Piensa en Python

Aprende a pensar como un informático

2da Edición, Versión 2.4.0

Piensa en Python

Aprende a pensar como un informático

2da Edición, Versión 2.4.0

Allen Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

La forma original de este libro está en código fuente de LaTeX. La compilación de esta fuente de LaTeX tiene el efecto de generar una representación de un libro de texto independiente del dispositivo, la cual se puede convertir a otros formatos e imprimir.

La fuente de LaTeX para este libro está disponible en `http://www.thinkpython2.com`

Título original: *Think Python: How to Think Like a Computer Scientist*

Traducción de Jorge Espinoza.

Prefacio

La extraña historia de este libro

En enero de 1999, me estaba preparando para enseñar un curso introductorio de programación en Java. Lo había enseñado tres veces y me estaba frustrando. La tasa de fracaso en el curso era muy alta y, aún para estudiantes que tenían éxito, el nivel general de logros era muy bajo.

Uno de los problemas que vi tenía relación con los libros. Eran muy grandes, con demasiados detalles innecesarios sobre Java, y no tenían suficiente orientación de alto nivel acerca de cómo programar. Y todos sufrían el efecto trampilla: comenzaban con facilidad, seguían gradualmente y luego, en algún lugar alrededor del Capítulo 5, se caían. Los estudiantes conseguían demasiado material nuevo, muy rápido, y yo ocupaba el resto del semestre recogiendo los pedazos.

Dos semanas antes del primer día de clases, decidí escribir mi propio libro. Mis objetivos eran:

- Que sea corto. Es mejor para los estudiantes leer 10 páginas que no leer 50 páginas.
- Tener cuidado con el vocabulario. Intenté minimizar la jerga y definir cada término en el primer uso.
- Construir de manera gradual. Para evitar trampillas, tomé los temas más difíciles y los dividí en series de pasos pequeños.
- Concentrarse en la programación, no en el lenguaje de programación. Incluí el mínimo subconjunto útil de Java y excluí el resto.

Necesitaba un título, así que por capricho escogí *Aprende a pensar como un informático*.

Mi primera versión fue áspera, pero funcionó. Los estudiantes hicieron la lectura y entendieron lo suficiente como para que yo pudiera ocupar el tiempo de la clase en los temas difíciles, los temas interesantes y (más importante) dejar a los estudiantes practicar.

Publiqué el libro bajo la Licencia de documentación libre de GNU, la cual permite a los usuarios copiar, modificar y distribuir el libro.

Lo que ocurrió después es la parte genial. Jeff Elkner, un profesor de escuela secundaria en Virginia, adoptó mi libro y lo tradujo a Python. Él me envió una copia de su traducción y yo tuve la experiencia inusual de aprender Python leyendo mi propio libro. Como Green Tea Press, publiqué la primera versión en Python en 2001.

En 2003 empecé a enseñar en el Olin College y tuve que enseñar Python por primera vez. El contraste con Java fue notable. Los estudiantes se esforzaban menos, aprendían más, trabajaban en más proyectos interesantes y generalmente se divertían mucho.

Desde entonces he contiunado desarrollando el libro, corrigiendo errores, mejorando algunos de los ejemplos y agregando material, especialmente ejercicios.

El resultado es este libro, ahora con el título menos grandioso *Piensa en Python*. Algunos de los cambios son:

- Agregué una sección sobre depuración al final de cada capítulo. Estas secciones presentan técnicas generales para encontrar y evitar errores de programación y advertencias sobre trampas de Python.
- Agregué más ejercicios, que van desde pruebas cortas de comprensión hasta algunos proyectos sustanciales. La mayoria de los ejercicios incluyen un enlace a mi solución.
- Agregué una serie de estudios de caso: ejemplos más largos con ejercicios, soluciones y discusión.
- Expandí la discusión de planes de desarrollo de programa y pautas de diseño básicas.
- Agregué apéndices sobre depuración y análisis de algoritmos.

La segunda edición de *Piensa en Python* tiene nuevas características:

- El libro y todo el código de apoyo han sido actualizados a Python 3.
- Agregué unas pocas secciones, y más detalles en la web, para ayudar a los principiantes a empezar a ejecutar Python en un navegador, así que no tienes que lidiar con la instalación de Python hasta que quieras hacerlo.
- Para el Capítulo 4.1 cambié mi propio paquete de gráfica tortuga, llamado Swampy, por un módulo de Python más estándar, `turtle`, que es más fácil de instalar y más poderoso.
- Agregué un nuevo capítulo llamado "Trucos extra", el cual introduce algunas características adicionales de Python que no son estrictamente necesarias, pero a veces son prácticas.

Espero que disfrutes trabajando con este libro y que te ayude a aprender a programar y a pensar como un informático, al menos un poco.

Allen B. Downey

Olin College

Agradecimientos

Muchas gracias a Jeff Elkner, quien tradujo mi libro de Java a Python, lo cual comenzó este proyecto y me presentó lo que ha resultado ser mi lenguaje favorito.

Gracias también a Chris Meyers, quien contribuyó a varias secciones de *How to Think Like a Computer Scientist*.

Gracias a la Free Software Foundation por desarrollar la Licencia de documentación libre de GNU, la cual me ayudó a hacer posible mi colaboración con Jeff y Chris, y a Creative Commons por la licencia que uso ahora.

Gracias a los editores de Lulu que trabajaron en *How to Think Like a Computer Scientist*.

Gracias a los editores de O'Reilly Media que trabajaron en *Think Python*.

Gracias a todos los estudiantes que trabajaron con las primeras versiones de este libro y a todos los colaboradores (nombrados a continuacion) que enviaron correcciones y sugerencias.

Lista de colaboradores

Más de 100 lectores perspicaces y atentos han enviado sugerencias y correcciones en los últimos años. Sus contribuciones, y su entusiasmo por este proyecto, han sido una ayuda enorme.

Si tienes una sugerencia o corrección, por favor envía un email a `feedback@thinkpython.com`. Si hago un cambio basado en tu retroalimentación, te agregaré a la lista de colaboradores (a menos que pidas que te omita).

Si incluyes al menos una parte de la oración en donde aparece el error, eso me facilita la búsqueda. Los números de página y de sección también están bien, pero no es tan fácil trabajar estos. ¡Gracias!

- Lloyd Hugh Allen envió una corrección a la Sección 8.4.
- Yvon Boulianne envió una corrección a un error semántico en el Capítulo 5.
- Fred Bremmer envió una corrección en la Sección 2.1.
- Jonah Cohen escribió los scripts de Perl que convierten la fuente de LaTeX de este libro en un hermoso HTML.
- Michael Conlon envió una corrección gramatical en el Capítulo 2 y una mejora de estilo en el Capítulo 1, e inició la discusión sobre los aspectos técnicos de los intérpretes.
- Benoît Girard envió una corrección a un error chistoso en la Sección 5.6.
- Courtney Gleason y Katherine Smith escribieron `horsebet.py`, que fue usado como un estudio de caso en una versión anterior del libro. Su programa encontrarse ahora en el sitio web.
- Lee Harr envió más correcciones de las que cabrían acá en una lista y, de hecho, debería aparecer como uno de los principales editores del texto.
- James Kaylin es un estudiante que usa el texto. Ha enviado numerosas correcciones.
- David Kershaw arregló la función incorrecta `catTwice` en la Sección 3.10.
- Eddie Lam ha enviado numerosas correcciones a los Capítulos 1, 2 y 3. También arregló el Makefile para que cree un índice la primera vez que se ejecute y nos ayudó a configurar un esquema de versionamiento.

- Man-Yong Lee envió una corrección al código de ejemplo en la Sección 2.4.
- David Mayo advirtió que la palabra "inconsciente" en el Capítulo 1 necesitaba ser cambiada a "subconsciente".
- Chris McAloon envió muchas correcciones a las Secciones 3.9 y 3.10.
- Matthew J. Moelter ha sido un colaborador por mucho tiempo que envió numerosas correcciones y sugerencias al libro.
- Simon Dicon Montford informó sobre una definición de función faltante y muchos errores tipográficos en el Capítulo 3. Además, encontró errores en la función `increment` en el Capítulo 13.
- John Ouzts corrigió la definición de "valor de retorno" en el Capítulo 3.
- Kevin Parks envió valiosos comentarios y sugerencias en cuanto a cómo mejorar la distribución del libro.
- David Pool envió un error tipográfico en el glosario del Capítulo 1, así como amables palabras de aliento.
- Michael Schmitt envió una corrección al capítulo de archivos y excepciones.
- Robin Shaw señaló un error en la Sección 13.1, donde la función printTime se usó en un ejemplo sin estar definida.
- Paul Sleigh encontró un error en el Capítulo 7 y un error en el script de Perl de Jonah Cohen que genera HTML a partir de LaTeX.
- Craig T. Snydal está probando el texto en un curso en la Drew University. Ha aportado muchas sugerencias y correcciones valiosas.
- Ian Thomas y sus alumnos están usando el texto en un curso de programación. Ellos son los primeros en probar los capítulos de la segunda mitad del libro, y han hecho numerosas correcciones y sugerencicas.
- Keith Verheyden envió una correción en el Capítulo 3.
- Peter Winstanley nos hizo saber sobre un error en nuestro Latin que estuvo por mucho tiempo en el Capítulo 3.
- Chris Wrobel hizo correcciones al código en el capítulo de entrada/salida de archivo y excepciones.
- Moshe Zadka ha hecho contribuciones invaluables a este proyecto. Además de escribir el primer borrador del capítulo de Diccionarios, proporcionó constante orientación en las primeras etapas del libro.
- Christoph Zwerschke envió muchas correcciones y sugerencias pedagógicas, y explicó la diferencia entre *gleich* y *selbe*.
- James Mayer nos envió una gran cantidad de errores ortográfico y tipográficos, incluyendo dos en la lista de colaboradores.
- Hayden McAfee encontró una inconsistencia potencialmente confusa entre dos ejemplos.
- Angel Arnal es parte de un equipo internacional de traductores trabajando en la versión en español del texto. Además, encontró muchos errores en la versión en inglés.

- Tauhidul Hoque y Lex Berezhny crearon las ilustraciones en el Capítulo 1 y mejoraron muchas de las otras ilustraciones.
- Dr. Michele Alzetta encontró un error en el Capítulo 8 y envió algunos comentarios pedagógicos interesantes y sugerencias sobre Fibonacci y Old Maid.
- Andy Mitchell encontró un error tipográfico en el Capítulo 1 y un ejemplo incorrecto en el Capítulo 2.
- Kalin Harvey sugirió una aclaración en el Capítulo 7 y captó algunos errores tipográficos.
- Christopher P. Smith encontró muchos errores tipográficos y nos ayudó a actualizar el libro para Python 2.2.
- David Hutchins encontró un error tipográfico en el Prólogo.
- Gregor Lingl está enseñando Python en una escuela secundaria en Vienna, Austria. Está trabajando en una traducción del libro al alemán y encontró un par de errores malos en el Capítulo 5.
- Julie Peters encontró un error tipográfico en el Prefacio.
- Florin Oprina envió una mejora a `makeTime`, una corrección a `printTime` y un buen error tipográfico.
- D. J. Webre sugirió una aclaración en el Capítulo 3.
- Ken encontró un puñado de errores en los Capítulos 8, 9 y 11.
- Ivo Wever encontró un error tipográfico en el Capítulo 5 y sugirió una aclaración en el Capítulo 3.
- Curtis Yanko sugirió una aclaración en el Capítulo 2.
- Ben Logan envió una serie de errores tipográficos y problemas con la traducción del libro a HTML.
- Jason Armstrong vio la palabra que faltaba en el Capítulo 2.
- Louis Cordier notó un lugar en el Capítulo 16 donde el código no coincidía con el texto.
- Brian Cain sugirió varias aclaraciones en los Capítulos 2 y 3.
- Rob Black envió un montón de correcciones, incluyendo algunos cambios para Python 2.2.
- Jean-Philippe Rey de la École Centrale Paris envió una serie de parches, incluyendo algunas actualizaciones para Python 2.2 y otras mejoras para pensar.
- Jason Mader en la George Washington University hizo una serie de sugerencias y correcciones útiles.
- Jan Gundtofte-Bruun nos recordó que "a error" es un error.
- Abel David y Alexis Dinno nos recordaron que el plural de "matrix" es "matrices", no "matrixes". Este error estuvo en el libro por años, pero dos lectores con las mismas iniciales lo informaron en el mismo día. Extraño.
- Charles Thayer nos animó a deshacernos de los punto y coma que habíamos puesto al final de algunas sentencias y a limpiar nuestro uso de "argumento" y "parámetro".
- Roger Sperberg advirtió sobre una lógica retorcida en el Capítulo 3.

- Sam Bull advirtió sobre un párrafo confuso en el Capítulo 2.
- Andrew Cheung señaló dos instancias de "use before def".
- C. Corey Capel vio la palabra que faltaba en el Tercer Teorema de la Depuración y un error tipográfico en el Capítulo 4.
- Alessandra ayudó a aclarar alguna confusión con Turtle.
- Wim Champagne encontró un error en un ejemplo de diccionario.
- Douglas Wright señaló un problema con la división entera en `arco`.
- Jared Spindor encontró algo de basura al final de una oración.
- Lin Peiheng envió una serie de sugerencias muy útiles.
- Ray Hagtvedt envió dos errores y un no tan error.
- Torsten Hübsch señaló una inconsistencia en Swampy.
- Inga Petuhhov corrigió un ejemplo en el Capítulo 14.
- Arne Babenhauserheide envió muchas correcciones útiles.
- Mark E. Casida es es bueno mirando palabras repetidas.
- Scott Tyler rellenó una que faltaba. Y envió un montón de correcciones.
- Gordon Shephard envió varias correcciones, todas en correos separados.
- Andrew Turner encontró un error en el Capítulo 8.
- Adam Hobart arregló un problema con la división entera en `arco`.
- Daryl Hammond y Sarah Zimmerman advirtieron que serví `math.pi` demasiado pronto. Y Zim vio un error tipográfico.
- George Sass encontró un error en una sección de Depuración.
- Brian Bingham sugirió el Ejercicio 11.5.
- Leah Engelbert-Fenton advirtió que usé `tuple` como un nombre de variable, contrario a mi propio consejo. Y luego encontró un montón de errores tipográficos y un "use before def".
- Joe Funke vio un error tipográfico.
- Chao-chao Chen encontró una inconsistencia en el ejemplo de Fibonacci.
- Jeff Paine sabe la diferencia entre space y spam.
- Lubos Pintes envió un error tipográfico.
- Gregg Lind y Abigail Heithoff sugirieron el Ejercicio 14.3.
- Max Hailperin ha enviado una serie de correcciones y sugerencias. Max es uno de los autores del extraordinario *Concrete Abstractions*, que tal vez quieras leer cuando termines con este libro.
- Chotipat Pornavalai encontró un error en un mensaje de error.
- Stanislaw Antol envió una lista de sugerencias muy útiles.
- Eric Pashman envió una serie de correcciones para los Capítulos 4–11.

- Miguel Azevedo encontró algunos errores tipográficos.
- Jianhua Liu envió una larga lista de correcciones.
- Nick King encontró una palabra que faltaba.
- Martin Zuther envió una larga lista de sugerencias.
- Adam Zimmerman encontró una inconsistencia en mi ejemplo de una "instancia" y muchos otros errores.
- Ratnakar Tiwari sugirió una nota al pie explicando los triángulos degenerados.
- Anurag Goel sugirió otra solución para `es_abecedario` y envió algunas correcciones adicionales. Y sabe cómo deletrear Jane Austen.
- Kelli Kratzer vio uno de los errores tipográficos.
- Mark Griffiths señaló un ejemplo confuso en el Capítulo 3.
- Roydan Ongie encontró un error en mi método de Newton.
- Patryk Wolowiec me ayudó con un problema en la versión HTML.
- Mark Chonofsky me habló de una nueva palabra clave en Python 3.
- Russell Coleman me ayudó con mi geometría.
- Nam Nguyen encontró un error tipográfico y advirtió que usé el patrón Decorator pero sin mencionar el nombre.
- Stéphane Morin envió varias correcciones y sugerencias.
- Paul Stoop corrigió un error tipográfico en `usa_solo`.
- Eric Bronner advirtió sobre una confusión en la discusión del orden de operaciones.
- Alexandros Gezerlis estableció un nuevo estándar para el número y la calidad de sugerencias que envió. ¡Estamos profundamente agradecidos!
- Gray Thomas distingue su derecha de su izquierda.
- Giovanni Escobar Sosa envió una larga lista de correcciones y sugerencias.
- Daniel Neilson corrigió un error sobre el orden de operaciones.
- Will McGinnis advirtió que `polilinea` fue definida de manera diferente en dos lugares.
- Frank Hecker advirtió sobre un ejercicio que estaba subespecificado y algunos enlaces rotos.
- Animesh B me ayudó a limpiar un ejemplo confuso.
- Martin Caspersen encontró dos errores de redondeo.
- Gregor Ulm envió varias correcciones y sugerencias.
- Dimitrios Tsirigkas me sugirió que aclarara un ejercicio.
- Carlos Tafur envió una página de correcciones y sugerencias.
- Martin Nordsletten encontró un error en una solución de un ejercicio.
- Sven Hoexter advirtió que una variable con nombre `input` ensombrece a una función incorporada.

- Stephen Gregory advirtió el problema con `cmp` en Python 3.
- Ishwar Bhat corrigió mi enunciado del último teorema de Fermat.
- Andrea Zanella tradujo el libro al italiano y envió una serie de correcciones en el camino.
- Muchas, muchas gracias a Melissa Lewis y Luciano Ramalho por los excelentes comentarios y sugerencias sobre la segunda edición.
- Gracias a Harry Percival de PythonAnywhere por su ayuda al hacer que la gente comience ejecutando Python en un navegador.
- Xavier Van Aubel hizo muchas correcciones útiles en la segunda edición.
- William Murray corrigió mi definición de división entera.
- Per Starbäck me puso al día sobre nuevas líneas universales en Python 3.
- Laurent Rosenfeld y Mihaela Rotaru tradujeron este libro al francés. En el camino, me enviaron muchas correcciones y sugerencias.

  Adicionalmente, las personas que vieron errores tipográficos o hicieron correcciones incluyen a Czeslaw Czapla, Dale Wilson, Francesco Carlo Cimini, Richard Fursa, Brian McGhie, Lokesh Kumar Makani, Matthew Shultz, Viet Le, Victor Simeone, Lars O.D. Christensen, Swarup Sahoo, Alix Etienne, Kuang He, Wei Huang, Karen Barber y Eric Ransom.

Índice general

Capítulo 1

El camino del programa

El objetivo de este libro es enseñarte a pensar como un informático. Esta forma de pensar combina algunas de las mejores características de las matemáticas, la ingeniería y las ciencias naturales. Al igual que los matemáticos, los informáticos usan lenguajes formales para denotar ideas (específicamente, computaciones). Al igual que los ingenieros, diseñan cosas, ensamblando componentes en sistemas y evaluando compensasiones entre alternativas. Al igual que los científicos, observan el comportamiento de sistemas complejos, a partir de hipótesis, y prueban predicciones.

La habilidad más importante de un informático es la **resolución de problemas**. La resolución de problemas supone la capacidad de formular problemas, pensar creativamente en soluciones y emitir una solución de manera clara y precisa. Como resultado, el proceso de aprender a programar es una excelente oportunidad para practicar habilidades de resolución de problemas. Es por eso que este capítulo se llama "El camino del programa".

En un nivel, aprenderás a programar, una habilidad útil por sí misma. En otro nivel, usarás la programación como un medio para un fin. Mientras avancemos, ese fin se volverá más claro.

1.1. ¿Qué es un programa?

Un **programa** es una secuencia de instrucciones que especifica cómo realizar una computación. La computación puede ser algo matemático, tal como resolver un sistema de ecuaciones o encontrar las raíces de un polinomio, pero también puede ser una computación simbólica, tal como buscar y reemplazar texto en un documento, o algo gráfico, como procesar una imagen o reproducir un video.

Los detalles se ven diferentes en diferentes lenguajes, pero unas pocas instrucciones básicas aparecen en casi todos los lenguajes:

entrada (input): Obtener datos desde el teclado, un archivo, la red u otro dispositivo.

salida (output): Mostrar datos en la pantalla, guardarlos en un archivo, enviarlos a través de la red, etc.

matemáticas: Realizar operaciones matemáticas básicas como la suma y la multiplicación.

ejecución condicional: Verificar ciertas condiciones y ejecutar el código apropiado.

repetición: Realizar alguna acción repetidas veces, generalmente con alguna variación.

