe Photoshop, 12
agregación, 377, 393, 401
agudeza, 42
AIFF, 25, 212
alcance, 58, 162, 178, 188
  global, 58, 190, 192
  local, 58
alfabeto en Latín, 247
álgebra, 19
algoritmo, xix, 4, 8, 168, 187, 202, 235, 341
  de optimización, 348
  diseño, 235
  espacio, 342
  mezcla, 235
  muestreo, 235
  reflejo, 187
  tiempo, 342
algoritmos, 3
algoritmos de ordenamiento, 343
alias, 162
alt, 292
amplitud, 146, 161
análisis orientado a objetos, 377
ancla, 292
  etiqueta, 292, 293
anidar, 76, 245
animaciones, 313, 314
anydbm, 300
append, 266
append(algo), 251
Apple QuickTime Player Pro, 161
apply, 373
apply(), 359
aprendizaje de máquina, 349
árbol, 358
  en listas, 254
  hijos, 254
  hoja, 254
  padre, 254
  rama, 254
archivo, 18, 253, 370
  apertura, 255
  lectura, 255
  métodos, 255, 410
  ruta, 18
archivo.close(), 256
archivo.read(), 255
archivo.readlines(), 255
archivo.write(unacadena), 256
archivos binarios, 255
arcos, 396
argumentos
  de palabras clave, 328
  opcionales, 328
armónicos, 148
arreglo, 17, 41, 54, 154, 180
  asociativos, 302
  bidimensional, 41
  copia, 180
  de caracteres, 245
  elemento, 154, 155
  índice, 155
  notación, 76, 175
  unidimensional, 41
arribaAbajo(), 375
asume, 35
ataque, caída y sostenido (ASD) envoltura, 215
Audacity, 12
aumentarVolumen, 161
AutoCAD, 133
avance paso a paso, 57
avanzar, 259
AVI, 314, 315
azul, 11

## B

bajo consumo de energía, 4
barras diagonales inversas, 30
bases de datos, 4, 299
  conexión, 305
  para crear páginas Web, 307
  tablas, 301
Berners-Lee, Tim, 272
Biblioteca de imágenes de Python (PIL), 236
bibliotecas
  establecer rutas, 184
Big-O, notación, 342
binario, 9
bit, 8, 170

como información, 170
de signo, 153
blockingPlay, 159, 171
blockingPlay(), 390
bloque, 31, 54
anidado, 55
BMP, 134
botón load (cargar), 20
Brando, Marlon, 161
break, 280, 286, 408
brilloCombinar, 330
brilloPixel, 328
Brooks, Frederick P., 241
bus del sistema, 352
buscarSecuencia, 260
Bush, Vannevar, 273
búsqueda
binaria, 346
lineal, 345
byte, 8, 9, 17

## C

C, 7
C++, 6, 18, 402
cadena, 17, 22, 29, 37, 245, 253
búsqueda, 249, 250
concatenación, 247
subcadena, 248
vacía, 23
cadena[n:m], 248, 266
cadena[n], 248, 266
caja Joe, 399
cálculo, 13
calendario, 265
cambiarFondo, 126, 321, 322
cambiarRojo, 66
cambiarVolumen, 167
cambio de color, 61
campos, 299
canal, 43
canal alfa, 44
capitalize(), 249
carácter, 245
carga, 20
carpetas, 254
cartas modelo, 257
caso promedio, 346
CD, 11
cebollas, 270
células, 377
chromakey, 127, 130, 225, 226, 324
ciclo, 147
for, 54, 55, 75, 161, 165
infinito, 279
o iterar, 161
while, 220, 279, 408
ciclos, 407
por segundo, 147
ciencias computacionales, 12, 335
clase, 378, 384, 402, 411
hija, 383, 402
instancias, 384
NP, 348
P, 347, 348
padre, 383, 402
claves, 300
cliente, 272
CMYK, modelo de color, 43
codificación de longitud de tirada (RLE), 134
codificaciones, 8, 9, 24
código, 17
Código estándar estadounidense para el intercambio de información (ASCII), 10, 23, 246
obtener la asignación, 23
collage, 94
color, 11, 43, 71, 113
distancia, 108
sepia, 113
comandos, 30, 31
comentario, 181
comillas, 22
dobles, 245, 246
sencillas, 245, 246
triple, 245
comparación entre copia y referencia, 92
compilador, 337, 339, 340
complejidad, 377
complemento a dos, 154
compresión, 134, 146
con pérdidas, 41, 134, 212
sin pérdidas, 41, 134, 212
comprimido, 19, 41, 46
computación
de medios, xviii, 10-12
integrada (embedded computing), 351
social, 5
computadora, 8
condiciones límite, 228
conexión, 305
constantes, 5, 72
constructor, 392, 411
consulta, 301
conversión
analógica a digital (ADC), 151
digital a analógica (DAC), 152
coordenadas, 42
cartesianas, 108
copiado, 34, 86
copiar, 184, 319
corchetes, 175
count, 266
count(algo), 251
crearAtardecer, 64
Crowther, William, 218
CSound, 215
cuadrado, 208
cuadros
clave, 314
por segundo (fps), 314
cuerpo de la función, 31
cursiva, 398
cursor, 305