Lo creas o no, eso es prácticamente todo lo que hay. Cada programa que has usado, no importa cuán complicado, está compuesto de instrucciones que se parecen mucho a estas. Así que puedes pensar en la programación como el proceso de separar una tarea grande y compleja en subtareas cada vez más pequeñas, hasta que cada subtarea sea lo suficientemente simple para hacerla con una de estas instrucciones básicas.

1.2. Ejecutar Python

Uno de los desafíos de comenzar con Python es que quizás debas instalar Python y software relacionado en tu computador. Si conoces bien tu sistema operativo, y especialmente si manejas bien la interfaz de línea de comandos, no tendrás problemas instalando Python. Sin embargo, puede ser doloroso para los principiantes aprender sobre administración del sistema y programación al mismo tiempo.

Para evitar ese problema, recomiendo que comiences ejecutando Python en un navegador. Después, cuando te familiarices con Python, haré sugerencias para instalar Python en tu computador.

Hay una serie de páginas web que puedes usar para ejecutar Python. Si ya tienes una favorita, ve y úsala. De otra manera, recomiendo PythonAnywhere. En `http://tinyurl.com/thinkpython2e` proporciono instrucciones detalladas para comenzar.

Hay dos versiones de Python, llamadas Python 2 y Python 3. Son muy similares, así que si aprendes una, es fácil cambiar a la otra. De hecho, hay solo unas pocas diferencias que encontrarás como principiante. Este libro está escrito para Python 3, pero incluyo algunas notas sobre Python 2.

El **intérprete** de Python es un programa que lee y ejecuta código de Python. Dependiendo de tu entorno, puedes iniciar el intérprete haciendo clic en un ícono o escribiendo `python` en una línea de comandos. Cuando se inicia, deberías ver una salida como esta:

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Las primeras tres líneas contienen información acerca del intérprete y el sistema operativo en el cual se está ejecutando, así que puede ser diferente para ti. Sin emgargo, deberías verificar que el número de versión, que en este ejemplo es `3.4.0`, comience con 3, el cual indica que estás ejecutando Python 3. Si comienza con 2, estás ejecutando (lo adivinaste) Python 2.

La última línea es un **prompt** que indica que el intérprete está listo para que introduzcas código. Si escribes una línea de código y presionas la tecla Enter, el intérprete muestra el resultado:

```
>>> 1 + 1
2
```

Ahora estás listo para comenzar. A partir de aquí, doy por sentado que sabes cómo iniciar el intérprete de Python y ejecutar código.

1.3. El primer programa

Tradicionalmente, el primer programa que escribes en un nuevo lenguaje de programación se llama "Hola, mundo" porque todo lo que hace es mostrar las palabras "Hola, mundo". En Python, se ve así:

```
>>> print('Hola, mundo')
```

Este es un ejemplo de una **sentencia print**, aunque en realidad no imprime nada en papel. Esta sentencia muestra un resultado en la pantalla. En este caso, el resultado es la frase

```
Hola, mundo
```

Las comillas en el programa marcan el principio y el final del texto a visualizar; estas no aparecen en el resultado.

Los paréntesis indican que `print` es una función. Llegaremos a las funciones en el Capítulo 3.

En Python 2, la sentencia print es un poco diferente; no es una función, así que no usa paréntesis.

```
>>> print 'Hola, mundo'
```

Esta distinción tendrá más sentido pronto, pero eso es suficiente para comenzar.

1.4. Operadores aritméticos

Después de "Hola, mundo", el siguiente paso es la aritmética. Python proporciona **operadores**, los cuales son símbolos especiales que representan computaciones como la suma y la multiplicación.

Los operadores `+`, `-` y `*` realizan sumas, restas y multiplicaciones, como en los siguientes ejemplos:

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

El operador `/` realiza divisiones:

```
>>> 84 / 2
42.0
```

Puedes preguntarte por qué el resultado es `42.0` en lugar de `42`. Lo explicaré en la sección siguiente.

Finalmente, el operador `**` realiza potenciaciones; es decir, eleva un número a una potencia:

```
>>> 6**2 + 6
42
```

En algunos otros lenguajes, `^` se usa para potenciación, pero en Python es un operador bit a bit llamado XOR. Si no conoces los operadores bit a bit, el resultado te sorprenderá:

```
>>> 6 ^ 2
4
```

No cubriré operadores bit a bit en este libro, pero puedes leer sobre estos en `http://wiki.python.org/moin/BitwiseOperators`.

1.5. Valores y tipos

Un **valor** es una de las cosas básicas con las que funciona un programa, como una letra o un número. Algunos valores que hemos visto hasta ahora son `2`, `42.0` y `'Hola, mundo'`.

Estos valores pertenecen a diferentes **tipos**: `2` es un **entero** (en inglés, *integer*), `42.0` es un **número de coma flotante** (en inglés, *floating-point number*), y `'Hola, mundo'` es una **cadena** (en inglés, *string*), llamada así porque las letras que contiene están unidas.

Si no estás seguro de qué tipo tiene un valor, el intérprete te lo puede decir:

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hola, mundo')
<class 'str'>
```

En estos resultados, la palabra "class" se usa en el sentido de una categoría; un tipo es una categoría de valores.

Evidentemente, los enteros pertenecen al tipo `int`, las cadenas pertenecen al tipo `str` y los números de coma flotante pertenecen al tipo `float`.

¿Qué pasa con los valores como `'2'` y `'42.0'`? Se ven como números, pero están en comillas como cadenas.

```
>>> type('2')
<class 'str'>
>>> type('42.0')
<class 'str'>
```

Son cadenas.

Cuando escribas un entero grande, tal vez sientas la tentación de usar comas entre grupos de dígitos, como en `1,000,000`. No es un *entero* legal en Python, pero es legal:

```
>>> 1,000,000
(1, 0, 0)
```

¡Eso no es lo que esperábamos en absoluto! Python interpreta `1,000,000` como una secuencia de enteros separados por comas. Aprenderemos más sobre este tipo de secuencia más adelante.

1.6. Lenguajes formales y lenguajes naturales

Los **lenguajes naturales** son los lenguajes que hablan las personas, tales como el inglés, el español y el francés. No fueron diseñados por las personas (aunque las personas traten de imponerles algo de orden); evolucionan de manera natural.

Los **lenguajes formales** son lenguajes que están diseñados por personas para aplicaciones específicas. Por ejemplo, la notación que usan los matemáticos es un lenguaje formal que es particularmente bueno al denotar relaciones entre números y símbolos. Los químicos usan un lenguaje formal para representar la estructura química de las moléculas. Y más importante:

> **Los lenguajes de programación son lenguajes formales que han sido diseñados para expresar computaciones.**

Los lenguajes formales tienden a tener reglas de **sintaxis** que gobiernan la estructura de las sentencias. Por ejemplo, en matemáticas la sentencia $3 + 3 = 6$ tiene sintaxis correcta, pero $3+ = 3\$6$ no la tiene. En química, H_2O es una fórmula sintácticamente correcta, pero ${}_2Zz$ no.

Hay dos tipos de reglas de sintaxis, relacionadas con los **tokens** y la estructura. Los tokens son los elementos básicos del lenguaje, tales como palabras, números y elementos químicos. Uno de los problemas con $3+ = 3\$6$ es que \$ no es un token legal en matemáticas (al menos por lo que sé). Del mismo modo, ${}_2Zz$ no es legal porque no hay ningún elemento con la abreviatura Zz.

El segundo tipo de regla de sintaxis tiene relación con la manera en que los tokens están combinados. La ecuación $3 + /3$ es ilegal porque aunque $+$ y / son tokens legales, no puedes tener uno justo después del otro. Del mismo modo, en una fórmula químca el subíndice viene después del nombre del elemento, no antes.

Esta es un@ oración en e$paño/ bien estructurada con t*kens inválidos. Esta oración todos los tokens válidos tiene, pero estructura inválida presenta.

Cuando lees una oración en español o una sentencia en un lenguaje formal, tienes que descifrar la estructura (aunque en un lenguaje natural lo haces de manera subconsciente). Este proceso se llama **análisis sintáctico** (en inglés, *parsing*).

Aunque los lenguajes formales y los lenguajes naturales tienen muchas características en común —tokens, estructura y sintaxis— existen algunas diferencias:

ambigüedad: Los lenguajes naturales están llenos de ambigüedad, con la cual las personas lidian mediante el uso de pistas contextuales y otra información. Los lenguajes formales están diseñados para ser casi o completamente inequívocos, lo cual significa que cualquier sentencia tiene exactamente un significado, sin importar el contexto.

redundancia: Para compensar la ambigüedad y reducir los malentendidos, los lenguajes naturales emplean mucha redundancia. Como consecuencia, a menudo son verbosos. Los lenguajes formales son menos redundantes y más concisos.

literalidad: Los lenguajes naturales están llenos de modismo y metáfora. Si yo digo "Cayó en la cuenta", probablemente no hay ninguna cuenta ni nadie cayendo (este modismo significa que alguien entendió algo después de un periodo de confusión). Los lenguajes formales expresan exactamente lo que dicen.

Debido a que todos crecemos hablando lenguajes naturales, a veces es difícil adaptarse a los lenguajes formales. La diferencia entre lenguaje formal y lenguaje natural es como la diferencia entre poesía y prosa, pero más aún:

Poesía: Las palabras se usan por su sonido tanto como por su significado y todo el poema junto crea un efecto o respuesta emocional. La ambigüedad no solo es común sino a menudo deliberada.

Prosa: El significado literal de las palabras es más importante y la estructura aporta más significado. La prosa es más susceptible de análisis que la poesía pero todavía a menudo ambigua.

Programas: El significado de un programa de computador es inequívoco y literal, y puede entenderse enteramente analizando los tokens y su estructura.

Los lenguajes formales son más densos que los lenguajes naturales, así que leerlos requiere más tiempo. Además, la estructura es importante, por lo que no siempre es mejor leer de arriba a abajo y de izquierda a derecha. En su lugar, hay que aprender a analizar sintácticamente el programa en tu cabeza, identificar los tokens e interpretar la estructura. Finalmente, los detalles importan. Pequeños errores en la ortografía y puntuación, que puedes cometer con los lenguajes naturales, pueden hacer una gran diferencia en un lenguaje formal.

1.7. Depuración

Los programadores cometen errores. Por razones caprichosas, los **errores de programación** se llaman *bugs* y el proceso de localizarlos se llama **depuración** (en inglés, *debugging*).

La programación, y especialmente la depuración, a veces provoca emociones fuertes. Si estás luchando con un error de programación difícil, podrías sentir ira, desánimo o vergüenza.

Hay evidencia de que las personas naturalmente responden a los computadores como si estos fueran personas. Cuando funcionan bien, pensamos en ellos como compañeros de equipo, y cuando son obstinados o rudos, respondemos a ellos de la misma manera que respondemos a las personas rudas y obstinadas (Reeves and Nass, *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*).

Prepararse para estas reacciones podría ayudarte a lidiar con ellas. Un enfoque es pensar en el computador como un empleado con ciertas fortalezas, como la velocidad y la precisión, y debilidades particulares, como la falta de empatía y la incapacidad para comprender el panorama general.

Tu trabajo es ser un buen jefe: encontrar maneras de aprovechar las fortalezas y mitigar las debilidades. Y encontrar maneras de usar tus emociones para abordar el problema, sin dejar que tus reacciones interfieran en tu capacidad de trabajar eficazmente.

Aprender a depurar puede ser frustrante, pero es una habilidad valiosa que es útil para muchas actividades más allá de la programación. Al final de cada capítulo hay una sección, como esta, con mis sugerencias para la depuración. ¡Espero que ayuden!

1.8. Glosario

resolución de problemas: El proceso de formular un problema, encontrar una solución y expresarla.

lenguaje de alto nivel: Un lenguaje de programación como Python que está diseñado para que los humanos puedan leer y escribir fácilmente.

lenguaje de bajo nivel: Un lenguaje de programación que está diseñado para que sea fácil de ejecutar por un computador; también llamado "lenguaje de máquina" o "lenguaje ensamblador".

portabilidad: Una propiedad de un programa que puede ejecutarse en más de un tipo de computador.

intérprete: Un programa que lee otro programa y lo ejecuta.

prompt: Caracteres mostrados por el intérprete que indican que está listo para recibir la entrada del usuario.

programa: Un conjunto de instrucciones que especifica una computación.

sentencia print: Una instrucción que hace que el intérprete de Python muestre un valor en la pantalla.

operador: Un símbolo especial que representa una computación simple como suma, multiplicación o concatenación de cadenas.

valor: Una de las unidades básicas de datos, como un número o una cadena, que manipula un programa.

tipo: Una categoría de valores. Los tipos que hemos visto hasta ahora son los enteros (tipo `int`), los números de coma flotante (tipo `float`) y cadenas (tipo `str`).

entero: Un tipo que representa números enteros.

coma flotante: Un tipo que representa números con partes fraccionarias.

cadena: Un tipo que representa secuencias de caracteres.

lenguaje natural: Cualquiera de los lenguajes que hablan las personas y que evolucionaron de manera natural.

lenguaje formal: Cualquiera de los lenguajes que las personas han diseñado para propósitos específicos, tales como representar ideas matemáticas o programas de computador; todos los lenguajes de programación son lenguajes formales.

token: Uno de los elementos básicos de la estructura sintáctica de un programa, análogo a una palabra en un lenguaje natural.

sintaxis: Las reglas que rigen la estructura de un programa.

análisis sintáctico (*parse*): Examinar un programa y analizar la estructura sintáctica.

error de programación (*bug*): Un error en un programa.

depuración (*debugging*): El proceso de encontrar y corregir errores de programación.

1.9. Ejercicios

Ejercicio 1.1. *Es una buena idea leer este libro en frente de un computador para que puedas probar los ejemplos mientras avanzas.*

Cada vez que experimentes con una nueva característica, deberías intentar cometer errores. Por ejemplo, en el programa "Hola, mundo", ¿qué ocurre si omites una de las comillas? ¿Y si omites ambas? ¿Qué ocurre si escribes `print` *de manera incorrecta?*

Este tipo de experimento te ayuda a recordar lo que leíste; también te ayuda cuando estás programando porque logras saber lo que significan los mensajes de error. Es mejor cometer errores ahora y a propósito que después y de manera accidental.

1. *En una sentencia print, ¿qué ocurre si omites uno de los paréntesis, o ambos?*
2. *Si estás intentando imprimir una cadena con print, ¿qué ocurre si omites una de las comillas, o ambas?*
3. *Puedes usar un signo menos para hacer un número negativo como* `-2`*. ¿Qué ocurre si pones un signo más antes de un número? ¿Qué pasa con* `2++2`*?*
4. *En notación matemática, los ceros a la izquierda están bien, como en* `09`*. ¿Qué ocurre si intentas esto en Python? ¿Qué pasa con* `011`*?*
5. *¿Qué ocurre si tienes dos valores sin operador entre ellos?*

Ejercicio 1.2. *Inicia el intérprete de Python y úsalo como una calculadora.*

1. *¿Cuántos segundos hay en 42 minutos con 42 segundos?*
2. *¿Cuántas millas hay en 10 kilómetros? Pista: hay 1.61 kilómetros en una milla.*
3. *Si corres una carrera de 10 kilómetros en 42 minutos con 42 segundos, ¿Cuál es tu ritmo promedio (tiempo por milla en minutos y segundos)? ¿Cuál es tu rapidez promedio en millas por hora?*

Capítulo 2

Variables, expresiones y sentencias

Una de las características más poderosas de un lenguaje de programación es la posibilidad de manipular **variables**. Una variable es un nombre que se refiere a un valor.

2.1. Sentencias de asignación

Una **sentencia de asignación** crea una nueva variable y le da un valor:

```
>>> mensaje = 'Y ahora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897932
```

Este ejemplo hace tres asignaciones. La primera asigna una cadena a una nueva variable llamada `mensaje`; la segunda pone al entero `17` en `n`; la tercera asigna el valor (aproximado) de π a `pi`.

Una forma común de representar en papel las variables es escribir el nombre con una flecha apuntando a su valor. Este tipo de figura se llama **diagrama de estado** porque muestra en qué estado está cada una de las variables (piénsalo como el estado mental de la variable). La Figura 2.1 muestra el resultado del ejemplo anterior.

2.2. Nombres de variable

Los programadores generalmente escogen nombres para sus variables que sean significativos: documentan para qué se usa la variable.

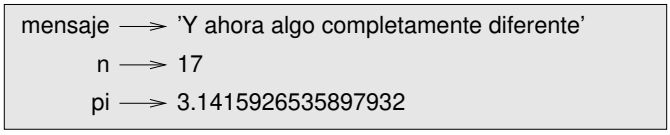

Figura 2.1: Diagrama de estado.

Los nombres de variable pueden ser tan largos como quieras. Pueden contener tanto letras como números, pero no pueden comenzar con un número. Es legal usar letras mayúsculas, pero es convencional usar solo minúsculas para los nombres de variables.

El guión bajo, `_`, puede aparecer en un nombre. A menudo se usa en nombres con varias palabras, tales como `tu_nombre` o `velocidad_de_golondrina_sin_carga`.

Si le das un nombre ilegal a una variable, obtendrás un error de sintaxis:

```
>>> 76trombones = 'gran desfile'
SyntaxError: invalid syntax
>>> mas@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Cimología teórica avanzada'
SyntaxError: invalid syntax
```

`76trombones` es ilegal porque comienza con un número. `mas@` es ilegal porque contiene un carácter ilegal, `@`. Sin embargo, ¿qué tiene de malo `class`?

Resulta que `class` es una de las **palabras clave** de Python. El intérprete usa las palabras clave para reconocer la estructura del programa y no se pueden usar como nombres de variable.

Python 3 tiene estas palabras clave:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

No tienes que memorizar esta lista. En la mayoría de los entornos de desarrollo, las palabras clave se muestran con un color diferente; si intentas usar una como un nombre de variable, lo sabrás.

2.3. Expresiones y sentencias

Una **expresión** es una combinación de valores, variables y operadores. Un valor por sí mismo es considerado una expresión, y por consigiuente es una variable, así que las siguientes son todas expresiones legales:

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

Cuando escribes una expresión en el prompt, el intérprete la **evalúa**, lo cual significa que encuentra el valor de la expresión. En este ejemplo, `n` tiene el valor 17 y `n + 25` tiene el valor 42.

Una **sentencia** es una unidad de código que tiene un efecto, como crear una variable o mostrar un valor.

```
>>> n = 17
>>> print(n)
```

La primera línea es una sentencia de asignación que le da un valor a `n`. La segunda línea es una sentencia print que muestra el valor de `n`.

Cuando escribes una sentencia, el intérprete la **ejecuta**, lo cual significa que hace lo que dice la sentencia. En general, las sentencias no tienen valores.

2.4. Modo Script

Hasta ahora hemos ejecutado Python en **modo interactivo**, lo cual significa que interactúas directamente con el intérprete. El modo interactivo es una buena manera de comenzar, pero si estás trabajando con más que unas pocas líneas de código, puede ser algo torpe.

La alternativa es guardar código en un archivo llamado **script** y entonces usar el intérprete en **modo script** para ejecutar el script. Por convención, los scripts de Python tienen nombres que terminan con `.py`.

Si sabes cómo crear y ejecutar un script en tu computador, estás listo para seguir. De lo contrario, recomiendo de nuevo usar PythonAnywhere. He publicado instrucciones para usarlo en modo script en `http://tinyurl.com/thinkpython2e`.

Debido a que Python proporciona ambos modos, puedes probar pedazos de código en modo interactivo antes de ponerlos en un script. Sin embargo, hay diferencias entre el modo interactivo y el modo script que pueden confundir.

Por ejemplo, si usas Python como una calculadora, puedes escribir

```
>>> millas = 26.2
>>> millas * 1.61
42.182
```

La primera línea asigna un valor a `millas`, pero no tiene un efecto visible. La segunda línea es una expresión, por lo que el intérprete la evalúa y muestra el resultado. Resulta que una maratón es de unos 42 kilómetros.

Sin embargo, si escribes el mismo código dentro de un script y lo ejecutas, no obtienes ninguna salida. En modo script una expresión, por sí misma, no tiene efecto visible. Python evalúa la expresión, pero no muestra el resultado. Para mostrar el resultado, necesitas una sentencia `print` como esta:

```
millas = 26.2
print(millas * 1.61)
```

Este comportamiento puede confundir al principio. Para comprobar tu comprensión, escribe las siguientes sentencias en el intérprete de Python y mira lo que hacen:

```
5
x = 5
x + 1
```

Ahora pon las mismas sentencias en un script y ejecútalo. ¿Cuál es la salida? Modifica el script transformando cada expresión en una sentencia print y luego ejecútalo de nuevo.

2.5. Orden de operaciones

Cuando una expresión contiene más de un operador, el orden de evaluación depende del **orden de operaciones**. Para operadores matemáticos, Python sigue la convención matemática. El acrónimo **PEMDAS** es una manera útil de recordar las reglas:

- Los **P**aréntesis tienen la mayor prioridad y se pueden usar para forzar una expresión a evaluar en el orden que tú quieras. Ya que las expresiones en paréntesis se evalúan primero, `2 * (3-1)` es 4, y `(1+1)**(5-2)` es 8. También puedes usar paréntesis para hacer una expresión más fácil de leer, como en `(minuto * 100) / 60`, incluso si no cambia el resultado.
- Los **E**xponentes de potencias tienen la siguiente proridad, así que `1 + 2**3` es 9, no 27, y `2 * 3**2` es 18, no 36.
- La **M**ultiplicación y la **D**ivisión tienen mayor prioridad que la **A**dición (suma) y la **S**ustracción (resta). Así que `2*3-1` es 5, no 4, y `6+4/2` es 8, no 5.
- Los operadores con la misma prioridad se evalúan de izquierda a derecha (excepto la potenciación). Así que en la expresión `grados / 2 * pi`, la división ocurre primero y el resultado se multiplica por `pi`. Para dividir por 2π, puedes usar paréntesis o escribir `grados / 2 / pi`.

Yo no me esfuerzo mucho en recordar la prioridad de los operadores. Si no puedo saber mirando la expresión, uso paréntesis para hacerlo obvio.

2.6. Operaciones con cadenas

En general, no puedes realizar operaciones matemáticas con cadenas, incluso si las cadenas parecen números, por lo que las siguientes son ilegales:

```
'comida'-'china'    'huevos'/'fácil'    'tercero'*'un encanto'
```

Pero hay dos excepciones, + y *.

El operador + realiza una **concatenación**, lo cual significa que une las cadenas enlazándolas de extremo a extremo. Por ejemplo:

```
>>> primero = 'throat'
>>> segundo = 'warbler'
>>> primero + segundo
throatwarbler
```

El operador * también funciona en cadenas; hace repetición. Por ejemplo, `'Spam'*3` es `'SpamSpamSpam'`. Si uno de los valores es una cadena, el otro tiene que ser un entero.

Este uso de + y * tiene sentido por analogía con la suma y la multiplicación. Tal como `4*3` es equivalente a `4+4+4`, esperamos que `'Spam'*3` sea lo mismo que `'Spam'+'Spam'+'Spam'`, y lo es. Por otro lado, hay una manera significativa en la que la concatenación y la repetición son diferentes de la suma y multiplicación de enteros. ¿Puedes pensar en una propiedad que tiene la suma que la concatenación no tiene?

2.7. Comentarios

A medida que los programas se hacen más grandes y complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos y a menudo es difícil mirar un pedazo de código y descifrar lo que hace, o por qué.

Por esta razón, es una buena idea añadir notas a tus programas para explicar en lenguaje natural lo que hace el programa. Estas notas se llaman **comentarios**, y comienzan con el símbolo #:

```
# calcular el porcentaje de la hora que ha transcurrido
porcentaje = (minuto * 100) / 60
```

En este caso, el comentario aparece en una línea por sí sola. Puedes también poner comentarios al final de una línea:

```
porcentaje = (minuto * 100) / 60     # porcentaje de una hora
```

Todo desde el # hasta el final de la línea es ignorado: no tiene efecto en la ejecución del programa.

Los comentarios son más útiles cuando documentan características del código que no son obvias. Es razonable asumir que el lector puede descifrar *qué* hace el código; es más útil explicar *por qué*.

Este comentario es redundante con el código e inútil:

```
v = 5     # asigna 5 a v
```

Este comentario contiene información útil que no está en el código:

```
v = 5     # velocidad en metros/segundos.
```

Los buenos nombres de variable pueden reducir la necesidad de comentarios, pero los nombres largos pueden hacer que las expresiones complejas sean difíciles de leer, así que hay una compensación.

2.8. Depuración

En un programa pueden ocurrir tres tipos de errores: errores de sintaxis, errores de tiempo de ejecución y errores semánticos. Es útil distinguir entre ellos para rastrearlos de manera más rápida.

Error de sintaxis: La "sintaxis" se refiere a la estructura de un programa y las reglas sobre esa estructura. Por ejemplo, los paréntesis tienen que venir en pares que coincidan, por lo que `(1 + 2)` es legal, pero `8)` es un **error de sintaxis**.

Si hay un error de sintaxis en cualquier lugar de tu programa, Python muestra un mensaje de error y se detiene, y no podrás ejecutar el programa. Durante las primeras semanas de tu carrera de programación, podrías pasar mucho tiempo rastreando errores de sintaxis. A medida que ganes experiencia, cometerás menos errores y los encontrarás más rápido.

Error de tiempo de ejecución: El segundo tipo de error es un error de tiempo de ejecución, llamado así porque el error no aparece hasta después que el programa ha comenzado a ejecutarse. Estos errores también se llaman **excepciones** porque usualmente indican que algo excepcional (y malo) ha ocurrido.

Los errores de tiempo de ejecución son poco comunes en programas simples que verás en los primeros capítulos, así que puede pasar un tiempo antes de que encuentres uno.

Error semántico: El tercer tipo de error es "semántico", lo cual significa que se relaciona con el significado. Si hay un error semántico en tu programa, se ejecutará sin generar mensajes de error, pero no hará lo correcto. Hará otra cosa. Específicamente, hará lo que le dijiste que hiciera.

Identificar errores semánticos puede ser complicado porque requiere que trabajes hacia atrás mirando la salida del programa e intentando averiguar lo que hace.

2.9. Glosario

variable: Un nombre que se refiere a un valor.

asignación: Una sentencia que asigna un valor a una variable.

diagrama de estado: Una representación gráfica de un conjunto de variables y los valores a los cuales se refieren.

palabra clave: Una palabra reservada que se usa como parte de la sintaxis de un programa; no puedes usar palabras claves tales como `if`, `def` y `while` como nombres de variables.

operando: Uno de los valores en los cuales opera un operador.

expresión: Una combinación de variables, operadores y valores que representa un resultado único.

evaluar: Simplificar una expresión realizando las operaciones para obtener un valor único.

sentencia: Una sección de código que representa un comando o acción. Hasta aquí, las sentencias que hemos visto son asignaciones y sentencias print.

ejecutar: Llevar a efecto una sentencia y hacer lo que dice.

modo interactivo: Una manera de usar el intérprete de Python escribiendo código en el prompt.

modo script: Una manera de usar el intérprete de Python para leer código de un script y ejecutarlo.

script: Un programa almacenado en un archivo.

orden de operaciones: Reglas que gobiernan el orden en el cual se evalúan las expresiones que involucran múltiples operadores y operandos.

concatenar: Unir dos operandos de extremo a extremo.

comentario: Información en un programa que está destinada a otros programadores (o cualquiera que lea el código fuente) y no tiene efecto en la ejecución del programa.

error de sintaxis: Un error en un programa que hace imposible reconocer la estructura sintáctica (y por lo tanto imposible de interpretar).

excepción: Un error que es detectado mientras el programa se ejecuta.

semántica: El significado de un programa.

error semántico: Un error en un programa que supone hacer algo distinto a lo que el programador pretendía.

2.10. Ejercicios

Ejercicio 2.1. *Repitiendo mi consejo del capítulo anterior, cuando aprendas una nueva característica, deberías intentar probarla en modo interactivo y cometer errores a propósito para ver qué sale mal.*

- *Hemos visto que* `n = 42` *es legal. ¿Qué hay de* `42 = n`*?*
- *¿Qué ocurre con* `x = y = 1`*?*
- *En algunos lenguajes cada sentencia termina con un punto y coma,* `;`*. ¿Qué ocurre si pones un punto y coma al final de una sentencia de Python?*
- *¿Qué ocurre si pones un punto al final de una sentencia?*
- *En notación matemática puedes multiplicar x e y así: xy. ¿Qué ocurre si intentas eso en Python?*

Ejercicio 2.2. *Practica usando el intérprete de Python como una calculadora:*

1. *El volumen de una esfera con radio r es* $\frac{4}{3}\pi r^3$*. ¿Cuál es el volumen de una esfera con radio 5?*
2. *Supongamos que el precio original de un libro es $24.95, pero las librerías obtienen un 40 % de descuento. El envío cuesta $3 para la primera copia y 75 centavos por cada copia adicional. ¿Cuál es el costo al por mayor para 60 copias?*
3. *Si dejo mi casa a las 6:52 a.m. y corro 1 milla a un ritmo fácil (8 minutos y 15 segundos por milla), luego 3 millas a tempo (7 minutos y 12 segundos por milla) y 1 milla a ritmo fácil de nuevo, ¿a qué hora llego a casa para el desayuno?*

Capítulo 3

Funciones

En el contexto de la programación, una **función** es una secuencia de sentencias que realiza una computación y posee un nombre. Cuando defines una función, especificas el nombre y la secuencia de sentencias. Después, puedes "llamar" a la función por su nombre.

3.1. Llamadas a funciones

Ya hemos visto un ejemplo de una **llamada a función**:

```
>>> type(42)
<class 'int'>
```

El nombre de la función es `type`. La expresión en paréntesis se llama **argumento** de la función. El resultado, para esta función, es el tipo del argumento.

Es común decir que una función "toma" un argumento y "devuelve" un resultado. El resultado también se llama **valor de retorno** (en inglés, *return value*).

Python proporciona funciones que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte en un entero, si puede, o de lo contrario reclama:

```
>>> int('32')
32
>>> int('Hola')
ValueError: invalid literal for int(): Hola
```

`int` puede convertir valores de coma flotante en enteros, pero no redondea; corta la parte de fracción:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` convierte enteros y cadenas en números coma flotante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Por último, `str` convierte su argumento en una cadena:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2. Funciones matemáticas

Python tiene un módulo matemático que proporciona la mayor parte de las funciones matemáticas conocidas. Un **módulo** es un archivo que contiene una colección de funciones relacionadas entre sí.

Antes de que podamos usar las funciones de un módulo, tenemos que importarlo con una **sentencia import**:

```
>>> import math
```

Esta sentencia crea un **objeto de módulo** llamado math. Si muestras el objeto de módulo en pantalla, obtienes información sobre este:

```
>>> math
<module 'math' (built-in)>
```

El objeto de módulo contiene las funciones y variables definidas en el módulo. Para tener acceso a una de las funciones, tienes que especificar el nombre del módulo y el nombre de la función, separados por un punto. Este formato se llama **notación de punto**.

```
>>> relacion = potencia_senal / potencia_ruido
>>> decibeles = 10 * math.log10(relacion)

>>> radianes = 0.7
>>> altura = math.sin(radianes)
```

El primer ejemplo usa `math.log10` para calcular una relación señal/ruido en decibeles (suponiendo que `potencia_senal` y `potencia_ruido` están definidas). El módulo math también proporciona `log`, que calcula logaritmos en base `e`.

El segundo ejemplo encuentra el seno de `radianes`. El nombre de la variable `radianes` es un indicio de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) toman argumentos en radianes. Para convertir de grados a radianes, divide por 180 y multiplica por π:

```
>>> grados = 45
>>> radianes = grados / 180.0 * math.pi
>>> math.sin(radianes)
0.707106781187
```

La expresión `math.pi` obtiene la variable `pi` del módulo math. Su valor es una aproximación en coma flotante de π, con precisión de alrededor de 15 digitos.

Si sabes trigonometría, puedes verificar los resultados anteriores comparándolos con la raíz cuadrada de dos, dividida por dos:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3. Composición

Hasta aquí, hemos visto los elementos de un programa —variables, expresiones y sentencias— de forma aislada, sin hablar sobre cómo combinarlos.

Una de las características más útiles de los lenguajes de programación es su posibilidad de tomar pequeños bloques de construcción y **componerlos**. Por ejemplo, el argumento de una función puede ser cualquier tipo de expresión, incluyendo operadores aritméticos:

```
x = math.sin(grados / 360.0 * 2 * math.pi)
```

También llamadas a funciones:

```
x = math.exp(math.log(x+1))
```

Casi en cualquier lugar que puedes poner un valor, puedes poner una expresión arbitraria, con una excepción: el lado izquierdo de una sentencia de asignación tiene que ser un nombre de variable. Cualquier otra expresión en el lado izquierdo es un error de sintaxis (veremos excepciones a esta regla más tarde).

```
>>> minutos = horas * 60                # correcto
>>> horas * 60 = minutos                # ¡incorrecto!
SyntaxError: can't assign to operator
```

3.4. Agregar nuevas funciones

Hasta aquí, solo hemos estado usando las funciones que vienen con Python, pero también es posible agregar nuevas funciones. Una **definición de función** especifica el nombre de una nueva función y la secuencia de sentencias que se ejecutan cuando la función es llamada.

Aquí hay un ejemplo:

```
def imprimir_letra():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` es una palabra clave que indica que esta es una definición de función. El nombre de la función es `imprimir_letra`. Las reglas para los nombres de funciones son las mismas que para los nombres de variables: las letras, los números y el guión bajo son legales, pero el primer carácter no puede ser un número. No puedes usar una palabra clave como nombre de una función, y deberías evitar tener una variable y una función con el mismo nombre.

Los paréntesis vacíos después del nombre indican que esta función no toma ningún argumento.

La primera línea de la definición de función se llama **encabezado** (en inglés, *header*); el resto se llama **cuerpo** (en inglés, *body*). El encabezado debe terminar con el signo de dos puntos y el cuerpo debe tener sangría. Por convención, la sangría siempre se hace con cuatro espacios. El cuerpo puede contener cualquier número de sentencias.

Las cadenas en las sentencias print están encerradas en comillas dobles. Las comillas simples y las comillas dobles hacen lo mismo; la mayoría de la gente usa comillas simples excepto en casos como este donde una comilla simple (que también es un apóstrofe) aparece en la cadena.

Todas las comillas (simples y dobles) deben ser "comillas rectas", usualmente ubicadas cerca de Enter en el teclado. Las "comillas tipográficas", como las de esta oración, no son legales en Python.

Si escribes una definición de función en modo interactivo, el intérprete imprime puntos (...) que te hacen saber que la definición no está completa:

```
>>> def imprimir_letra():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...
```

Para terminar una función, tienes que insertar una línea vacía.

Al definir una función se crea un **objeto de función**, que tiene tipo `function`:

```
>>> print(imprimir_letra)
<function imprimir_letra at 0xb7e99e9c>
>>> type(imprimir_letra)
<class 'function'>
```

La sintaxis para llamar a la nueva función es la misma que para las funciones incorporadas:

```
>>> imprimir_letra()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Una vez que hayas definido una función, puedes usarla dentro de otra función. Por ejemplo, para repetir el estribillo anterior, podríamos escribir una función llamada `repetir_letra`:

```
def repetir_letra():
    imprimir_letra()
    imprimir_letra()
```

Y luego llamar a `repetir_letra`:

```
>>> repetir_letra()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Pero así no es realmente como sigue la canción.

3.5. Definiciones y usos

Reuniendo los fragmentos de código de la sección anterior, el programa completo se ve así:

```
def imprimir_letra():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repetir_letra():
    imprimir_letra()
    imprimir_letra()

repetir_letra()
```

Este programa contiene dos definiciones de función: `imprimir_letra` y `repetir_letra`. Las definiciones de funciones se ejecutan al igual que otras sentencias, pero el efecto es crear objetos de función. Las sentencias dentro de la función no se ejecutan hasta que la función es llamada, y la definición de función no genera salida.

Como podrías esperar, tienes que crear la función antes de que puedas ejecutarla. En otras palabras, la definición de función tiene que efectuarse antes de que la función sea llamada.

Como ejercicio, mueve la última línea de este programa hasta el principio, así la llamada a la función aparece antes que las definiciones. Ejecuta el programa y mira qué mensaje de error obtienes.

Ahora regresa la llamada de función al final y mueve la definición de `imprimir_letra` a después de la definición de `repetir_letra`. ¿Qué ocurre cuando ejecutas este programa?

3.6. Flujo de ejecución

Para estar seguro de que una función está definida antes de su primer uso, tienes que conocer el orden en que se ejecutan las sentencias, lo cual se llama **flujo de ejecución**.

La ejecución siempre comienza con la primera sentencia del programa. Las sentencias se ejecutan una a la vez, en orden desde arriba hacia abajo.

Las definiciones de función no alteran el flujo de ejecución del programa, pero recuerda que las sentencias dentro de la función no se ejecutan hasta que la función es llamada.

Una llamada a función es como un desvío en el flujo de ejecución. En lugar de ir a la siguiente sentencia, el flujo salta al cuerpo de la función, ejecuta las sentencias que están allí y luego regresa para retomar donde lo había dejado.

Eso suena bastante simple, hasta que recuerdas que una función puede llamar a otra. Mientras está en el medio de una función, el programa quizás tenga que ejecutar las sentencias en otra función. Luego, mientras se ejecuta esa nueva función, ¡el programa quizás tenga que ejecutar otra función más!

Afortunadamente, Python es bueno haciendo seguimiento de dónde está, así que cada vez que se completa una función, el programa retoma donde lo había dejado en la función que la llamó. Cuando llega al final del programa, termina.

En resumen, cuando lees un programa, no siempre quieres leer desde arriba hacia abajo. A veces tiene más sentido si sigues el flujo de ejecución.

3.7. Parámetros y argumentos

Algunas de las funciones que hemos visto requieren argumentos. Por ejemplo, cuando llamas a `math.sin` pasas un número como argumento. Algunas funciones toman más de un argumento; `math.pow` toma dos: la base y el exponente.

Dentro de la función, los argumentos son asignados a variables llamadas **parámetros**. Aquí hay una definición para una función que toma un argumento:

```
def impr_2veces(bruce):
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro con nombre `bruce`. Cuando la función es llamada, esta imprime el valor del parámetro (sea lo que sea) dos veces.

Esta función puede usarse con cualquier valor que se pueda imprimir.

```
>>> impr_2veces('Spam')
Spam
Spam
>>> impr_2veces(42)
42
42
>>> impr_2veces(math.pi)
3.14159265359
3.14159265359
```

Las mismas reglas de composición que se aplican a las funciones incorporadas también se aplican a las funciones definidas por el programador, así que podemos usar cualquier tipo de expresión como un argumento para `impr_2veces`:

```
>>> impr_2veces('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> impr_2veces(math.cos(math.pi))
-1.0
-1.0
```

El argumento es evaluado antes de que se llame a la función, por lo que en los ejemplos las expresiones `'Spam '*4` y `math.cos(math.pi)` son evaluadas una sola vez.

También puedes usar una variable como un argumento:

```
>>> michael = 'Eric, the half a bee.'
>>> impr_2veces(michael)
Eric, the half a bee.
Eric, the half a bee.
```

El nombre de la variable que pasamos como argumento (`michael`) no tiene nada que ver con el nombre del parámetro (`bruce`). No importa cómo se le llame al valor en su casa (en la sentencia llamadora); aquí en `impr_2veces`, a todos les llamamos `bruce`.

3.8. Las variables y los parámetros son locales

Cuando creas una variable dentro de una función, esta es **local**, lo cual significa que existe solamente dentro de la función. Por ejemplo:

```
def cat_2veces(parte1, parte2):
    cat = parte1 + parte2
    impr_2veces(cat)
```

Esta función toma dos argumentos, los concatena e imprime el resultado dos veces. Aquí hay un ejemplo que la usa:

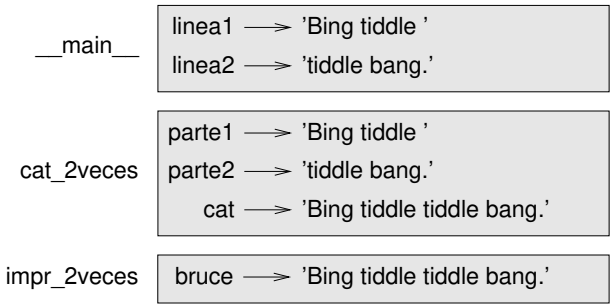

Figura 3.1: Diagrama de pila.