## D

datetime, 265
datos, 17, 359
declaración de una variable, 26
def, 31, 34, 37
definición de funciones, 31
delimitador de ruta, 24, 262
delimitadores, 247
depuración, 166, 169, 217, 226, 228, 229
uso de instrucciones print, 85
desacoplada, 221
descargado, 259
descomposición del problema, 295
descomposición jerárquica, 65
destino, 86, 292
detección de bordes, 120
diccionarios, 302, 410
diente de sierra, 148
difuminado, 118, 120
Digital, 11
digitalización, 11
digitalización de medios

¿por qué?, 12
color, 43
imágenes, 42
sonidos, 151
direcciones IP, 271
directorio
de servidor, 272
raíz, 254
directorios, 18, 254
disco duro, 18, 351
discos, 18
diseño, 216
abajo-arriba, 217, 225
algoritmo y, 235
distancia, 108
euclidiana, 108
división, 150, 248
Doctor, 265
doctype, 290
dominio, 377
de frecuencia, 150
del tiempo, 150
Dreamfall, 218
drop, 388
DVD, 314
Dynabook, 14

## E

eco, 196
editor, 19
de sonido, 149
educación liberal, 13
efectos colaterales, 27, 65, 84, 363
El Padrino, 161
elegirSalon, 220-222, 227, 233
elemento, 155
de imagen, 11, 42
elfo, 367
elif, 239, 282
Eliza, 265
else, 136, 355, 370, 373
else:, 373
email, 265
empalme, 177
en cascada, 250
endswith, 266
endswith(sufijo), 249
ensamblador, 336
entero, 17, 22, 29, 37, 200
escritura, 22
entorno, 8
entradas, 23, 31
enunciados aleatorios, 264
envoltura, 215
errores, 157
bugs, 217, 229
comparación entre experto y normal, 157
escala de grises, 46, 69
escalaGrisesNuevo, 115
escalas, 100
escapes
de barra diagonal inversa, 247
de cadenas, 409
escribirCuadro(), 320
esDirectorio, 370
espacio, 342
espacio de intercambio, 352
especialización, 402
especificación, 135
estado, 361
estados de información, 170
esteganografía, 284
estilo, 253
estructuras de datos, 4, 236
Ethernet, 271
etiquetas, 290
evaluación, 29, 34
execute, 305
explore(imagen), 56
expresión, 22, 28, 30
lógica, 108
extensión de archivo, 24

## F

Facebook, 4, 5, 273
falso, 249
fetchone(), 306
ficción interactiva, 241
filter, 373
filter(), 359
filtro, 211
find, 266
find(), 250
find(cad), 249
find(cad,inicio), 249
find(cad,inicio,fin), 249
física, color, 11
Flash, 133
float, 286
float(), 279
flujo de control, 63
for, 236, 407
for pixel in getPixels(imagen), 54
formato de película, 314
frecuencia, 146, 147, 152
en voz, 170
from media import *, 184, 236
from miSonido import *, 185
from módulo import *, 263
from módulo import nombre, 262
FTP, 272
ftplib, 236, 276, 286
fuente, 253
cómo se define, 133
fuertemente tipificado, 18
función
de copia general, 97
int, 199
funciones, 22, 29, 63, 64, 355, 359, 407
argumentos, 34
ayudantes, 122
cuándo crear una, 36
definición, 31
granularidad, 356
invocación, 29
lambda, 360
llamadas, 29
numéricas, 409
parámetros, 34
valores de entrada, 34
variables de entrada, 34
funciones utilitarias, 82, 296
reutilización, 28
fundamental, 148