```
>>> linea1 = 'Bing tiddle '
>>> linea2 = 'tiddle bang.'
>>> cat_2veces(linea1, linea2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Cuando `cat_2veces` termina, la variable `cat` se destruye. Si intentamos imprimirla, obtenemos una excepción:

```
>>> print(cat)
NameError: name 'cat' is not defined
```

Los parámetros también son locales. Por ejemplo, afuera de `impr_2veces`, no hay tal cosa como `bruce`.

3.9. Diagramas de pila

Para hacer un seguimiento de qué variables se pueden usar en qué lugar, a veces es útil dibujar un **diagrama de pila** (en inglés, *stack diagram*). Al igual que los diagramas de estado, los diagramas de pila muestran el valor de cada variable, pero también muestran la función a la cual pertenece cada variable.

Cada función se representa por un **marco** (en inglés, *frame*). Un marco es un recuadro que tiene el nombre de una función al lado y los parámetros y variables de la función adentro. El diagrama de pila para el ejemplo anterior se muestra en la Figura 3.1.

Los marcos se organizan en una pila que indica cuál función llama a cuál, y así sucesivamente. En este ejemplo, `impr_2veces` fue llamado por `cat_2veces`, y `cat_2veces` fue llamado por `__main__`, el cual es un nombre especial para el marco más alto. Cuando creas una variable afuera de todas las funciones, pertenece a `__main__`.

Cada parámetro se refiere al mismo valor que su argumento correspondiente. Así que, `parte1` tiene el mismo valor que `linea1`, `parte2` tiene el mismo valor que `linea2` y `bruce` tiene el mismo valor que `cat`.

Si ocurre un error durante una llamada de función, Python imprime el nombre de la función, el nombre de la función que la llamó y el nombre de la función que llamó a *esa*, todo el camino de vuelta a `__main__`.

Por ejemplo, si intentas acceder a `cat` desde adentro de `impr_2veces`, obtienes un `NameError`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_2veces(linea1, linea2)
  File "test.py", line 5, in cat_2veces
    impr_2veces(cat)
  File "test.py", line 9, in impr_2veces
    print(cat)
NameError: name 'cat' is not defined
```

Esta lista de funciones se llama **rastreo** (en inglés, *traceback*). Te dice en qué archivo de programa ocurrió el error, y en qué línea, y qué funciones se estaban ejecutando en ese momento. Además, te muestra la línea de código que causó el error.

El orden de las funciones en el rastreo es el mismo que el orden de los marcos en el diagrama de pila. La función que se está ejecutando actualmente está al final.

3.10. Funciones productivas y funciones nulas

Algunas de las funciones que hemos usado, tales como las funciones matemáticas, devuelven resultados; por falta de un mejor nombre, las llamo **funciones productivas**. Otras funciones, como `impr_2veces`, realizan una acción pero no devuelven un valor. Son llamadas **funciones nulas** (en inglés, *void functions*).

Cuando llamas a una función productiva, casi siempre quieres hacer algo con el resultado; por ejemplo, podrías asignarlo a una variable o usarlo como parte de una expresión:

```
x = math.cos(radianes)
dorado = (math.sqrt(5) + 1) / 2
```

Cuando llamas a una función en modo interactivo, Python muestra el resultado:

```
>>> math.sqrt(5)
2.2360679774997898
```

Pero en un script, si llamas a una función productiva por sí sola, ¡el valor de retorno se pierde para siempre!

```
math.sqrt(5)
```

Este script calcula la raíz cuadrada de 5, pero ya que no almacena ni muestra el resultado, no es muy útil.

Las funciones nulas podrían mostrar algo en la pantalla o tener algún otro efecto, pero no tienen un valor de retorno. Si asignas el resultado a una variable, obtienes un valor especial llamado `None`.

```
>>> resultado = impr_2veces('Bing')
Bing
Bing
>>> print(resultado)
None
```

El valor `None` no es lo mismo que la cadena `'None'`. Es un valor especial que tiene su propio tipo:

```
>>> type(None)
<class 'NoneType'>
```

Las funciones que hemos escrito hasta ahora son todas nulas. Comenzaremos a escribir funciones productivas en unos capítulos más adelante.

3.11. ¿Por qué funciones?

Puede que no esté claro por qué vale la pena el problema de dividir un programa en funciones. Hay muchas razones:

- Crear una nueva función te da la oportunidad de ponerle nombre a un grupo de sentencias, lo cual hace que tu programa sea más fácil de leer y depurar.
- Las funciones pueden hacer que un programa sea más pequeño al eliminar código repetitivo. Después, si quieres hacer un cambio, solo tienes que hacerlo en un lugar.
- Dividir un programa largo en funciones te permite depurar las partes una a la vez y luego reunirlas en un todo funcional.
- Las funciones bien diseñadas son a menudo útiles para muchos programas. Una vez que escribes y depuras una, la puedes reusar.

3.12. Depuración

Una de las habilidades más importantes que adquirirás es la depuración. Aunque puede ser frustrante, la depuración es una de las partes más intelectualmente ricas, desafiantes e interesantes de la programación.

En algunas formas la depuración es como un trabajo de detective. Te enfrentas a pistas y tienes que inferir los procesos y eventos que te guían a los resultados que ves.

La depuración es también como una ciencia experimental. Una vez que tienes una idea sobre qué va mal, modificas tu programa e intentas de nuevo. Si tu hipótesis era correcta, puedes predecir el resultado de la modificación y das un paso más cerca hacia un programa que funcione. Si tu hipótesis era incorrecta, tienes que inventar una nueva. Como señaló Sherlock Holmes, "Una vez descartado lo imposible, lo que queda, por improbable que parezca, debe ser la verdad." (A. Conan Doyle, *El signo de los cuatro*)

Para algunas personas, programar y depurar son lo mismo. Es decir, programar es el proceso de depurar gradualmente un programa hasta que haga lo que tú quieres. La idea es que deberías comenzar con un programa que funcione y hacer pequeñas modificaciones, depurándolas a medida que avanzas.

Por ejemplo, Linux es un sistema operativo que contiene millones de líneas de código, pero comenzó como un programa simple que Linus Torvalds usaba para explorar el chip Intel 80386. Según Larry Greenfield, "Uno de los proyectos anteriores de Linus era un programa que cambiaría entre imprimir AAAA y BBBB. Esto evolucionó más tarde a Linux." (*The Linux Users' Guide* Beta Version 1).

3.13. Glosario

función: Una secuencia de sentencias que tiene nombre y realiza alguna operación útil. Las funciones pueden o no tomar argumentos y pueden o no producir un resultado.

definición de función: Una sentencia que crea una nueva función, especificando su nombre, parámetros y las sentencias que contiene.

objeto de función: Un valor creado por una definición de función. El nombre de la función es una variable que se refiere a un objeto de función.

encabezado: La primera línea de una definición de función.

cuerpo: La secuencia de sentencias dentro de una definición de función.

parámetro: Un nombre usado dentro de una función para referirse al valor pasado como argumento.

llamada a función: Una sentencia que ejecuta una función. Consiste en el nombre de la función seguido de una lista de argumentos en paréntesis.

argumento: Un valor proporcionado a la función cuando la función es llamada. Este valor es asignado al parámetro correspondiente en la función.

variable local: Una variable definida dentro de una función. Una variable local solo puede usarse dentro de su función.

valor de retorno: El resultado de una función. Si una llamada a función se usa como expresión, el valor de retorno es el valor de la expresión.

función productiva: Una función que devuelve un valor.

función nula: Una función que siempre devuelve `None`.

`None`**:** Un valor especial devuelto por funciones nulas.

módulo: Un archivo que contiene una colección de funciones relacionadas entre sí y otras definiciones.

sentencia import: Una sentencia que lee un archivo de módulo y crea un objeto de módulo.

objeto de módulo: Un valor creado por una sentencia `import` que proporciona acceso a los valores definidos en el módulo.

notación de punto: La sintaxis para llamar a una función de otro módulo especificando el nombre del módulo, seguido de un punto y el nombre de la función.

composición: Usar una expresión como parte de una expresión más grande, o una sentencia como parte de una sentencia más grande.

flujo de ejecución: El orden en que las sentencias se ejecutan.

diagrama de pila: Una representación de una pila de funciones, sus variables y los valores a los cuales se refieren.

marco: Un recuadro en un diagrama de pila que representa una llamada a función. Contiene las variables locales y los parámetros de la función.

rastreo: Una lista de las funciones que se están ejecutando, impresas cuando ocurre una excepción.

3.14. Ejercicios

Ejercicio 3.1. *Escribe una función con nombre* `justificar_derecha` *que tome una cadena con nombre* `s` *como parámetro e imprima la cadena con suficientes espacios al inicio de tal manera que la última letra de la cadena esté en la columna 70 de la pantalla.*

```
>>> justificar_derecha('monty')
                                                              monty
```

Pista: usa la repetición de cadenas y la concatenación. Además, Python proporciona una función incorporada llamada `len` *que devuelve la longitud de una cadena, por lo que el valor de* `len('monty')` *es 5.*

Ejercicio 3.2. *Un objeto de función es un valor que puedes asignar a una variable o pasarlo como argumento. Por ejemplo,* `hacer_2veces` *es una función que toma un objeto de función como argumento y lo llama dos veces:*

```
def hacer_2veces(f):
    f()
    f()
```

Aquí hay un ejemplo que usa `hacer_2veces` *para llamar a una función con nombre* `imprimir_spam` *dos veces.*

```
def imprimir_spam():
    print('spam')

hacer_2veces(imprimir_spam)
```

1. *Escribe este ejemplo en un script y pruébalo.*
2. *Modifica* `hacer_2veces` *para que tome dos argumentos, un objeto de función y un valor, y llame a la función dos veces, pasando al valor como argumento.*
3. *Copia la definición de* `impr_2veces`*, presentada previamente en este capítulo, a tu script.*
4. *Usa la versión modificada de* `hacer_2veces` *para llamar a* `impr_2veces` *dos veces, pasando a* `'spam'` *como argumento.*
5. *Define una nueva función llamada* `hacer_4veces` *que tome un objeto de función y un valor y llame a la función cuatro veces, pasando al valor como argumento. Debería haber solo dos sentencias en el cuerpo de esta función, no cuatro.*

Solution: `http://thinkpython2.com/code/do_four.py`.

Ejercicio 3.3. *Nota: Este ejercicio debería hacerse usando solo las sentencias y otras características que hemos aprendido hasta ahora.*

1. *Escribe una función que dibuje una cuadrícula como la siguiente:*

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

Pista: para imprimir más de un valor en una línea, puedes imprimir una secuencia de valores separada por comas:

```
print('+', '-')
```

Por defecto, `print` *avanza a la siguiente línea, pero puedes anular ese comportamiento y poner un espacio al final, como esto:*

```
print('+', end=' ')
print('-')
```

La salida de estas sentencias es `'+ -'` *en la misma línea. La salida desde la siguiente sentencia print debería comenzar en la siguiente línea.*

2. *Escribe una función que dibuje una cuadrícula similar con cuatro filas y cuatro columnas.*

Solución: `http://thinkpython2.com/code/grid.py`*. Crédito: este ejercicio está basado en un ejercicio de Oualline,* Practical C Programming, Third Edition*, O'Reilly Media, 1997.*

Capítulo 4

Estudio de caso: diseño de interfaz

Este capítulo presenta un estudio de caso que demuestra un proceso para diseñar funciones que interactúen entre sí.

Se presenta el módulo `turtle`, el cual te permite crear imágenes usando gráficas tortuga. El módulo `turtle` está incluido en la mayoría de las instalaciones de Python, pero si estás ejecutando Python usando PythonAnywhere, no podrás ejecutar los ejemplos de tortuga (al menos no podías cuando escribí esto).

Si ya has instalado Python en tu computador, deberías poder ejecutar los ejemplos. Si no, ahora es un buen momento para instalarlo. He publicado instrucciones en `http://tinyurl.com/thinkpython2e`.

Los códigos de ejemplo de este capítulo están disponibles en `http://thinkpython2.com/code/polygon.py`.

4.1. El módulo turtle

Para verificar si tienes el módulo `turtle`, abre el intérprete de Python y escribe

```
>>> import turtle
>>> bob = turtle.Turtle()
```

Cuando ejecutes este código, debería crearse una nueva ventana con una flecha pequeña que representa la tortuga. Cierra la ventana.

Crea un archivo con nombre `mipoligono.py` y escribe en él las siguientes líneas de código:

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

El módulo `turtle` (con 't' minúscula) proporciona una función llamada `Turtle` (con 'T' mayúscula) que crea un objeto Turtle, el cual asignamos a una variable con nombre `bob`. Al imprimir `bob` se muestra algo como:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

Esto significa que `bob` se refiere a un objeto con tipo `Turtle` como se define en el módulo `turtle`.

`mainloop` le dice a la ventana que espere a que el usuario haga algo, aunque en este caso no hay mucho que pueda hacer el usuario excepto cerrar la ventana.

Una vez que creas una tortuga, puedes llamar a un **método** para moverla alrededor de la ventana. Un método es similar a una función, pero usa una sintaxis un poco diferente. Por ejemplo, para mover la tortuga adelante:

```
bob.fd(100)
```

El método, `fd` (*forward*), está asociado con el objeto tortuga que llamamos `bob`. Llamar a un método es como hacer una solicitud: le estás pidiendo a `bob` que se mueva hacia adelante.

El argumento de `fd` es una distancia en pixeles, por lo que el tamaño real depende de tu pantalla.

Otros métodos que puedes llamar en un objeto Turtle son `bk` (*backward*) para retroceder, `lt` (*left turn*) para girar a la izquierda y `rt` (*right turn*) para girar a la derecha. El argumento para `lt` y `rt` es un ángulo en grados.

Además, cada Turtle sostiene una pluma, que está arriba o abajo; si la pluma está abajo, la tortuga deja un rastro cuando se mueve. Los métodos `pu` y `pd` representan "*pen up*" y "*pen down*".

Para dibujar un ángulo recto, agrega estas líneas al programa (después de crear a `bob` y antes de llamar a `mainloop`):

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

Cuando ejecutes este programa, deberías ver a `bob` moverse al este y luego al norte, dejando dos segmentos de línea atrás.

Ahora modifica el programa para dibujar un cuadrado. ¡No continues hasta que lo hayas hecho funcionar!

4.2. Repetición simple

Es probable que hayas escrito algo así:

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

Podemos hacer lo mismo de manera más concisa con una sentencia `for`. Agrega este ejemplo a `mipoligono.py` y ejecútalo de nuevo:

```
for i in range(4):
    print('¡Hola!')
```

Deberías ver algo como esto:

```
¡Hola!
¡Hola!
¡Hola!
¡Hola!
```

Este es el uso más simple de una sentencia `for`; después veremos más. Pero eso debería ser suficiente para dejarte reescribir tu programa que dibuja cruadrados. No continues hasta que lo hagas.

Aquí hay una sentencia `for` que dibuja un cuadrado:

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

La sintaxis de una sentencia `for` es similar a una definición de función. Tiene un encabezado que termina con el signo dos puntos y un cuerpo con sangrías. El cuerpo puede contener cualquier número de sentencias.

Una sentencia `for` también es llamada **bucle** porque el flujo de ejecución pasa por el cuerpo y luego vuelve hacia arriba. Es este caso, pasa por el cuerpo cuatro veces.

Esta versión es en realidad un poco diferente del código que dibuja cuadrados propuesto anteriormente porque hace otro giro después de dibujar el último lado del cuadrado. El giro extra toma más tiempo, pero simplifica el código si hacemos lo mismo en cada paso por el bucle. Esta versión también tiene el efecto de regresar a la tortuga a su posición inicial, apuntando a la dirección inicial.

4.3. Ejercicios

Lo siguiente es una serie de ejercicios que usan el módulo `turtle`. Pretenden ser divertidos, pero también tienen un punto. Mientras trabajes en ellos, piensa cuál es el punto.

Las siguientes secciones tienen soluciones a los ejercicios, así que no las mires hasta que hayas terminado (o al menos intentado).

1. Escribe una función llamada `cuadrado` que tome un parámetro con nombre `t`, que es una tortuga. Debería usar la tortuga para dibujar un cuadrado.

   Escribe una llamada a función que pase a `bob` como argumento de `cuadrado`, y luego ejecuta el programa de nuevo.

2. Agrega otro parámetro, con nombre `longitud`, a `cuadrado`. Modifica el cuerpo para que la longitud de los lados sea `longitud`, y luego modifica la llamada a función para poner un segundo argumento. Ejecuta el programa de nuevo. Prueba tu programa con un rango de valores para `longitud`.

3. Haz una copia de `cuadrado` y cambia el nombre a `poligono`. Agrega otro parámetro con nombre `n` y modifica el cuerpo para que dibuje un polígono regular con n lados. Pista: los ángulos exteriores de un polígono regular con n lados son de $360/n$ grados.

4. Escribe una función llamada `circulo` que tome una tortuga, `t`, y radio, `r`, como parámetros y dibuje un círculo aproximado llamando a `poligono` con una longitud y número de lados apropiado. Prueba tu función con un rango de valores de `r`.

   Pista: averigua cuál es el perímetro del círculo y asegúrate de que se cumpla que `longitud * n = perimetro`.

5. Haz una versión más general de `circulo` llamada `arco` que tome un parámetro adicional `angulo`, que determine qué fracción de un círculo dibujar. `angulo` está en grados, así que cuando `angulo=360`, `arco` debería dibujar un círculo completo.

4.4. Encapsulamiento

El primer ejercicio te pide poner tu código que dibuja cuadrados dentro de una definición de función y luego llamar a la función, pasando a la tortuga como parámetro. Aquí está la solución:

```
def cuadrado(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

cuadrado(bob)
```

Las sentencias de más adentro, `fd` y `lt`, tienen doble sangría para mostrar que están dentro del bucle `for`, el cual está dentro de la definición de función. La siguiente línea, `cuadrado(bob)`, está alineada con el margen izquierdo, lo cual indica el término tanto del bucle `for` como de la definición de función.

Dentro de la función, `t` se refiere a la misma tortuga `bob`, por lo que `t.lt(90)` tiene el mismo efecto que `bob.lt(90)`. En ese caso, ¿por qué no llamar al parámetro `bob`? La idea es que `t` puede ser cualquier tortuga, no solo `bob`, así que podrías crear una segunda tortuga y pasarla como argumento a `cuadrado`:

```
alice = turtle.Turtle()
cuadrado(alice)
```

El acto de envolver un pedazo de código en una función se llama **encapsulamiento**. Uno de los beneficios del encapsulamiento es que adjunta un nombre al código, lo cual sirve como una especie de documentación. Otra ventaja es que si reutilizas el código, ¡es más conciso llamar a una función dos veces que copiar y pegar el cuerpo!

4.5. Generalización

El siguiente paso es agregar un parámetro `longitud` a `cuadrado`. Aquí hay una solución:

```
def cuadrado(t, longitud):
    for i in range(4):
        t.fd(longitud)
        t.lt(90)

cuadrado(bob, 100)
```

El acto de agregar un parámetro a una función se llama **generalización** porque hace que la función sea más general: en la versión anterior, el cuadrado tiene siempre el mismo tamaño; en esta versión puede ser cualquier tamaño.

El siguiente paso es también una generalización. En lugar de dibujar cuadrados, `poligono` dibuja polígonos regulares con cualquier número de lados. Aquí hay una solución:

```
def poligono(t, n, longitud):
    angulo = 360 / n
    for i in range(n):
        t.fd(longitud)
        t.lt(angulo)

poligono(bob, 7, 70)
```

Este ejemplo dibuja un polígono de 7 lados de longitud 70.

Si estás usando Python 2, el valor de `angulo` podría ser incorrecto debido a una división entera. Una solución simple es calcular `angulo = 360.0 / n`. Dado que el numerador es un número de coma flotante, el resultado es de coma flotante.

Cuando una función tiene más que unos pocos argumentos numéricos, es fácil olvidar qué son, o en qué orden deberían estar. En ese caso a menudo es una buena idea incluir los nombres de los parámetros en la lista de argumentos:

```
poligono(bob, n=7, longitud=70)
```

Estos se llaman **argumentos de palabra clave** porque incluyen a los nombres de parámetro tratándolos como "palabras clave" (no confundir con las palabras clave de Python como `while` y `def`).

Esta sintaxis hace que el programa sea más legible. Es también un recordatorio sobre cómo funcionan los argumentos y los parámetros: cuando llamas a una función, los argumentos son asignados a los parámetros.

4.6. Diseño de interfaz

El siguiente paso es escribir `circulo`, el cual toma un radio, `r`, como parámetro. Aquí hay una solución simple que usa a `poligono` para dibujar un polígono de 50 lados:

```
import math

def circulo(t, r):
    perimetro = 2 * math.pi * r
    n = 50
    longitud = perimetro / n
    poligono(t, n, longitud)
```

La primera linea calcula el perímetro de un círculo con radio `r` usando la fórmula $2\pi r$. Dado que usamos `math.pi`, tenemos que importar `math`. Por convención, las sentencias `import` usualmente están al comienzo del script.

`n` es el número de segmentos de línea en tu aproximación de un círculo, por lo que `longitud` es la longitud de cada segmento. Así, `poligono` dibuja un polígono de 50 lados que aproxima un círculo con radio `r`.

Una limitación de esta solución es que n es una constante, lo cual significa que para círculos muy grandes, los segmentos de línea son muy largos, y para círculos pequeños, ocupamos mucho tiempo dibujando segmentos muy pequeños. Una solución sería generalizar la función para que tome a n como parámetro. Esto le daría al usuario (quien llame a circulo) más control, pero la interfaz sería menos limpia.

La **interfaz** de una función es un resumen de cómo se usa: ¿cuáles son los parámetros? ¿Qué hace la función? ¿Y cuál es el valor de retorno? Una interfaz es "limpia" si permite a la sentencia llamadora hacer lo que quiere sin lidiar con detalles innecesarios.

En este ejemplo, r forma parte de la interfaz porque especifica el círculo a dibujar. n es menos apropiado porque pertenece a los detalles de *cómo* debería dibujarse el círculo.

En lugar de desordenar la interfaz, es mejor escoger un valor apropiado de n que dependa del perímetro:

```
def circulo(t, r):
    perimetro = 2 * math.pi * r
    n = int(perimetro / 3) + 3
    longitud = perimetro / n
    poligono(t, n, longitud)
```

Ahora, el número de segmentos es un entero cercano a perimetro/3, por lo que la longitud de cada segmento es aproximadamente 3, lo cual es suficientemente pequeño para que el círculo se vea bien, pero lo suficientemente grande para ser eficiente, y aceptable para cualquier tamaño de círculo.

Sumar 3 a n garantiza que el polígono tenga al menos 3 lados.

4.7. Refactorización

Cuando escribí circulo, fui capaz de reutilizar poligono porque un polígono de muchos lados es una buena aproximación de un círculo. Pero arco no es tan cooperativo; no podemos usar poligono o circulo para dibujar un arco.

Una alternativa es comenzar con una copia de poligono y transformarla en arco. El resultado podría verse así:

```
def arco(t, r, angulo):
    longitud_arco = 2 * math.pi * r * angulo / 360
    n = int(longitud_arco / 3) + 1
    longitud_paso = longitud_arco / n
    angulo_paso = angulo / n

    for i in range(n):
        t.fd(longitud_paso)
        t.lt(angulo_paso)
```

La segunda mitad de esta función se parece a poligono, pero no podemos reutilizar poligono sin cambiar la interfaz. Podríamos generalizar poligono para que tome un ángulo como tercer argumento, ¡pero entonces poligono ya no sería un nombre apropiado! En cambio, llamemos polilinea a la función más general:

```
def polilinea(t, n, longitud, angulo):
    for i in range(n):
        t.fd(longitud)
        t.lt(angulo)
```

Ahora podemos reescribir `poligono` y `arco` para que use a `polilinea`:

```
def poligono(t, n, longitud):
    angulo = 360.0 / n
    polilinea(t, n, longitud, angulo)

def arco(t, r, angulo):
    longitud_arco = 2 * math.pi * r * angulo / 360
    n = int(longitud_arco / 3) + 1
    longitud_paso = longitud_arco / n
    angulo_paso = float(angulo) / n
    polilinea(t, n, longitud_paso, angulo_paso)
```

Finalmente, podemos reescribir `circulo` para que use a `arco`:

```
def circulo(t, r):
    arco(t, r, 360)
```

Este proceso —reorganizar un programa para mejorar las interfaces y facilitar la reutilización de código— se llama **refactorización**. En este caso, notamos que había código similar en `arco` y `poligono`, así que "lo factorizamos" en `polilinea`.

Si hubiéramos planificado con anticipación, podríamos haber escrito `polilinea` primero y evitar la refactorización, pero a menudo no sabes lo suficiente al comienzo de un proyecto para diseñar todas las interfaces. Una vez que comienzas a escribir código, entiendes mejor el problema. A veces la refactorización es una señal de que has aprendido algo.

4.8. Un plan de desarrollo

Un **plan de desarrollo** es un proceso para escribir programas. El proceso que usamos en este estudio de caso es "encapsulamiento y generalización". Los pasos de este proceso son:

1. Comenzar escribiendo un programa pequeño sin definiciones de función.
2. Una vez que funciona el programa, identifica una parte coherente, encapsula la parte en una función y dale un nombre.
3. Generaliza la función agregando parámetros apropiados.
4. Repite los pasos 1–3 hasta que tengas un conjunto de funciones eficaces. Copia y pega código que funcione para evitar repetir (y volver a depurar).
5. Busca oportunidades para mejorar el programa refactorizando. Por ejemplo, si tienes código similar en muchos lugares, considera factorizarlo dentro de una función general apropiada.

Este proceso tiene algunos inconvenientes —más tarde veremos alternativas— pero puede ser útil si no sabes de antemano cómo dividir el programa en funciones. Este enfoque te permite diseñar a medida que avanzas.

4.9. docstring

Un **docstring** es una cadena al comienzo de una función que explica la interfaz ("doc" es la abreviatura de "documentation"). Aquí hay un ejemplo:

```
def polilinea(t, n, longitud, angulo):
    """Dibuja n segmentos de línea con la longitud dada
    y el ángulo (en grados) entre ellos.  t es una tortuga.
    """
    for i in range(n):
        t.fd(longitud)
        t.lt(angulo)
```

Por convención, todos los docstrings son cadenas entre triple comillas, también conocidas como cadenas multilínea porque las triple comillas permiten expandir la cadena a más de una línea.

Es breve, pero contiene la información escencial que alguien necesitaría para usar esta función. Explica de manera concisa lo que hace la función (sin entrar en detalles sobre cómo lo hace). Explica qué efecto tiene cada parámetro en el comportamiento de la función y de qué tipo debería ser cada parámetro (si no es obvio).

Escribir este tipo de documentación es una parte importante del diseño de la interfaz. Una interfaz bien diseñada debería ser simple de explicar; si tienes dificultades al explicar una de tus funciones, quizás la interfaz podría mejorar.

4.10. Depuración

Una interfaz es como un contrato entre una función y la sentencia llamadora. La llamadora acepta proporcionar ciertos argumentos y la función acepta hacer cierto trabajo.

Por ejemplo, `polilinea` requiere cuatro argumentos: `t` tiene que ser Turtle; `n` tiene que ser un entero; `longitud` debería ser un número positivo; y `angulo` tiene que ser un número, que se entiende que está en grados.

Estos requisitos se llaman **precondiciones** porque se supone que son verdaderos antes de que la función comience a ejecutarse. De forma opuesta, las condiciones al final de la función son **postcondiciones**. Las postcondiciones incluyen el efecto previsto de la función (como al dibujar segmentos de línea) y cualquier efecto secundario (como mover la tortuga o hacer otros cambios).

Las precondiciones son responsabilidad de la llamadora. Si la llamadora viola una precondición (¡debidamente documentada!) y la función no funciona de forma correcta, el error está en la llamadora, no en la función.

Si las precondiciones se satisfacen y las postcondiciones no, el error está en la función. Si tus pre y post condiciones están claras, pueden ayudar con la depuración.

4.11. Glosario

método: Una función que se asocia a un objeto y se llama usando notación de punto.

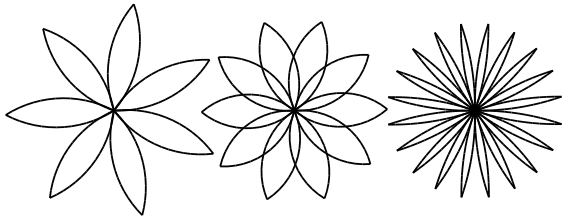

Figura 4.1: Flores tortuga.

bucle: Una parte de un programa que puede ejecutarse de forma repetida.

encapsulamiento: El proceso de transformar una secuencia de sentencias en una definición de función.

generalización: El proceso de reemplazar algo innecesariamente específico (como un número) con algo apropiadamente general (como una variable o parámetro).

argumento de palabra clave: Un argumento que incluye el nombre del parámetro tratándolo como "palabra clave".

interfaz: Una descripción de cómo usar una función, incluyendo el nombre y descripciones de los argumentos y el valor de retorno.

refactorización: El proceso de modificar un programa que funciona para mejorar las interfaces de funciones y otras cualidades del código.

plan de desarrollo: Un proceso para escribir programas.

docstring: Una cadena que aparece en la parte superior de una definición de función para documentar la interfaz de la función.

precondición: Un requisito que debería satisfacer la sentencia llamadora antes de que la función comience.

postcondición: Un requisito que debería satisfacer la función antes de que termine.

4.12. Ejercicios

Ejercicio 4.1. *Descarga el código de este capítulo en* `http://thinkpython2.com/code/polygon.py`.

1. *Dibuja un diagrama de pila que muestre el estado del programa al ejecutar* `circulo(bob, radio)`. *Puedes hacer la aritmética a mano o agregar sentencias* `print` *al código.*

2. *La versión de* `arco` *en la Sección 4.7 no es muy precisa debido a que la aproximación lineal del círculo está siempre afuera del verdadero círculo. Como resultado, la tortuga termina a unos pocos pixeles de distancia del destino correcto. Mi solución muestra una manera de reducir el efecto de este error. Lee el código y ve si tiene sentido para ti. Si dibujas un diagrama, podrías ver cómo trabaja.*

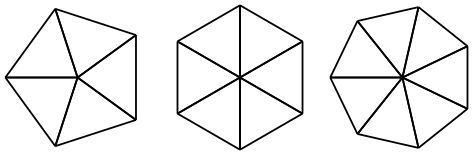

Figura 4.2: Pasteles tortuga.

Ejercicio 4.2. *Escribe un conjunto de funciones apropiadamente generales que puedan dibujar flores como en la Figura 4.1.*

Solución: `http://thinkpython2.com/code/flower.py`, *también requiere* `http://thinkpython2.com/code/polygon.py`.

Ejercicio 4.3. *Escribe un conjunto de funciones apropiadamente generales que puedan dibujar formas como en la Figura 4.2.*

Solución: `http://thinkpython2.com/code/pie.py`.

Ejercicio 4.4. *Las letras del alfabeto se pueden construir desde un número moderado de elementos básicos, como líneas verticales y horizontales y unas pocas curvas. Diseña un alfabeto que pueda dibujarse con un número mínimo de elementos básicos y luego escribe funciones que dibujen las letras.*

Deberías escribir una función para cada letra, con nombres `dibujar_a`, `dibujar_b`, *etc., y poner tus funciones en un archivo con nombre* `letras.py`. *Puedes descargar una "máquina de escribir tortuga" desde* `http://thinkpython2.com/code/typewriter.py` *para ayudarte a probar tu código.*

Puedes obtener una solución en `http://thinkpython2.com/code/letters.py`; *también requiere* `http://thinkpython2.com/code/polygon.py`.

Ejercicio 4.5. *Lee sobre espirales en* `http://es.wikipedia.org/wiki/Espiral`; *luego escribe un programa que dibuje una espiral arquimediana (o uno de los otros tipos). Solución:* `http://thinkpython2.com/code/spiral.py`.

Capítulo 5

Condicionales y recursividad

El tema principal de este capítulo es la sentencia `if`, la cual ejecuta código diferente dependiendo del estado del programa. Pero primero quiero presentar dos operadores nuevos: división entera y módulo.

5.1. División entera y módulo

El operador **división entera**, `//`, divide dos números y redondea a un entero hacia abajo. Por ejemplo, supongamos que la duración de una película es 105 minutos. Quizás quieres saber cuánto dura en horas. La división convencional devuelve un número de coma flotante:

```
>>> minutos = 105
>>> minutos / 60
1.75
```

Pero normalmente no escribimos las horas con decimales. La división entera devuelve el número entero de horas, redondeando:

```
>>> minutos = 105
>>> horas = minutos // 60
>>> horas
1
```

Para obtener el resto, podrías restar una hora en minutos:

```
>>> resto = minutos - horas * 60
>>> resto
45
```

Una alternativa es usar el **operador de módulo**, `%`, el cual divide dos números y devuelve el resto.

```
>>> resto = minutos % 60
>>> resto
45
```

El operador de módulo es más útil de lo que parece. Por ejemplo, puedes verificar si un número es divisible por otro: si `x%y` es cero, entonces `x` es divisible por `y`.

Además, puedes extraer el dígito de más a la derecha o más dígitos de un número. Por ejemplo, `x%10` entrega el dígito de más a la derecha de `x` (en base 10). De manera similar, `x%100` entrega los dos últimos dígitos.

Si estás usando Python 2, la división funciona diferente. El operador división, `/`, realiza una división entera si ambos operandos son enteros, y la división de coma flotante si cualquiera de los dos operandos es un `float`.

5.2. Expresión booleana

Una **expresión booleana** es una expresión que es verdadera o falsa. Los siguientes ejemplos usan el operador `==`, el cual compara dos operandos y produce `True` si son iguales y `False` si no lo son:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` y `False` son valores especiales que pertenecen al tipo `bool`; no son cadenas:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

El operador `==` es uno de los **operadores relacionales**; los otros son:

```
x != y               # x no es igual a y
x > y                # x es mayor que y
x < y                # x es menor que y
x >= y               # x es mayor o igual que y
x <= y               # x es menor o igual que y
```

Aunque estas operaciones probablemente sean familiares para ti, los símbolos de Python son diferentes a los símbolos matemáticos. Un error común es usar el signo igual simple (`=`) en lugar de un signo igual doble (`==`). Recuerda que `=` es un operador de asignación y `==` es un operador relacional. No hay tal cosa como `=<` o `=>`.

5.3. Operadores lógicos

Existen tres **operadores lógicos**: `and`, `or` y `not`. La semántica (significado) de estos operadores es similar a su significado en inglés. Por ejemplo, `x > 0 and x < 10` es verdadera solo si `x` es mayor que 0 *y* menor que 10.

`n%2 == 0 or n%3 == 0` es verdadera si *cualquiera o ambas* condiciones son verdaderas, es decir, si el número es divisible por 2 *o* 3.

Finalmente, el operador `not` niega una expresión booleana, por lo que `not (x > y)` es verdadera si `x > y` es falsa, es decir, si `x` es menor o igual que `y`.

Estrictamente hablando, los operandos de los operadores lógicos deberían ser expresiones booleanas, pero Python no es muy estricto. Cualquier número distinto de cero es interpretado como `True`:

```
>>> 42 and True
True
```

Esta flexibilidad puede ser útil, pero hay algunas sutilezas que podrían ser confusas. Quizás quieras evitar esto (a menos que sepas lo que estás haciendo).

5.4. Ejecución condicional

Para escribir programas útiles, casi siempre necesitamos la capacidad de verificar las condiciones y cambiar el comportamiento del programa como corresponde. Las **sentencias condicionales** nos dan esta capacidad. La forma más simple es la sentencia `if`:

```
if x > 0:
    print('x es positivo')
```

La expresión booleana después de `if` se llama **condición**. Si es verdadera, se ejecutan las sentencias con sangría. Si no, no pasa nada.

Las sentencias `if` tienen la misma estructura que las definiciones de función: un encabezado seguido de un cuerpo con sangrías. Las sentencias como esta se llaman **sentencias compuestas**.

No hay límite en el número de sentencias que pueden aparecer en el cuerpo, pero tiene que haber al menos una. A veces, es útil tener un cuerpo sin sentencias (usualmente para reservar lugar a código que no has escrito todavía). En ese caso, puedes usar la sentencia `pass`, la cual no hace nada.

```
if x < 0:
    pass          # PENDIENTE: ¡falta manejar los valores negativos!
```

5.5. Ejecución alternativa

Una segunda forma de la sentencia `if` es la "ejecución alternativa", en la cual hay dos posibilidades y la condición determina cuál se ejecuta. La sintaxis se ve así:

```
if x % 2 == 0:
    print('x es par')
else:
    print('x es impar')
```

Si el resto de dividir `x` por 2 es 0, entonces sabemos que `x` es par y el programa muestra un mensaje correspondiente. Si la condición es falsa, se ejecuta el segundo conjunto de sentencias. Dado que la condición debe ser verdadera o falsa, se ejecutará exactamente una de las alternativas. Las alternativas se llaman **ramas**, porque son ramas en el flujo de ejecución.

5.6. Condicionales encadenados

A veces hay más de dos posibilidades y necesitamos más de dos ramas. Una manera de expresar una computación como esa es un **condicional encadenado**:

```
if x < y:
    print('x es menor que y')
elif x > y:
    print('x es mayor que y')
else:
    print('x e y son iguales')
```

`elif` es una abreviación de "else if". De nuevo, se ejecutará exactamente una rama. No hay límite en el número de sentencias `elif`. Si hay una cláusula `else`, tiene que estar al final, pero no tiene que haber una.

```
if opcion == 'a':
    dibujar_a()
elif opcion == 'b':
    dibujar_b()
elif opcion == 'c':
    dibujar_c()
```

Cada condición es verificada en orden. Si la primera es falsa, se verifica la siguiente, y así sucesivamente. Si una de ellas es verdadera, se ejecuta la rama correspondiente y la sentencia termina. Incluso si más de una condición es verdadera, solo se ejecuta la primera rama verdadera.

5.7. Condicionales anidados

Un condicional puede también estar anidado dentro de otro. Podríamos haber escrito el ejemplo de la sección anterior de esta forma:

```
if x == y:
    print('x e y son iguales')
else:
    if x < y:
        print('x es menor que y')
    else:
        print('x es mayor que y')
```

El condicional de más afuera contiene dos ramas. La primera rama contiene una sentencia simple. La segunda rama contiene otra sentencia `if`, la cual tiene dos ramas propias. Aquellas dos ramas son sentencias simples, aunque también podrían haber sido sentencias condicionales.

A pesar de que la sangría de las sentencias hacen evidente la estructura, los **condicionales anidados** se vuelven difíciles de leer rápidamente. Es una buena idea evitarlos cuando puedas.

Los operadores lógicos a menudo proporcionan una manera de simplificar las sentencias condicionales anidadas. Por ejemplo, podemos reescribir el siguiente código usando un único condicional:

```
if 0 < x:
    if x < 10:
        print('x es un número positivo de un dígito.')
```

La sentencia `print` solo se ejecuta si pasamos por los dos condicionales, así que podemos obtener el mismo efecto con el operador `and`:

```
if 0 < x and x < 10:
    print('x es un número positivo de un dígito.')
```

Para este tipo de condición, Python proporciona una opción más concisa:

```
if 0 < x < 10:
    print('x es un número positivo de un dígito.')
```

5.8. Recursividad

Es legal para una función llamar a otra; es legal también para una función llamarse a sí misma. Puede que no sea obvio por qué eso es una buena idea, pero resulta ser una de las cosas más mágicas que puede hacer un programa. Por ejemplo, mira la siguiente función:

```
def cuenta_reg(n):
    if n <= 0:
        print('¡Despegue!')
    else:
        print(n)
        cuenta_reg(n-1)
```

Si `n` es 0 o negativo, muestra la palabra, "¡Despegue!" De lo contrario, muestra a `n` y luego llama a la función con nombre `cuenta_reg` —a sí misma— pasando a `n-1` como argumento.

¿Qué ocurre si llamamos a esta función así?

```
>>> cuenta_reg(3)
```

La ejecución de `cuenta_reg` comienza con `n=3`, y dado que `n` es mayor que 0, muestra el valor 3 y se llama a sí misma...

> La ejecución de `cuenta_reg` comienza con `n=2`, y dado que `n` es mayor que 0, muestra el valor 2 y se llama a sí misma...
>
> > La ejecución de `cuenta_reg` comienza con `n=1`, y dado que `n` es mayor que 0, muestra el valor 1 y se llama a sí misma...
> >
> > > La ejecución de `cuenta_reg` comienza con `n=0`, y dado que `n` no es mayor que 0, muestra la palabra "¡Despegue!" y luego vuelve.
> >
> > La `cuenta_reg` que obtuvo `n=1` vuelve.
>
> La `cuenta_reg` que obtuvo `n=2` vuelve.

La `cuenta_reg` que obtuvo `n=3` vuelve.

Y luego estás de regreso en `__main__`. Por lo tanto, la salida completa se ve así:

```
3
2
1
¡Despegue!
```

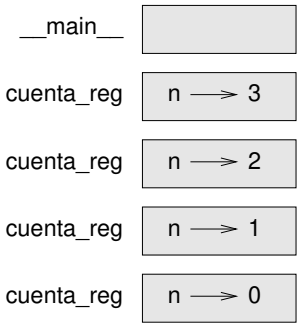

Figura 5.1: Diagrama de pila.

Una función que se llama a sí misma es **recursiva**; el proceso de ejecutarla se llama **recursividad**.

Como ejemplo adicional, podemos escribir una función que imprima una cadena `n` veces.

```
def imprimir_n(s, n):
    if n <= 0:
        return
    print(s)
    imprimir_n(s, n-1)
```

Si `n <= 0` la **sentencia return** hace que se salga de la función. El flujo de ejecución vuelve inmediatamente a la sentencia llamadora y las líneas restantes de la función no se ejecutan.

El resto de la función es similar a `cuenta_reg`: muestra a `s` y luego se llama a sí misma para mostrar a `s` otras $n - 1$ veces. Entonces, el número de líneas de salida es `1 + (n - 1)`, lo cual suma `n`.

Para ejemplos simples como este, probablemente es más fácil usar un bucle `for`. Pero más adelante veremos ejemplos que son difíciles de escribir con un bucle `for` y fáciles de escribir con recursividad, así que es bueno comenzar pronto.

5.9. Diagramas de pila para funciones recursivas

En la Sección 3.9, usamos un diagrama de pila para representar el estado de un programa durante una llamada a función. El mismo tipo de diagrama puede ayudar a interpretar una función recursiva.

Cada vez que una función es llamada, Python crea un marco que contiene las variables locales y los parámetros de la función. Para una función recursiva, podría haber más de un marco en la pila al mismo tiempo.

La Figura 5.1 muestra un diagrama de pila para `cuenta_reg` llamada con `n = 3`.

Como siempre, la parte de arriba de la pila es el marco para `__main__`. Está vacío porque no creamos variables en `__main__` ni le pasamos argumentos.

Los cuatro marcos de `cuenta_reg` tienen valores diferentes para el parámetro `n`. La parte de abajo de la pila, donde `n=0`, se llama **caso base**. No hace una llamada recursiva, por lo que no hay más marcos.

Como ejercicio, dibuja un diagrama de pila para `imprimir_n` llamada con `s = 'Hola'` y `n=2`. Luego escribe una función llamada `hacer_n` que tome un objeto de función y un número, `n`, como argumentos, y que llame a dicha función `n` veces.

5.10. Recursividad infinita

Si una recursividad nunca llega a un caso base, sigue haciendo llamadas recursivas por siempre y el programa nunca termina. Esto se conoce como **recursividad infinita** y en general no es una buena idea. Aquí hay un programa mínimo con una recursividad infinita:

```
def recursivo():
    recursivo()
```

En la mayoría de los entornos de programación, un programa con recursividad infinita no se ejecuta realmente por siempre. Python entrega un mensaje de error cuando la recursividad alcanza la profundidad máxima:

```
  File "<stdin>", line 2, in recursivo
  File "<stdin>", line 2, in recursivo
  File "<stdin>", line 2, in recursivo
                  .
                  .
                  .
  File "<stdin>", line 2, in recursivo
RuntimeError: Maximum recursion depth exceeded
```

Este rastreo es un poco más grande del que vimos en el capítulo anterior. Cuando el error ocurre, ¡hay 1000 marcos de `recursivo` en la pila!

Si encuentras una recursividad infinita por accidente, revisa tu función para confirmar que hay un caso base que no hace una llamada recursiva. Y si hay un caso base, verifica si tienes garantizado alcanzarlo.

5.11. Entrada de teclado

Los programas que hemos escrito hasta ahora no admiten entradas del usuario. Simplemente hacen siempre lo mismo.

Python proporciona una función incorporada llamada `input` que detiene el programa y espera a que el usuario escriba algo. Cuando el usuario presiona Return o Enter, el programa continúa e `input` devuelve lo que el usuario escribió como una cadena. En Python 2, la misma función se llama `raw_input`.

```
>>> texto = input()
¿Qué estás esperando?
>>> texto
'¿Qué estás esperando?'
```

Antes de obtener la entrada del usuario, es una buena idea imprimir un mensaje que le diga al usuario qué escribir. `input` puede tomar un mensaje como argumento:

```
>>> nombre = input('¿Cuál...es tu nombre?\n')
¿Cuál...es tu nombre?
¡Arturo, Rey de los Britones!
>>> nombre
'¡Arturo, Rey de los Britones!'
```

La secuencia `\n` al final del mensaje representa una **nueva línea**, la cual es un carácter especial que provoca un salto de línea. Esa es la razón por la cual la entrada del usuario aparece debajo del mensaje.

Si esperas que el usuario escriba un entero, puedes intentar convertir el valor de retorno a `int`:

```
>>> mensaje = '¿Cuál...es la velocidad media de una golondrina sin carga?\n'
>>> velocidad = input(mensaje)
¿Cuál...es la velocidad media de una golondrina sin carga?
42
>>> int(velocidad)
42
```

Pero si el usuario escribe algo distinto a una cadena de dígitos, obtienes un error:

```
>>> velocidad = input(mensaje)
¿Cuál...es la velocidad media de una golondrina sin carga?
¿De qué especie, de la africana o de la europea?
>>> int(velocidad)
ValueError: invalid literal for int() with base 10
```

Más adelante veremos cómo tratar este tipo de error.

5.12. Depuración

Cuando ocurre un error de sintaxis o de tiempo de ejecución, el mensaje de error contiene mucha información, pero esto puede ser abrumador. Las partes más útiles suelen ser:

- Qué tipo de error fue y
- Dónde ocurrió.

Los errores de sintaxis son generalmente fáciles de encontrar, pero hay algunas trampas. Los errores de espacio en blanco pueden ser complicados porque los espacios y las sangrías son invisibles y estamos acostumbrados a ignorarlos.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

En este ejemplo, el problema es que la segunda línea está desajustada por un espacio. Pero el mensaje de error señala a `y`, lo cual es engañoso. En general, los mensajes de error indican dónde fue descubierto el problema, pero el error real podría estar antes en el código, a veces en una línea anterior. Lo mismo ocurre con los errores de tiempo de ejecución.

Supongamos que estás intentando calcular una relación señal/ruido en decibeles. La fórmula es $RSR_{db} = 10 \log_{10}(P_{señal}/P_{ruido})$. En Python, podrías escribir algo así:

```
import math
potencia_senal = 9
potencia_ruido = 10
relacion = potencia_senal // potencia_ruido
decibeles = 10 * math.log10(relacion)
print(decibeles)
```

Cuando ejecutas este programa, obtienes una excepción:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibeles = 10 * math.log10(relacion)
ValueError: math domain error
```

El mensaje de error indica la línea 5, pero no hay nada malo con esa línea. Para encontrar el error real, podría ser útil imprimir el valor de `relacion`, el cual resulta ser 0. El problema está en la línea 4, que usa división entera en lugar de división de coma flotante.

Deberías tomarte el tiempo de leer cuidadosamente los mensajes de error, pero no supongas que todo lo que dice es correcto.

5.13. Glosario

división entera: Un operador, denotado por `//`, que divide dos números y redondea a un entero hacia abajo (en sentido hacia el infinito negativo).

operador de módulo: Un operador, denotado con un signo de porcentaje (`%`), que trabaja con enteros y revuelve el resto de dividir un número por otro.

expresión booleana: Una expresión cuyo valor es `True` o `False`.

operador relacional: Uno de los operadores que comprara sus operandos: `==`, `!=`, `>`, `<`, `>=` y `<=`.

operador lógico: Uno de los operadores que combina expresiones booleanas: `and`, `or` y `not`.

sentencia condicional: Una sentencia que controla el flujo de ejecución dependiendo de una condición.

condición: La expresión booleana en una sentencia condicional que determina cuál rama se ejecuta.

sentencia compuesta: Una sentencia que consiste en un encabezado y un cuerpo. El encabezado termina con un signo de dos puntos (:). El cuerpo tiene sangrías relativas al encabezado.

rama: Una de las secuencias de sentencias alternativas en una sentencia condicional.

condicional encadenado: Una sentencia condicional con una serie de ramas alternativas.

condicional anidado: Una sentencia condicional que aparece en una de las ramas de otra sentencia condicional.

sentencia return: Una sentencia que provoca que una función termine inmediatamente y vuelva a la sentencia llamadora.

recursividad: El proceso de llamar a la función que ya se está ejecutando.

caso base: Una rama condicional de una función recursiva que no hace una llamada recursiva.

recursividad infinita: Una recursividad que no tiene un caso base, o nunca lo alcanza. Eventualmente, una recursividad infinita provoca un error de tiempo de ejecución.

5.14. Ejercicios

Ejercicio 5.1. *El módulo* `time` *proporciona una función, con el mismo nombre* `time`*, que devuelve el tiempo transcurrido desde la Hora Media de Greenwich (GMT) en "la época" (*epoch*), la cual es un momento arbitrario usado como punto de referencia. En sistemas UNIX, la época es el 1 de enero de 1970.*

```
>>> import time
>>> time.time()
1437746094.5735958
```

Escribe un script que lea el tiempo actual y lo convierta a una hora del día en horas, minutos y segundos, además del número de días desde la época.

Ejercicio 5.2. *El Último Teorema de Fermat dice que no hay enteros positivos a, b y c tales que*

$$a^n + b^n = c^n$$

para cualquier valor de n mayor que 2.

1. *Escribe una función con nombre* `comprobar_fermat` *que tome cuatro parámetros —*`a`*,* `b`*,* `c` *y* `n`*— y compruebe si se cumple el teorema de Fermat. Si n es mayor que 2 y*

   $$a^n + b^n = c^n$$

   el programa debería imprimir "¡Ay caramba, Fermat se equivocó!". De lo contrario, el programa debería imprimir "No, eso no funciona."

2. *Escribe una función que permita al usuario ingresar valores para* `a`*,* `b`*,* `c` *y* `n`*, los convierta a enteros y use a la función* `comprobar_fermat` *para comprobar si violan el teorema de Fermat.*

Ejercicio 5.3. *Si te dan tres palos, podrías ser capaz o no de formar un triángulo. Por ejemplo, si uno de los palos mide 12 pulgadas y los otros dos miden una pulgada, no serás capaz de hacer que los palos cortos se encuentren en el medio. Para tres longitudes cualesquiera, hay una prueba simple para ver si es posible formar un triángulo:*

> *Si cualquiera de las tres longitudes es mayor que la suma de las otras dos, entonces no puedes formar un triángulo. De lo contrario, sí puedes. (Si la suma de dos longitudes es igual a la tercera, forman lo que llaman un triángulo "degenerado".)*

1. *Escribe una función con nombre* `es_triangulo` *que tome tres enteros como argumentos e imprima "Sí" o "No", dependiendo de si puedes o no formar un triángulo con palos cuyas longitudes sean los enteros dados.*

2. *Escribe una función que permita al usuario ingresar tres longitudes de palos, los convierta a enteros y use a la función* `es_triangulo` *para comprobar si los palos con las longitudes dadas pueden formar un triángulo.*

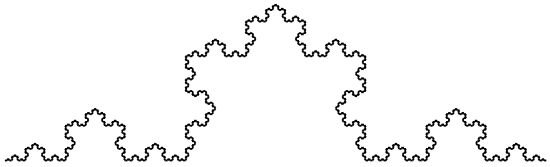

Figura 5.2: Una curva de Koch.

Ejercicio 5.4. *¿Cuál es la salida del siguiente programa? Dibuja un diagrama de pila que muestre el estado del programa cuando imprime el resultado.*

```
def recursivo(n, s):
    if n == 0:
        print(s)
    else:
        recursivo(n-1, n+s)

recursivo(3, 0)
```

1. *¿Qué ocurriría si llamas a esta función así:* `recursivo(-1, 0)`*?*
2. *Escribe un docstring que explique todo lo que alguien necesitaría saber para usar esta función (y nada más).*

Los siguientes ejercicios usan el módulo `turtle`, descrito en el Capítulo 4:
Ejercicio 5.5. *Lee la siguiente función y mira si puedes averiguar lo que hace (mira los ejemplos en el Capítulo 4). Luego ejecútala y mira si la entendiste bien.*

```
def dibujar(t, longitud, n):
    if n == 0:
        return
    angulo = 50
    t.fd(longitud*n)
    t.lt(angulo)
    dibujar(t, longitud, n-1)
    t.rt(2*angulo)
    dibujar(t, longitud, n-1)
    t.lt(angulo)
    t.bk(longitud*n)
```

Ejercicio 5.6. *La curva de Koch es un fractal que se ve algo como la Figura 5.2. Para dibujar una curva de Koch con longitud x, todo lo que tienes que hacer es*

1. *Dibujar una curva de Koch con longitud $x/3$.*
2. *Girar 60 grados a la izquierda.*
3. *Dibujar una curva de Koch con longitud $x/3$.*
4. *Girar 120 grados a la derecha.*
5. *Dibujar una curva de Koch con longitud $x/3$.*
6. *Girar 60 grados a la izquierda.*

7. *Dibujar una curva de Koch con longitud $x/3$.*

La excepción es si x es menor que 3: en ese caso, puedes simplemente dibujar una línea recta con longitud x.

1. *Escribe una función llamada* `koch` *que tome una tortuga y una longitud como parámetros y que use a la tortuga para dibujar una curva de Koch con la longitud dada.*

2. *Escribe una función llamada* `copo_de_nieve` *que dibuje tres curvas de Koch que hagan el contorno de un copo de nieve.*

   Solución: `http://thinkpython2.com/code/koch.py`.

3. *La curva de Koch se puede generalizar en muchas formas. Mira* `http://en.wikipedia.org/wiki/Koch_snowflake` *para ejemplos e implementa tu favorito.*

Capítulo 6

Funciones productivas

Muchas de las funciones de Python que hemos usado, tales como las funciones matemáticas, producen valores de retorno. Pero las funciones que hemos escrito son todas nulas: tienen un efecto, como imprimir un valor o mover una tortuga, pero no tienen un valor de retorno. En este capítulo aprenderás a escribir funciones productivas.

6.1. Valores de retorno

Al llamar a una función se genera un valor de retorno, que usualmente asignamos a una variable o usamos como parte de una expresión.

```
e = math.exp(1.0)
altura = radio * math.sin(radianes)
```

Las funciones que hemos escrito hasta ahora son todas nulas. Dicho en términos vagos, no tienen valor de retorno; de manera más precisa, su valor de retorno es `None`.

En este capítulo, vamos a escribir (finalmente) funciones productivas. El primer ejemplo es `area`, que devuelve el área de un círculo con un radio dado:

```
def area(radio):
    a = math.pi * radio**2
    return a
```

Hemos visto la sentencia `return` antes, pero en una función productiva la sentencia `return` incluye una expresión. Esta sentencia significa: "Sal inmediatamente de esta función y usa la siguiente expresión como valor de retorno." La expresión puede ser arbitrariamente complicada, por lo que podríamos haber escrito esta función de manera más concisa:

```
def area(radio):
    return math.pi * radio**2
```

Por otra parte, las **variables temporales** como `a` pueden hacer más fácil la depuración.

A veces es útil tener múltiples sentencias return, una en cada rama de un condicional:

```
def valor_absoluto(x):
    if x < 0:
        return -x
```

```
    else:
        return x
```

Dado que estas sentencias `return` están en un condicional alternativo, solo se ejecuta una.

Tan pronto como se ejecute una sentencia return, la función termina sin ejecutar ninguna de las sentencias posteriores. El código que aparece después de una sentencia `return`, o cualquier otro lugar que el flujo de ejecución nunca puede alcanzar, se llama **código muerto**.

En una función productiva, es una buena idea asegurarse de que cada camino posible a través del programa llegue a una sentencia `return`. Por ejemplo:

```
def valor_absoluto(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Esta función es incorrecta porque si `x` es 0, ninguna condición es verdadera, y la función termina sin llegar a una sentencia `return`. Si el flujo de ejecución llega al final de una función, el valor de retorno es `None`, lo cual no es el valor absoluto de 0.

```
>>> print(valor_absoluto(0))
None
```

Por cierto, Python proporciona una función incorporada llamada `abs` que calcula valores absolutos.

Como ejercicio, escribe una función `comparar` que tome dos valores, `x` e `y`, y devuelva `1` si `x > y`, devuelva `0` si `x == y` o devuelva `-1` si `x < y`.

6.2. Desarrollo incremental

A medida vayas escribiendo funciones más grandes, podrías encontrarte pasando más tiempo depurando.

Para lidiar con programas cada vez más complejos, tal vez quieras intentar un proceso llamado **desarrollo incremental**. El objetivo del desarrollo incremental es evitar largas sesiones de depuración agregando y probando solo un pedazo de código a la vez.

Como ejemplo, supongamos que quieres encontrar la distancia entre dos puntos, dados por las coordenadas (x_1, y_1) y (x_2, y_2). Por el teorema de Pitágoras, la distancia es:

$$\text{distancia} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

El primer paso es considerar cómo debería ser una función de `distancia` en Python. En otras palabras, ¿cuáles son las entradas (parámetros) y cuál es la salida (valor de retorno)?

En este caso, las entradas son dos puntos, que puedes representar usando cuatro números. El valor de retorno es la distancia representada por un valor de coma flotante.

Inmediatamente puedes escribir un esbozo de la función:

```
def distancia(x1, y1, x2, y2):
    return 0.0
```

Obviamente, esta versión no calcula distancias; siempre devuelve cero. Pero es sintácticamente correcta, y funciona, lo cual significa que puedes probarla antes de que la hagas más complicada.

Para probar la nueva función, llámala con argumentos de prueba:

```
>>> distancia(1, 2, 4, 6)
0.0
```

Escojo estos valores para que la distancia horizontal sea 3 y la distancia vertical sea 4; de esa forma, el resultado es 5, la hipotenusa de un triángulo rectángulo 3-4-5. Al probar una función, es útil saber la respuesta correcta.

En este punto hemos confirmado que la función es sintácticamente correcta y podemos comenzar agregando código al cuerpo. Un siguiente paso razonable es encontrar las diferencias $x_2 - x_1$ y $y_2 - y_1$. La siguiente versión almacena esos valores en variables temporales y las imprime.

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx es', dx)
    print('dy es', dy)
    return 0.0
```

Si la función está bien, debería mostrar `dx es 3` y `dy es 4`. Si es así, sabemos que la función obtiene los argumentos correctos y realiza el primer cálculo de manera correcta. Si no, solo hay que revisar unas pocas líneas.

Luego calculamos la suma de los cuadrados de `dx` y `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dcuadrado = dx**2 + dy**2
    print('dcuadrado es: ', dcuadrado)
    return 0.0
```

De nuevo, tendrías que ejecutar el programa en este punto y verificar la salida (que debería ser 25). Finalmente, puedes usar `math.sqrt` para calcular y devolver el resultado:

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dcuadrado = dx**2 + dy**2
    resultado = math.sqrt(dcuadrado)
    return resultado
```

Si eso funciona de manera correcta, estás listo. De lo contrario, tal vez quieras imprimir el valor de `resultado` antes de la sentencia return.

La versión final de la función no muestra nada cuando se ejecuta; solo devuelve un valor. Las sentencias `print` que escribimos son útiles para depurar, pero una vez que la función esté bien, deberías borrarlas. Un código como ese se llama **andamiaje** (en inglés, *scaffolding*) porque es útil para construir el programa pero no es parte del producto final.

Cuando empieces, deberías agregar solo una o dos líneas de código a la vez. A medida que ganes experiencia, podrías encontrarte escribiendo y depurando partes más grandes. De cualquier manera, el desarrollo incremental puede ahorrarte mucho tiempo de depuración.