## G

general, 63
generan, 339
getBlue, 71, 395, 403
getColor, 48, 71
getGreen, 71, 395, 403
getHeight, 47, 71
getLength, 171
getMediaPAth(nombreArchivoBase), 213
getMediaPath(nombreBase), 82, 105, 194
getMediaPath, 83, 194

getPixel, 47, 71
getPixels, 47, 54, 71, 76
getRed, 48, 71, 395, 403
getSampleObjectAt, 172
getSamples, 156, 161, 171
getSampleValue, 156
getSampleValueAt, 157, 172, 174
getSamplingRate(destino), 178
getSamplingRate, 158, 171
getSound, 172
getters, 393
getValue, 172
getWidth, 47, 71
getX, 47, 71, 129
getY, 47, 71, 129
GIF, 24, 134
gigabyte, 351
gigahertz (GHz), 337
girar imágenes, 98
glifo, 247
global, 58, 190, 192, 230, 355, 372, 373, 377
Gödel, Kurt, 284
Goldberg, Adele, 377, 399
Google, 7, 260
  Chrome, 12
  News, 296
gopher, 273
grabaciones
  análogas, 212
  digitales, 212
  fonográficas, 11
gráficos, 5
Guerra Fría, 272

## H

hardware, 151
  de la computadora, 151
heredados, 402
heredar, 383
herencia, 400, 402
herramienta
  de dibujo, 135
  de imagen, 51
  para pintar, 135
Hertz, 147
heurística, 349, 354
hexadecimal, 247, 291
Hipertexto, 272
Hiragana, 247
Hitchiker's Guide to the Galaxy, 218
hojas, 254
  de estilo en cascada, 289
HSB, modelo de color, 43
HSV, modelo de color, 43
HTML
  scraping, 276
HTTP, 272

## I

identificador, 18
if, 108, 201, 207, 213, 370, 408
  else, 136
imag, 26
imagen, 25, 26, 37, 41, 71
  convertir en sonido, 281
  makePicture(), 25, 26
  recorte, 316
  show, 26
imágenes, dibujar sobre, 131
import, 221, 261, 262
import módulo as nuevonombre, 263
importación, 184
importar un módulo, 261
imprimirTodosLosArchivos, 370
in, 304
índice, 155, 174
  de muestra, 201
indiceArchivo, 279
índices, 299
indiceSonido, 279, 281
Industrial Light & Magic, 7
Inform, 241
Ingalls, Dan, 173
ingeniería de software, 4, 13, 216
inicializa, 392
insert, 236
instancia, 391
int, 171, 199, 200, 213, 279
Intel, 10
inteligencia, 5
  artificial, 5, 265, 349
intensidad, 69, 147
intercambiar, 352
interfaces humano-computadora, 5
interfaz, 5
  digital de instrumentos musicales, 212
Internet, 271
interpretado, 337
intérprete, 338
intervalo
  de muestreo, 202
  de tono, 148
intratables, 348
invocación a una función, 367
isalpha, 266
isalpha(), 250
isdigit, 266
isdigit(), 250

## J

JAR, archivo, 305
Java, 6, 8, 18, 340, 402
java.awt.event, 263
java.io.File, 370
jerarquía, 65
JES, 8, 19, 159, 184, 341, 349
  área de comandos, 20
  área del programa, 20
  carga, 20
  computación de medios en, 21
  ejecución de programas fuera de, 236
  ejecución lenta, 20
  error común, 32
  funciones de medios, 184
  herramienta de imagen, 52
  inicio, 20
  MediaTools, 149
  programación en, 19, 20
  se ejecuta con lentitud, 20
  selector de colores en, 45
  sistema de ayuda, 50
JMV, 314
JPEG, 24, 41, 48, 134, 285
juego de aventuras, 218, 232
jugarJuego, 221
jugarJuego(), 224
Jython, 8, 19, 236, 340