Los aspectos clave del proceso son:

1. Comienza con un programa que funcione y haz pequeños cambios incrementales. En cualquier punto, si hay un error, deberías tener una buena idea de dónde está.

2. Usa variables que guarden valores intermedios de tal manera que puedas mostrarlos y verificarlos.

3. Una vez que el programa funciona, tal vez quieras borrar algo del andamiaje o consolidar varias sentencias en una expresión compuesta, pero solo si no hace al programa difícil de leer.

Como ejercicio, usa desarrollo incremental para escribir una función llamada `hipotenusa` que devuelva el largo de la hipotenusa de un triángulo rectánculo dadas las longitudes de los otros dos lados como argumentos. Graba cada etapa del proceso de desarrollo a medida que avances.

6.3. Composición

Como ya deberías esperar, puedes llamar a una función desde dentro de otra. Como ejemplo, escribiremos una función que tome dos puntos, el centro de un círculo y un punto de su perímetro, y calcule el área del círculo.

Supongamos que el punto central se almacena en las variables `xc` e `yc`, y el punto del perímetro está en `xp` e `yp`. El primer paso es encontrar el radio del círculo, que es la distiancia entre los dos puntos. Acabamos de escribir una función, `distancia`, que hace eso:

```
radio = distancia(xc, yc, xp, yp)
```

El siguiente paso es encontrar el área de un círculo con ese radio; también escribimos eso:

```
resultado = area(radio)
```

Encapsulando estos pasos en una función, obtenemos:

```
def area_circulo(xc, yc, xp, yp):
    radio = distancia(xc, yc, xp, yp)
    resultado = area(radio)
    return resultado
```

Las variables temporales `radio` y `resultado` son útiles para el desarrollo y la depuración, pero una vez que el programa funciona, podemos hacerlo más conciso componiendo las llamadas a funciones:

```
def area_circulo(xc, yc, xp, yp):
    return area(distancia(xc, yc, xp, yp))
```

6.4. Funciones booleanas

Las funciones pueden devolver booleanos, lo cual es a menudo conveniente para ocultar pruebas complicadas dentro de las funciones. Por ejemplo:

```
def es_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Es común darle a las funciones booleanas nombres que suenen como preguntas sí/no; `es_divisible` devuelve `True` o `False` para indicar si `x` es divisible por `y`.

Aquí hay un ejemplo:

```
>>> es_divisible(6, 4)
False
>>> es_divisible(6, 3)
True
```

El resultado del operador `==` es un booleano, por lo que podemos escribir la función de manera más concisa devolviéndolo directamente:

```
def es_divisible(x, y):
    return x % y == 0
```

Las funciones booleanas se usan a menudo en las sentencias condicionales:

```
if es_divisible(x, y):
    print('x es divisible por y')
```

Puede ser tentador escribir algo como:

```
if es_divisible(x, y) == True:
    print('x es divisible por y')
```

Pero la comparación extra es innecesaria.

Como ejercicio, escribe una función `esta_entre(x, y, z)` que devuelva `True` si $x \leq y \leq z$ o `False` si no.

6.5. Más recursividad

Hemos cubierto solo un pequeño subconjunto de Python, pero tal vez te interese saber que este subconjunto es un lenguaje de programación *completo*, lo cual significa que cualquier cosa que pueda ser calculada se puede expresar en este lenguaje. Cualquier programa alguna vez escrito puede ser reescrito usando solo las características del lenguaje que has aprendido hasta ahora (en realidad, necesitarías unos pocos comandos para controlar dispositivos como el mouse, discos, etc., pero eso eso todo).

Probar ese aserto es un ejercicio no trivial realizado por primera vez por Alan Turing, uno de los primeros informáticos (algunos discuten que fue un matemático, pero muchos de los primeros informáticos comenzaron como matemáticos). En consecuencia, se conoce como Tesis de Turing. Para un tratamiento más completo (y preciso) de la Tesis de Turing, recomiendo el libro de Michael Sipser *Introduction to the Theory of Computation*.

Para darte una idea de lo que puedes hacer con las herramientas que has aprendido hasta ahora, evaluaremos algunas funciones matemáticas definidas de forma recursiva. Una definición recursiva es similar a una definición circular, en el sentido de que la definición contiene una referencia a lo que se está definiendo. Una definición verdaderamente circular no es muy útil:

vorpal: Un adjetivo usado para describir algo que es vorpal.

Si vieras esa definición en el diccionario, quizás te moleste. Por otra parte, si buscaras la definición de la función factorial, denotada con el símbolo !, podrías obtener algo así:

$$0! = 1$$
$$n! = n(n-1)!$$

Esta definición dice que el factorial de 0 es 1 y el factorial de cualquier otro valor, n, es n multiplicado por el factorial de $n-1$.

Así que 3! es 3 por 2!, lo cual es 2 por 1!, lo cual es 1 por 0!. Poniéndolo todo junto, 3! es igual a 3 por 2 por 1 por 1, lo cual es 6.

Si puedes escribir una definición recursiva de algo, puedes escribir un programa en Python para evaluarla. El primer paso es decidir cuáles deberían ser los parámetros. En este caso debería estar claro que `factorial` toma un entero:

```
def factorial(n):
```

Si el argumento es 0, todo lo que tenemos que hacer es devolver 1:

```
def factorial(n):
    if n == 0:
        return 1
```

De lo contrario, y esta es la parte interesante, tenemos que hacer una llamada recursiva para encontrar el factorial de $n-1$ y luego multiplicarlo por n:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recur = factorial(n-1)
        resultado = n * recur
        return resultado
```

El flujo de ejecución para este programa es similar al flujo de `cuenta_reg` de la Sección 5.8. Si llamamos a `factorial` con el valor 3:

Dado que 3 no es 0, tomamos la segunda rama y calculamos el factorial de `n-1`...

> Dado que 2 no es 0, tomamos la segunda rama y calculamos el factorial de `n-1`...
>
> > Dado que 1 no es 0, tomamos la segunda rama y calculamos el factorial de `n-1`...
> >
> > > Dado que 0 es igual a 0, tomamos la primera rama y devolvemos 1 sin hacer más llamadas recursivas.
> >
> > El valor de retorno, 1, se multiplica por n, que es 1, y se devuelve el resultado.
>
> El valor de retorno, 1, se multiplica por n, que es 2, y se devuelve el resultado.

El valor de retorno (2) se multiplica por n, que es 3, y el resultado, 6, se convierte en el valor de retorno de la llamada a función que comenzó todo el proceso.

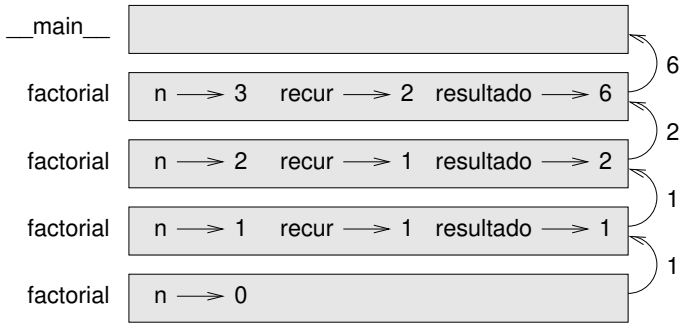

Figura 6.1: Diagrama de pila.

La Figura 6.1 muestra cómo se ve el diagrama de pila para esta sucesión de llamadas a función.

Los valores de retorno se muestran volviendo hacia arriba en la pila. En cada marco, el valor de retorno es el valor de `resultado`, que es el producto de `n` y `recur`.

En el último marco, las variables locales `recur` y `resultado` no existen, porque la rama que las crea no se ejecuta.

6.6. Salto de fe

Seguir el flujo de ejecución es una manera de leer programas, pero puede volverse abrumador rápidamente. Una alternativa es lo que yo llamo el "salto de fe". Cuando llegas a una llamada a función, en lugar de seguir el flujo de ejecución, *supones* que la función funciona correctamente y que devuelve el resultado correcto.

De hecho, ya estás practicando este salto de fe cuando usas funciones incorporadas. Cuando llamas a `math.cos` o `math.exp`, no examinas los cuerpos de esas funciones. Sólo supones que funcionan porque las personas que escribieron las funciones incorporadas eran buenos programadores.

Lo mismo es verdad cuando llamas a una de tus propias funciones. Por ejemplo, en la Sección 6.4, escribimos una función llamada `es_divisible` que determina si un número es divisible por otro. Una vez que nos hemos convencido de que esta función es correcta —examinando el código y probándolo— podemos usar la función sin mirar el cuerpo otra vez.

Lo mismo es verdad para las funciones recursivas. Cuando llegues a la llamada recursiva, en lugar de seguir el flujo de ejecución, deberías suponer que la llamada recursiva funciona (devuelve el resultado correcto) y luego preguntarte: "suponiendo que puedo encontrar el factorial de $n - 1$, ¿puedo calcular el factorial de n?" Está claro que puedes, multiplicando por n.

Desde luego, es un poco extraño suponer que la función hace lo correcto cuando no has terminado de escribirla, ¡pero es por eso que se llama salto de fe!

6.7. Un ejemplo más

Después de `factorial`, el ejemplo más común de una función matemática definida de manera recursiva es `fibonacci`, que tiene la siguiente definición (ver `http://es.wikipedia.org/wiki/Sucesión_de_Fibonacci`):

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2) \end{aligned}$$

Traducido a Python, se ve así:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Si intentas seguir el flujo de ejecución aquí, incluso para valores de n bastante pequeños, tu cabeza estalla. Pero de acuerdo al salto de fe, si supones que las dos llamadas recursivas funcionan correctamente, entonces está claro que obtienes el resultado correcto al sumarlas.

6.8. Verificar tipos

¿Qué ocurre si llamamos a `factorial` y le entregamos 1.5 como argumento?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

Se ve como una recursividad infinita. ¿Cómo puede ser? La función tiene un caso base: cuando `n == 0`. Pero si `n` no es un entero, podemos *perder* el caso base y seguir con la recursividad sin parar.

En la primera llamada recursiva, el valor de `n` es 0.5. En la siguiente, es -0.5. Desde allí, se hace menor (más negativo), pero nunca será 0.

Tenemos dos opciones. Podemos intentar generalizar la función `factorial` para que funcione con números de coma flotante o podemos hacer que `factorial` verifique el tipo de sus argumentos. La primera opción se llama la función gamma y está un poco más allá del alcance de este libro. Entonces iremos por la segunda.

Podemos usar la función incorporada `isinstance` para verificar el tipo del argumento. Mientras estemos en ello, podemos también asegurarnos de que el argumento sea positivo:

```
def factorial(n):
    if not isinstance(n, int):
        print('El factorial solo está definido para enteros.')
        return None
    elif n < 0:
```

```
        print('El factorial no está definido para enteros negativos.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

El primer caso base se encarga de los no enteros; el segundo se encarga de los enteros negativos. En ambos casos, el programa imprime un mensaje de error y devuelve `None` para indicar que algo anduvo mal:

```
>>> print(factorial('fred'))
El factorial solo está definido para enteros.
None
>>> print(factorial(-2))
El factorial no está definido para enteros negativos.
None
```

Si pasamos ambas verificaciones, sabemos que n es un entero no negativo, por lo que podemos probar que la recursividad termina.

Este programa demuestra un patrón a veces llamado **guardián**. Los primeros dos condicionales actúan como guardianes, protegiendo el código que sigue de valores que podrían provocar un error. Los guardianes hacen posible probar la exactitud del código.

En la Sección 11.4 veremos una alternativa más flexible que imprime un mensaje de error: plantear una excepción.

6.9. Depuración

Separar un programa grande en funciones pequeñas crea puntos de control naturales para la depuración. Si una función no está funcionando, hay tres posibilidades a considerar:

- Hay algo mal en los argumentos que obtiene la función; se viola una precondición.
- Hay algo mal en la función; se viola una postcondición.
- Hay algo mal en el valor de retorno o la manera en que se usa.

Para descartar la primera posibilidad, puedes agregar una sentencia `print` al comienzo de la función y mostrar los valores de los parámetros (y quizás sus tipos). O bien puedes escribir código que verifique las precondiciones de manera explícita.

Si los parámetros se ven bien, agrega una sentencia `print` antes de cada sentencia `return` y muestra el valor de retorno. Si es posible, verifica el resultado a mano. Considera llamar a la función con valores que faciliten la verificación del resultado (como en la Sección 6.2).

Si la función parece funcionar, mira la llamada a función para asegurarte de que el valor de retorno se está usando correctamente (¡o si al menos se está usando!).

Agregar sentencias print al comienzo y al final de una función puede ayudar a hacer más visible el flujo de ejecución. Por ejemplo, aquí hay una versión de `factorial` con sentencias print:

```
def factorial(n):
    espacio = ' ' * (4 * n)
    print(espacio, 'factorial', n)
    if n == 0:
        print(espacio, 'devolviendo 1')
        return 1
    else:
        recursivo = factorial(n-1)
        resultado = n * recursivo
        print(espacio, 'devolviendo', resultado)
        return resultado
```

`espacio` es una cadena de caracteres de espacio que controla la sangría de la salida. Aquí está el resultado de `factorial(4)` :

```
                factorial 4
            factorial 3
         factorial 2
     factorial 1
 factorial 0
 devolviendo 1
     devolviendo 1
         devolviendo 2
             devolviendo 6
                 devolviendo 24
```

Si estás confundido acerca del flujo de ejecución, este tipo de salidas puede ser útil. Desarrollar andamiaje eficaz toma algo de tiempo, pero un poco de andamiaje puede ahorrar mucha depuración.

6.10. Glosario

variable temporal: Una variable usada para almacenar un valor intermedio en una computación compleja.

código muerto: Parte de un programa que nunca se ejecuta, a menudo debido a que aparece después de una sentencia `return`.

desarrollo incremental: Un plan de desarrollo de programa destinado a evitar la depuración agregando y probando solo un pedazo de código a la vez.

andamiaje: Código que se usa durante el desarrollo de un programa pero no es parte de la versión final.

guardián: Un patrón de programación que usa una sentencia condicional para verificar y encargarse de circunstancias que podrían provocar un error.

6.11. Ejercicios

Ejercicio 6.1. *Dibuja un diagrama de pila para el siguiente programa. ¿Qué imprime el programa?*

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    cuadrado = b(total)**2
    return cuadrado

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

Ejercicio 6.2. *La función de Ackermann, $A(m,n)$, se define:*

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ y } n = 0 \\ A(m-1,A(m,n-1)) & \text{si } m > 0 \text{ y } n > 0. \end{cases}$$

Ver http://es.wikipedia.org/wiki/Función_de_Ackermann. Escribe una función con nombre `ack` *que evalúe la función de Ackermann. Usa tu función para evaluar* `ack(3, 4)`*, que debería ser 125. ¿Que ocurre para valores más grandes de* `m` *y* `n`*? Solución: http://thinkpython2.com/code/ackermann.py.*

Ejercicio 6.3. *Un palíndromo es una palabra que se deletrea igual hacia atrás y hacia adelante, como "noon" y "redivider". De manera recursiva, una palabra es un palíndromo si la primera y la última letra son la misma y el medio es un palíndromo.*

Las siguientes son funciones que toman una cadena como argumento y devuelven las letras primera, última y del medio:

```
def primera(palabra):
    return palabra[0]

def ultima(palabra):
    return palabra[-1]

def medio(palabra):
    return palabra[1:-1]
```

Veremos cómo funcionan en el Capítulo 8.

1. *Escribe estas funciones en un archivo con nombre* `palindromo.py` *y pruébalas. ¿Qué ocurre si llamas a* `medio` *con una cadena de dos letras? ¿De una letra? ¿Qué hay de la cadena vacía, la cual se escribe* `''` *y no contiene letras?*

2. *Escribe una función llamada* `es_palindromo` *que tome una cadena como argumento y devuelva* `True` *si es un palíndromo y* `False` *si no. Recuerda que puedes usar la función incorporada* `len` *para verificar la longitud de una cadena.*

Solución: `http://thinkpython2.com/code/palindrome_soln.py`.

Ejercicio 6.4. *Un número, a, es potencia de b si es divisible por b y además a/b es potencia de b. Escribe una función llamada* `es_potencia` *que tome parámetros* `a` *y* `b` *y devuelva* `True` *si* `a` *es potencia de* `b`*. Nota: tendrás que pensar en el caso base.*

Ejercicio 6.5. *El máximo común divisor (MCD) de a y b es el número más grande que divide a ambos sin obtener resto.*

Una manera de encontrar el MCD de dos números está basada en la observación de que si r es el resto de dividir a por b, entonces $mcd(a, b) = mcd(b, r)$*. Como caso base, podemos usar* $mcd(a, 0) = a$*.*

Escribe una función llamada `mcd` *que tome parámetros* `a` *y* `b` *y devuelva su máximo común divisor.*

Crédito: Este ejercicio está basado en un ejemplo de Structure and Interpretation of Computer Programs *de Abelson y Sussman.*

Capítulo 7

Iteración

Este capítulo trata sobre la iteración, que es la capacidad de ejecutar un bloque de sentencias de forma repetida. Vimos un tipo de iteración, usando recursividad, en la Sección 5.8. Vimos otro tipo, usando un blucle `for`, en la Sección 4.2. Es este capítulo veremos otro tipo, usando una sentencia `while`. Pero primero quiero decir un poco más sobre la asignación de variables.

7.1. Reasignación

Como habrás descubierto, es legal hacer más de una asignación a la misma variable. Una nueva asignación hace que una variable existente se refiera a un nuevo valor (y deje de referirse al valor antiguo).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

La primera vez que mostramos `x`, su valor es 5; la segunda vez, su valor es 7.

La Figura 7.1 muestra cómo se ve una **reasignación** en un diagrama de estado.

En este punto quiero abordar un origen común de confusión. Debido a que Python usa el signo igual (=) para la asignación, es tentador interpretar una sentencia del tipo `a = b` como una proposición matemática de igualdad; es decir, la afirmación de que `a` y `b` son iguales. Pero esta interpretación es incorrecta.

Primero, la igualdad es una relación simétrica y la asignación no lo es. Por ejemplo, en matemáticas, si $a = 7$ entonces $7 = a$. Pero en Python, la sentencia `a = 7` es legal y `7 = a` no lo es.

Además, en matemáticas, una proposición de igualdad es verdadera o falsa todo el tiempo. Si $a = b$ ahora, entonces a siempre será igual a b. En Python, una sentencia de asignación puede igualar dos variables, pero no tienen que permanecer iguales:

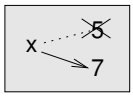

Figura 7.1: Diagrama de estado.

```
>>> a = 5
>>> b = a    # a y b son iguales ahora
>>> a = 3    # a y b ya no son iguales
>>> b
5
```

La tercera línea cambia el valor de `a` pero no cambia el valor de `b`, por lo que ya no son iguales.

La reasignación de variables es a menudo útil, pero deberías usarla con precaución. Si los valores de las variables cambian frecuentemente, puede hacer que el código sea difícil de leer y depurar.

7.2. Actualizar variables

Un tipo común de reasignación es la **actualización**, donde el nuevo valor de la variable depende del antiguo.