## K

Kaehler, Ted, 173
Katakana, 247
Kay, Alan, 173, 377, 399
kilobytes, 46

## L

lambda, 360, 361
len(), 247
lenguaje
  de marcado, 290
  de programación, 6, 17
  ensamblador, 336
  máquina, 336, 341
Lenguaje de marcación de hipertexto (HTML), 245, 246, 272, 289, 290, 294
Ley de Moore, 10
lineasHorizontales, 131
lineasVerticales, 131
Lisp, 7
lista, 251, 282, 410
  búsqueda, 304
listas
  desordenadas, 293
  ordenadas, 293
listdir(), 262
Load Program (cargar programa), botón, 20
local, 58, 188
localizadores uniformes de recursos, 272
lower, 266
lower(), 250
luminancia, 42, 44, 69, 70, 326
luz visible, 43

## M

mailto, 273
makeColor, 48, 69, 72
makeDarker, 72
makeEmptyPicture, 103, 316
makeEmptyPicture(anchura, altura), 105
makeEmptySound, 178, 179, 203
makeEmptySound(longitudEn Muestras), 178
makeEmptySoundBySeconds, 281
makeLighter, 68, 72
makePicture, 25, 37, 47
  lo que hace, 47
makeSound, 27, 37, 156, 171
  lo que hace, 156
makeSound(), 248
makeStyle(), 398
makeStyle(fuente,énfasis,tamaño), 398
makeTurtle, 378
makeWorld, 378
Maloney, John, 173
manipulación de medios, 363
manipular medios, 12
mantenimiento, 217
map, 359, 373, 393
mapas de bits, 41
Máquina
  de Turing, 348
  virtual, 340, 341
  virtual de Java, 340
marcador de posición, 66
máscara, 130
math, 265
matriz, 41
max, 168, 171, 251, 266
máximo, encontrar, 168
MediaTools, 51, 53, 149, 159
  aplicación, 53
  herramientas de imagen, 51
  herramientas de sonido, 149
  JES MediaTools, 159
Medios digitales, 11
megabyte, 46
megahertz (MHz), 337
mejor caso, 345
Memex, 273
memoria, 9, 19, 20, 351
  caché, 351
  de acceso aleatorio (RAM), 351
métodos, 248, 253, 377, 383, 391
  de cadenas, 409
  funcionales, 411
  setters y getters, 393
mezcla, 235
  imágenes, 122, 123
  sonidos, 195
Microsoft
  Internet Explorer, 12
  PowerPoint, 12
  Word, 12
MIDI, 212
miImagen, 35
min, 251, 266
modelo, 377
módem, 270
modo
  experto, 157
  puro, 247
modulación por codificación de pulsos (PCM), 154
modulador-demodulador, 270
módulo, 62, 261, 410
mono, 398
montaje, 313
Monty Python, 19
Moore, Gordon, 10
mostrarIntroduccion, 220, 221
mostrarSalon, 220-222, 233
mostrarSE, 222
MP3, 212
MP4, 212
MPEG, 314
MPEG-3, 212
muestras, 152, 170, 190
muestreo, 100, 200-202, 235
multiplicador, 168
música por computadora, 5
Myro, 236
MySQL, 305

## N

navegador, 273, 291
navegadores
  Braille, 292
  de audio, 292
negativo de la imagen, 68
negociación (handshake), 271
negrita, 398
Negroponte, Nicholas, 244
Nelson, Ted, 272
nivel de presión de sonido, 147
nodo, 254
nombre
  base, 253
  de archivo, 24, 37
  de archivo base, 24
  de archivo completo, 23
  global, 23
  local, 23
nombre de archivo
  ruta, 18
nombres de dominio, 271
not, 360
notación de complemento a dos, 153
normalización, 168
notación punto, 249, 379, 391
núcleos, 350

nudos, 4
número de punto flotante, 14, 22, 37
Nyquist, teorema de, 152
  aplicaciones, 153

## O

objeto
  de sonido, 155
  referencia, 388
objetos, 156, 248, 253, 381, 401
  clase, 378
  creación, 378
  de muestra, 156
  enviar mensajes a, 379
  imagen, 41
  rehusarse, 381
ocultar información en una imagen, 284
ogros, 270
ojo rojo, 111
ondas
  cuadradas, 207
  senoidales, 146, 203, 205
  triangulares, 210
open(), 255
open(nombrearchivo,cómo), 255
operaciones de secuencia, 409
operadores y funciones de representación, 408
optimización, 239, 344
ord, 37
ord(), 246
ord (de ordinal), 23
ordenamiento de burbuja, 343
ordenar, 251, 266
ordinal, 23
origen, 86
os, 262, 265, 266
  con la clase File de Java, 370
os.listdir, 324
os.listdir(), 262

## P

página
  de miniaturas, 296
  índice, 296
palabras reservadas, 18, 406
Papert, Seymour, 378
paquete, 270, 272
paralelo, 350
parámetro, 188
parámetros, 112
  de alcance, 188
PCM, 154
pegado, 34
películas, 313, 314
penDown(), 382
pensamiento computacional, 13
penUp(), 382
peor caso, 346
percepción, 44
Perlis, Alan, 13
perseguir, 388
persistencia de la visión, 313
Photoshop, 341, 349
pickAColor, 48, 72
pickAFile, 23, 27, 33, 37, 171
pico, 150
Pila, 366
pixelación, 118
píxeles, 11, 41, 71
pixelización, 43, 104
plain, 398
play, 27, 37, 159, 171
playAtRate, 171
playAtRateDur, 171
playMovie(película), 315
playNote, 213
polimorfismo, 395, 401
POP, 272
porcentaje, 53
Portal, 218
posterización, 115
Postscript, 133
Premio Turing de la ACM, 13, 377
presentación con diapositivas, orientada a objetos, 390
primerArchivo, 329
print, 22, 37, 219, 229, 236
print show(imagen), 27
printNow, 219, 222, 224, 229, 238
private, 393
Problema de la parada, 348
Problema del vendedor ambulante, 344
procedimientos, 218
procesador, 336
  núcleo, 350
proceso, 3, 8, 13
programa, 3, 4, 8, 16, 17, 31, 64, 139, 257
  definición, 17
  general, 34
  prueba, 60
programación, 217
  imperativa, 361
  orientada a objetos, 376, 377
  orientada a sustantivos, 377
  por procedimientos, 361, 376
programación funcional, 361, 369
  para medios, 362
propiedades emergentes, 5
protocolo, 270
Proveedor de servicios de Internet (ISP), 271
prueba, 164, 217, 226, 228
  de la caja de cristal, 226
  de la caja negra, 226
  de métodos, 169
  métodos, 164
  uso de instrucciones print, 85
psicoacústica, 147
pulsos, 350
Python, 6-8, 19, 32, 236
  definición de funciones, 31
  extensión para, 32
  uso de mayúsculas, 23
Python 3.0, 22

## Q

quicksort, 343
QuickTime, 314

## R

"rb" y "wb", 255
raise, 394
raíz, 254
random, 263, 265, 266
random(), 263
random.choice(), 263
range, 75, 105, 175, 191, 407
  intervalo negativo, 137
rangos de datos, 6
rarefacciones, 146
rastrear, 57
rastreo, 166
  uso de instrucciones print, 85
raw_input, 220
read, 255
receta, 3, 5, 8, 31, 64

computacional, 3, 8
recorrer, 369
recorrido, 57
recortar, 184, 316, 319
recorte, 89, 154, 169
recursividad, 364, 369
red, 4
redefine, 401
reduce, 361, 373
reduce(), 360
reducirRojo, 372
reducirRojoPorIndice, 373
reducirVolumen, 166
reemplazo de un color con otro, 108, 109
referencia, 388
referencias, 92
refinación de requerimientos, 217
reflejo, 187
reflejo de imagen, 78
reglas, 264
regresar, 62
remove, 266
remove(algo), 251
rEnNombre(), 360
repaint, 48
repisa, 300
replace, 266
replace(búsqueda,reemplazo), 250
representación vectorial, 133
representaciones gráficas de mapas de bits, 134
requerimientos, 217
requestInteger, 238
requestIntegerInRange, 238
requestNumber, 238
requestString, 219, 233, 238
restoArchivos, 329
resultados, 29
retroceder, 259
return, 66, 84, 167
reutilizable, 63
reutilización de código, 184
reutilizar, 377
reverse(), 251
reverse, 266
rfind(buscarcadena), 250
RGB, modelo de color, 43
Roads, Curtis, 173, 215
robustez, 377
rojo, 11
router, 271
rowcount, 306
ruido, 150, 211
ruta, 18, 24, 254, 272

## S

salidas, 23
sangría, 32
sansSerif, 398
Scheme, 6, 7
scraping, 276
  pantalla, 276
  Web, 276
screen scraping, 276
SDRAM, 351
secuencia, 76, 176
selección, 108
self, 383, 391
señal analógica, 151
separador de ruta, 24
serif, 398
servicios, 377
servidor, 272
servidores de nombres de dominio, 271
setBlue, 71, 395, 403
setColor, 46, 68, 71
setColor(color), 382
setGreen, 71, 395, 403
setLibPath(ruta), 185
setLibPath, 185, 236
setMediaPath(directorio), 105
setMediaPath, 83, 105, 194, 213
setPenWidth(anchura), 382
setPixel(), 316
setRed, 48, 62, 71, 395, 403
setSampleValue, 156, 172
setSampleValueAt, 158, 172, 175
setters, 393
setValue, 172
setVisible(false), 382
SGML, 289
Shannon, Claude, 284
show, 25, 36, 37, 47, 394
showInformation, 240
showVars, 230, 238
símbolo, 18
símbolos, 16
SimpleHTTPServer, 265
simulación, 264, 377
simulaciones, 388
sin, 265
sintaxis, 31
síntesis
  aditiva, 203
  FM o síntesis por modulación de frecuencias, 211
  sustractiva, 211
sintetizadores, 198
sistema
  de archivos, 18
  operativo, 18
sistemas, 5
  inteligentes, 5
Smalltalk, 377, 402
SMTP, 272
software, 4
sonido, 37, 145, 190
  al revés, 186
  amplitud, 146, 147
  aumentar el volumen, 161
  codificar, 151
  cómo manipular, 156
  decibeles, 147
  empalme, 177
  frecuencia, 146
  intensidad, 147
  makeSound(), 27
  manipulación, 156
  normalización, 167
  nuevo, 179
  play(), 27
  reducir volumen, 166
  tono, 147, 148
  visualización, 280
  volumen, 147, 161, 166
sonidos, 170
spam, 257
split, 266
split(delimitador), 252
SQL, 304
  cursor, 305
sqrt, 265
Squeak, 6, 173
src=, 292
startswith(prefijo), 249
startswith, 266
Stop, botón, 21
str, 277, 286
subcadena, 248
subclase, 383

subfunciones, 65, 218, 221, 295
subíndice, 176
subprograma, 180
subreceta, 168, 180
  copia de arreglos, 180
  muestreo, 200
sufijo de archivo, 253
Sun, 20
superclase, 383
sustitución, 29, 34
sustracción de fondo, 125, 225, 226
swapcase(), 250
sys.path, 236

## T

T, 6, 7
tabla hash, 302
tabulador, 32
tamaños de muestra, 153, 169,
teclados de muestreo, 198
teléfonos celulares, 4
teletipo, 317
temperamento igual, 147
Teorema de incompletitud, 284
teoría, 5
teoría de la información, 284
terabyte, 351
texto
  con formato, 252
  delimitado por comas, 252
  delimitado por tabuladores, 252
  estilo, 398
  estructurado, 251
tiempo, 342
  real, 149
Tim Berners-Lee, 273
tipos, 16, 18, 22
  arreglo de enteros, 17
  byte, 17
  cadena, 17
  definición, 18
  entero, 17
  punto flotante, 18
tiradas de estilo, 253
title(), 250
tono sepia, 113
Tortuga
  Confundida, 385
  turnToFace, 386
TortugaChica, 383, 384
TortugaConfundida, 385
tortugas, 378
  drop, 388
  enseñarles nuevos trucos, 383
trabajo en red, 5, 270
transformación, 86
transformada de Fourier, 150
transistor, 10
transparencia, 44
true, 249
TrueType, 133
tupla, 306
Turing, Alan, 348
turnToFace, 386
Twitter, 5

## U

umbral, 328
Unicode, 246
unimedia, 244, 277
unión, 301, 303
UNIX, shell, 237
upper, 250, 266
URL, 272
urllib, 236, 273, 275, 286
uso de mayúsculas, 23

## V

valMuestra, 211
valor
  absoluto, 121
  de umbral, 113
valores lógicos, 360
  al crear películas, 326
van Rossum, Guido, 19
variable de instancia, 306, 377, 391, 400
  privadas, 393
  setters y getters, 393
variables, 19, 26, 34, 406
velocidad
  de cuadro, 313
  de reloj, 350
velocidad de muestreo, 153, 158
  en síntesis aditiva, 204
ventana (de tiempo), 150
verbos, 376
verde, 11
vibrato, 196
video, 313, 326
  de origen, 322
  manipulación, 323
  proceso abajo-arriba, 326
vinculación, 28
vista
  de espectro, 150
  de señal, 149
visualización del sonido, 245, 280
volumen, 147, 166
volverse digital, 11

## W

"wt", 255
Wallace, Scott, 173
Watcher, 21, 230
WAV, 24, 161, 212
Web, 272, 273
Web scraping, 276
Weizenbaum, Joseph, 265
while, 238, 286
Wi-Fi, 271
Wing, Jeannette, 13
Woods, Don, 218
World Wide Web, 5, 245, 272
writePictureTo, 48, 71
writeSoundTo, 157, 171
writeTo(), 395
writeTo(nombrearchivo), 395
WWW, 245, 272

## X

XHTML, 289
XML, 289

## Z

zipfile, 265
Zork, 218

# Introducción a la computación y programación con Python

Esta nueva edición de *Introducción a la computación y programación con Python* se enfoca en la manipulación de imágenes, sonido, texto y películas, tal como lo harían los profesionales, pero con programas escritos por estudiantes. La mayoría de las personas utiliza aplicaciones de nivel profesional para realizar este tipo de manipulaciones, pero saber cómo escribir programas propios significa que se puede hacer más de lo que cualquier aplicación permite; de esta manera el poder de expresión no se verá limitado por la capacidad del software que se utilice.

La metodología de *computación en los medios* que maneja este libro parte de las razones por las que muchas personas utilizan las computadoras: manipular imágenes, explorar en busca de música digital, ver y crear páginas Web o realizar videos; y explica la programación y la computación en términos de estas actividades.

Entre las múltiples actualizaciones y mejoras de esta edición sobresalen las siguientes.

- Más términos de ciencias de la computación en los capítulos iniciales.
- Una presentación más temprana y con mayor detalle de las instrucciones de condición, incluyendo *else* y *elif*.
- Una sección sobre funciones y parámetros, donde se explica cuándo usar retorno y cuándo no.
- Una explicación más profunda sobre cómo funcionan las variables, en especial con respecto a los objetos.
- Más información sobre el reflejo de imágenes, con un ejemplo más generalizado.
- Ejemplos Web actualizados con referencias para acceder a sitios interesantes y modernos.
- Más información sobre las diferencias entre los formatos de imágenes.
- Una actualización de la sección sobre hardware y redes para relacionarla con el hardware más reciente, incluyendo los procesadores multinúcleo y los teléfonos móviles.
- Una definición más clara de lo que puede hacerse en Jython y CPython, en comparación con JES.

**Para obtener más información sobre este libro, visite:**

**www.pearsonenespanol.com/guzdial**

ISBN 978-607-32-2049-1

Visítenos en:
www.www.pearsonenespañol.com