```
>>> x = x + 1
```

Esto significa "obten el valor actual de `x`, suma uno, y luego actualiza `x` con el nuevo valor."

Si intentas actualizar una variable que no existe, obtienes un error, debido a que Python evalúa el lado derecho antes de asignar un valor a `x`:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Antes de que puedas actualizar una variable, la tienes que **inicializar**, usualmente con una simple asignación:

```
>>> x = 0
>>> x = x + 1
```

Actualizar una variable sumando 1 se llama **incremento**; restando 1 se llama **decremento**.

7.3. La sentencia `while`

Los computadores a menudo se usan para automatizar tareas repetitivas. Repetir tareas idénticas o similares sin cometer errores es algo que los computadores hacen bien y las personas hacen mal. En un programa de computador, la repetición también se llama **iteración**.

Ya hemos visto dos funciones, `cuenta_reg` e `imprimir_n`, que iteran usando recursividad. Debido a que la iteración es tan común, Python proporciona características del lenguaje que la hacen más fácil. Una es la sentencia `for` que vimos en la Sección 4.2. Volveremos a eso más adelante.

Otra es la sentencia `while`. Aquí hay una versión de `cuenta_reg` que usa una sentencia `while`:

```
def cuenta_reg(n):
    while n > 0:
        print(n)
        n = n - 1
    print('¡Despegue!')
```

Casi puedes leer la sentencia `while` como si fuera inglés. Significa, "Mientras `n` sea mayor que 0, muestra el valor de `n` y luego decrementa `n`. Cuando llegues a 0, muestra la palabra `¡Despegue!`"

De manera más formal, aquí está el flujo de ejecución para una sentencia `while`:

1. Determinar si la condición es verdadera o falsa.
2. Si es falsa, salir de la sentencia `while` y continuar con la ejecucion de la siguiente sentencia.
3. Si la condición es verdadera, ejecutar el cuerpo y luego volver al paso 1.

Este tipo de flujo se llama bucle porque el tercer paso hace que vuelva hacia arriba.

El cuerpo del bucle debería cambiar el valor de una o más variables de modo que la condición se vuelva falsa eventualmente y el bucle termine. De lo contrario, el bucle se repetirá por siempre, lo cual se llama **bucle infinito**. Una fuente interminable de diversión para informáticos es la observación de que las instrucciones en un champú, "Enjabonar, enjuagar, repetir", son un bucle infinito.

En el caso de `cuenta_reg`, podemos probar que el bucle termina: si `n` es cero o negativo, el bucle nunca se ejecuta. De lo contrario, `n` se hace más pequeño cada vez que se pasa por el blucle, por lo que eventualmente tenemos que llegar a 0.

Para otros bucles, no es tan fácil decirlo. Por ejemplo:

```
def sucesion(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n es par
            n = n / 2
        else:                 # n es impar
            n = n*3 + 1
```

La condición para este bucle es `n != 1`, por lo que el bucle continuará hasta que `n` sea `1`, que hace que la condición sea falsa.

En cada paso por el bucle, el programa muestra el valor de `n` y luego verifica si es par o impar. Si es par, `n` se divide por 2. Si es impar, el valor de `n` se reemplaza por `n*3 + 1`. Por ejemplo, si el argumento pasado a `sucesion` es 3, los valores resultantes de `n` son 3, 10, 5, 16, 8, 4, 2, 1.

Dado que `n` a veces aumenta y a veces disminuye, no hay demostración obvia de que `n` alcanzará el 1 alguna vez, o que el programa termina. Para algunos valores particulares de `n`, podemos probar que termina. Por ejemplo, si el valor inicial es una potencia de dos,

n será par cada vez que se pase por el bucle hasta que alcance el 1. El ejemplo anterior termina con tal sucesión, comenzando con 16.

La pregunta difícil es si podemos probar que este programa termina para *todos* los valores positivos de n. Hasta ahora, ¡nadie ha sido capaz de probarlo *o* refutarlo! (Ver http://es.wikipedia.org/wiki/Conjetura_de_Collatz.)

Como ejercicio, reescribe la función print_n de la Sección 5.8 usando iteración en lugar de recursividad.

7.4. break

A veces no sabes que es momento de terminar un bucle hasta que llegas a la mitad del cuerpo. En ese caso puedes usar la sentencia break para saltar hacia afuera del bucle.

Por ejemplo, supongamos que quieres tomar la entrada del usuario hasta que se escriba listo. Podrías escribir:

```
while True:
    linea = input('> ')
    if linea == 'listo':
        break
    print(linea)

print('¡Listo!')
```

La condición del bucle es True, lo cual siempre es verdadero, así que el bucle se ejecuta hasta que llega a la sentencia break.

En cada paso, solicita la entrada del usuario con un paréntesis angular. Si el usuario escribe listo, la sentencia break hace que se salga del bucle. De lo contrario, el programa repite lo que escriba el usuario y regresa a la parte superior del bucle. Aquí hay una ejecución de muestra:

```
> no listo
no listo
> listo
¡Listo!
```

Esta forma de escribir bucles while es común porque puedes v the type of a value.erificar la condición en cualquier lugar del bucle (no solo en la parte superior) y puedes expresar la condición de detención de manera afirmativa (“detente cuando esto ocurra”) en lugar de negativa (“sigue haciendo hasta que eso ocurra”).

7.5. Raíces cuadradas

Los bucles se usan a menudo en programas que calculan resultados numéricos iniciando con una respuesta aproximada y mejorándola iterativamente.

Por ejemplo, una manera de calcular raíces cuadradas es el método de Newton. Supongamos que quieres saber la raíz cuadrada de a. Si comienzas con casi cualquier estimación, x, puedes calcular una mejor estimación con la siguiente fórmula:

$$y = \frac{x + a/x}{2}$$

Por ejemplo, si a es 4 y x es 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

El resultado está más cerca de la respuesta correcta ($\sqrt{4} = 2$). Si repetimos el proceso con una nueva estimación, se acerca aún más:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

Después de algunas actualizaciones más, la estimación es casi exacta:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

En general no sabemos de antemano cuántos pasos toma llegar a la respuesta correcta, pero sabemos cuándo la obtenemos porque la estimación deja de cambiar:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

Cuando `y == x`, podemos parar. Aquí hay un bucle que comienza con una estimación inicial, `x`, y la mejora hasta que deja de cambiar:

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

Para la mayoría de los valores de `a` esto funciona bien, pero en general es peligroso probar la igualdad de números `float`. Los valores de coma flotante son solo aproximadamente correctos: la mayoría de los números racionales, como 1/3, y los números irracionales, como $\sqrt{2}$, no se pueden representar de manera exacta con un `float`.

En lugar de verificar si `x` e `y` son exactamente iguales, es más seguro usar la función `abs` para calcular el valor absoluto, o magnitud, de la diferencia entre estos:

```
    if abs(y-x) < epsilon:
        break
```

Donde `epsilon` tiene un valor como `0.0000001` que determina qué tan cerca es lo suficientemente cerca.

7.6. Algoritmos

El método de Newton es un ejemplo de **algoritmo**: es un proceso mecánico para resolver una categoría de problemas (en este caso, calcular raíces cuadradas).

Para entender qué es un algoritmo, quizás ayude comenzar con algo que no es un algoritmo. Cuando aprendiste a multiplicar números de un solo dígito, probablemente memorizaste la tabla de multiplicar. En realidad, memorizaste 100 soluciones específicas. Esa clase de conocimiento no es algorítmica.

Pero si eras "perezoso", podrías haber aprendido algunos trucos. Por ejemplo, para encontrar el producto de n y 9, puedes escribir $n - 1$ como el primer dígito y $10 - n$ como el segundo dígito. Este truco es una solución general para multiplicar cualquier número de un solo dígito por 9. ¡Eso es un algoritmo!

Del mismo modo, las técnicas que aprendiste para la suma con reserva, resta con préstamo y división larga son todas algoritmos. Una de las características de los algoritmos es que no requieren ninguna inteligencia para realizarlos. Son procesos mecánicos donde cada paso sigue al último de acuerdo a un conjunto simple de reglas.

Ejecutar algoritmos es aburrido, pero diseñarlos es interesante, intelectualmente desafiante y una parte central de las ciencias de la computación.

Algunas de las cosas que las personas hacen de manera natural, sin dificultad o pensamiento consciente, son las más difíciles de expresar de manera algorítmica. Entender un lenguaje natural es un buen ejemplo. Todos lo hacemos, pero hasta ahora nadie ha sido capaz de explicar *cómo* lo hacemos, al menos no en la forma de un algoritmo.

7.7. Depuración

A medida que comiences a escribir programas más grandes, podrías encontrarte pasando más tiempo depurando. Más código significa más posibilidades de cometer un error y más lugares para esconder errores de programación.

Una manera de acortar tu tiempo de depuración es la "depuración por bisección". Por ejemplo, si hay 100 líneas en tu programa y las revisas una a la vez, tomaría 100 pasos.

En cambio, intenta separar el problema por la mitad. Mira la mitad del programa, o cerca de esta, para un valor intermedio que puedas verificar. Agrega una sentencia `print` (o algo más que tenga un efecto verificable) y ejecuta el programa.

Si la verificación en el punto medio es incorrecta, debe haber un problema en la primera mitad del programa. Si es correcta, el problema está en la segunda mitad.

Cada vez que hagas una verificación como esta, reduces a la mitad el número de líneas que tienes que buscar. Después de seis pasos (lo cual es menos que 100), estarías con una o dos líneas de código, al menos en teoría.

En la práctica no siempre está claro cuál es la "mitad del programa" y no siempre es posible verificarla. No tiene sentido contar líneas y encontrar el punto medio exacto. En cambio, piensa en lugares en el programa donde podría haber errores y lugares donde es fácil poner una verificación. Luego, escoge un sitio donde creas que las posibilidades de que el error esté antes o después de la verificación son casi las mismas.

7.8. Glosario

reasignación: Asignar un nuevo valor a una variable que ya existe.

actualización: Una asignación donde el nuevo valor de una variable depende del antiguo.

inicialización: Una asignación que le da un valor inicial a una variable que será actualizada.

incremento: Una actualización que aumenta el valor de una variable (a menudo en uno).

decremento: Una actualización que disminuye el valor de una variable.

iteración: Ejecución repetida de un conjunto de sentencias usando una llamada a función recursiva o un bucle.

bucle infinito: Un bucle cuya condición para terminar nunca se satisface.

algoritmo: Un proceso general para resolver una categoría de problemas.

7.9. Ejercicios

Ejercicio 7.1. *Copia el bucle de la Sección 7.5 y encapsúlalo en una función llamada* `mi_sqrt` *que tome a* `a` *como parámetro, escoja un valor razonable de* `x` *y devuelva una estimación de la raíz cuadrada de* `a`*.*

Para probarla, escribe una función con nombre `probar_raiz_cuadrada` *que imprima una tabla como esta:*

```
a   mi_sqrt(a)    math.sqrt(a)  diferencia
-   ----------    ------------  ----------
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

La primera columna es un número, a; la segunda columna es la raíz cuadrada de a calculada con `mi_sqrt`*; la tercera columna es la raíz cuadrada calculada por* `math.sqrt`*; la cuarta columna es el valor absoluto de la diferencia entre las dos estimaciones.*

Ejercicio 7.2. *La función incorporada* `eval` *toma una cadena y la evalúa usando el intérprete de Python. Por ejemplo:*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

Escribe una función llamada `bucle_eval` *que, de manera iterativa, solicite la entrada del usuario, tome la entrada resultante y la evalúe usando* `eval`*, e imprima el resultado.*

Debería continuar hasta que el usuario ingrese `'listo'` *y luego devuelva el valor de la última expresión que evaluó.*

Ejercicio 7.3. *El matemático Srinivasa Ramanujan encontró una serie infinita que se puede usar para generar una aproximación numérica de* $1/\pi$*:*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Escribe una función llamada `estimacion_pi` *que use esta fórmula para calcular y devolver una estimación de* π*. Debería usar un bucle* `while` *para calcular términos de la sumatoria hasta que el último término sea más pequeño que* `1e-15` *(que es la notación de Python para* 10^{-15}*). Puedes verificar el resultado comparándolo con* `math.pi`*.*

Solución: `http://thinkpython2.com/code/pi.py`*.*

Capítulo 8

Cadenas

Las cadenas no son como los enteros, números de coma flotante y booleanos. Una cadena es una **secuencia**, lo cual significa que es una colección ordenada de valores. En este capítulo verás cómo acceder a los caracteres que forman una cadena y aprenderás sobre algunos de los métodos que proporcionan las cadenas.

8.1. Una cadena es una secuencia

Una cadena es una secuencia de caracteres. Puedes acceder a los caracteres uno a la vez con el operador de corchetes:

```
>>> fruta = 'banana'
>>> letra = fruta[1]
```

La segunda sentencia selecciona el carácter número 1 de `fruta` y lo asigna a `letra`.

La expresión en los corchetes se llama **índice**. El índice indica cuál carácter en la secuencia quieres (de ahí el nombre).

Pero podrías no obtener lo que esperas:

```
>>> letra
'a'
```

Para la mayoría de las personas, la primera letra de `'banana'` es `b`, no `a`. Pero para los informáticos, el índice es un desplazamiento desde el comienzo de la cadena, y el desplazamiento de la primera letra es cero.

```
>>> letra = fruta[0]
>>> letra
'b'
```

Entonces `b` es la 0-ésima letra (“cero-ésima”) de `'banana'`, `a` es la 1-ésima letra (“uno-ésima”) y `n` es la 2-ésima letra (“dos-ésima”).

Como índice, puedes usar una expresión que contenga variables y operadores:

```
>>> i = 1
>>> fruta[i]
```

```
'a'
>>> fruta[i+1]
'n'
```

Pero el valor del índice tiene que ser un entero. De lo contrario, obtienes:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers
```

8.2. `len`

`len` es una función incorporada que devuelve el número de caracteres en una cadena:

```
>>> fruta = 'banana'
>>> len(fruta)
6
```

Para obtener la última letra de una cadena, quizás te tientes a intentar algo así:

```
>>> longitud = len(fruta)
>>> ultima = fruta[longitud]
IndexError: string index out of range
```

El motivo del `IndexError` es que no hay letra en `'banana'` con el índice 6. Dado que comenzamos a contar desde cero, las seis letras se enumeran del 0 al 5. Para obtener el último carácter, tienes que restar 1 a `longitud`:

```
>>> ultima = fruta[longitud-1]
>>> ultima
'a'
```

O bien puedes usar índices negativos, que cuentan hacia atrás desde el final de la cadena. La expresión `fruta[-1]` entrega la última letra, `fruta[-2]` entrega la penúltima, y así sucesivamente.

8.3. Recorrido con un bucle `for`

Muchas computaciones implican procesar una cadena trabajando un carácter a la vez. A menudo parten al comienzo, seleccionan cada carácter en turno, le hacen algo, y continúan hasta el final. Este patrón de procesamiento se llama **recorrido**. Una manera de escribir un recorrido es con un bucle `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice = indice + 1
```

Este bucle recorre la cadena y muestra cada letra en una línea. La condición del bucle es `indice < len(fruta)`, entonces cuando `indice` es igual a la longitud de la cadena, la condición es falsa, y el cuerpo del bucle no se ejecuta. El último carácter accedido es el que tiene índice `len(fruta)-1`, que es el último carácter en la cadena.

Como ejercicio, escribe una función que tome una cadena como argumento y muestre las letras hacia atrás, una por línea.

Otra manera de escribir un recorrido es con un bucle `for`:

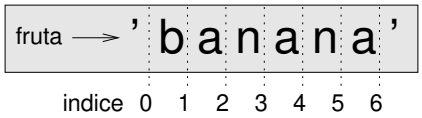

Figura 8.1: Índices de corte.

```
for letra in fruta:
    print(letra)
```

En cada paso por el bucle, el siguiente carácter en la cadena se asigna a la variable `letra`. El bucle continúa hasta que no quedan caracteres.

El siguiente ejemplo muestra cómo usar la concatenación (suma de cadenas) y un bucle `for` para generar una serie abecedaria (es decir, en orden alfabético). En el libro de Robert McCloskey *Abran paso a los patitos*, los nombres de los patitos son Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. Este bucle muestra dichos nombres en orden:

```
prefijos = 'JKLMNOPQ'
sufijo = 'ack'

for letra in prefijos:
    print(letra + sufijo)
```

La salida es:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Desde luego, eso no es del todo correcto porque "Ouack" y "Quack" están mal escritos. Como ejercicio, modifica el programa para arreglar este error.

8.4. Cortes de cadena

Un segmento de una cadena se llama **corte** (en inglés, *slice*). Seleccionar un corte es similar a seleccionar un carácter:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

El operador `[n:m]` devuelve la parte de la cadena desde el "n-ésimo" carácter al "m-ésimo" carácter, incluyendo el primero pero excluyendo el último. Este comportamiento es contra-intuitivo, pero tal vez ayude imaginar los índices apuntando *entre* los caracteres, como en la Figura 8.1.

Si omites el primer índice (antes del signo de dos puntos), el corte comienza al principio de la cadena. Si omites el segundo índice, el corte llega hasta el final de la cadena:

```
>>> fruta = 'banana'
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

Si el primer índice es mayor o igual al segundo, el resultado es una **cadena vacía**, representada por dos comillas:

```
>>> fruta = 'banana'
>>> fruta[3:3]
''
```

Una cadena vacía no contiene caracteres y tiene longitud 0, pero aparte de eso, es lo mismo que cualquier otra cadena.

Continuando con este ejemplo, ¿qué crees que significa `fruta[:]`? Pruébalo y mira.

8.5. Las cadenas son inmutables

Es tentador usar el operador `[]` en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena. Por ejemplo:

```
>>> saludo = 'Hola, mundo'
>>> saludo[0] = 'J'
TypeError: 'str' object does not support item assignment
```

El "objeto" en este caso es la cadena y el "ítem" es el carácter que intentaste asignar. Por ahora, un objeto es lo mismo que un valor, pero refinaremos esa definición más adelante (Sección 10.10).

La razón del error es que las cadenas son **inmutables**, lo cual significa que no puedes cambiar una cadena que ya existe. Lo mejor que puedes hacer es crear una nueva cadena que sea una variación de la original:

```
>>> saludo = 'Hola, mundo'
>>> nuevo_saludo = 'J' + saludo[1:]
>>> nuevo_saludo
'Jola, mundo'
```

Este ejemplo concatena una nueva primera letra con un corte de `saludo`. No tiene efecto en la cadena original.

8.6. Buscar

¿Qué hace la siguiente función?

```
def find(palabra, letra):
    indice = 0
    while indice < len(palabra):
        if palabra[indice] == letra:
            return indice
        indice = indice + 1
    return -1
```

En un sentido, `find` es la inversa del operador `[]`. En lugar de tomar un índice y extraer el carácter correspondiente, toma un carácter y encuentra el índice donde aparece ese carácter. Si el carácter no se encuentra, la función devuelve `-1`.

Este es el primer ejemplo que hemos visto de una sentencia `return` dentro de un bucle. Si `palabra[indice] == letra`, la función se sale del bucle y devuelve inmediatamente.

Si el carácter no aparece en la cadena, el programa termina el bucle de manera normal y devuelve `-1`.

Este patrón de computación —recorrer una secuencia y devolver cuando encontramos lo que buscamos— se llama **búsqueda**.

Como ejercicio, modifica `find` para que tenga un tercer parámetro, el índice en `palabra` donde debería comenzar a buscar.

8.7. Bucles y conteo

El siguiente programa cuenta el número de veces que aparece la letra `a` en una cadena:

```
palabra = 'banana'
contar = 0
for letra in palabra:
    if letra == 'a':
        contar = contar + 1
print(contar)
```

Este programa demuestra otro patrón de computación llamado **contador**. La variable `contar` se inicializa en 0 y luego se incrementa cada vez que se encuentra una `a`. Cuando el bucle termina, `contar` contiene el resultado: el número total de letras `a`.

Como ejercicio, encapsula este código en una función con nombre `contar` y generalízalo para que acepte la cadena y la letra como argumentos.

Luego reescribe la función de modo que, en lugar de recorrer la cadena, use la versión de tres parámetros de `find` de la sección anterior.

8.8. Métodos de cadena

Las cadenas proporcionan métodos que realizan una variedad de operaciones útiles. Un método es similar a una función —toma argumentos y devuelve un valor— pero la sintaxis es diferente. Por ejemplo, el método `upper` toma una cadena y devuelve una nueva cadena con todas las letras mayúsculas.

En lugar de la sintaxis de función `upper(palabra)`, usa la sintaxis de método `palabra.upper()`.

```
>>> palabra = 'banana'
>>> nueva_palabra = palabra.upper()
>>> nueva_palabra
'BANANA'
```

Esta forma de notación de punto especifica el nombre del método, `upper`, y el nombre de la cadena a la cual se le aplica el método, `palabra`. Los paréntesis vacíos indican que este método no toma argumentos.

Una llamada a un método se llama **invocación**; en este caso, diríamos que estamos invocando a `upper` en `palabra`.

Resulta que hay un método de cadena con nombre `find` que es notablemente similar a la función que escribimos:

```
>>> palabra = 'banana'
>>> indice = palabra.find('a')
>>> indice
1
```

En este ejemplo, invocamos a `find` en `palabra` y pasamos como parámetro la letra que buscamos.

En realidad, el método `find` es más general que nuestra función; puede encontrar subcadenas, no solo caracteres:

```
>>> palabra.find('na')
2
```

Por defecto, `find` comienza al principio de la cadena, pero puede tomar un segundo argumento, el índice donde debería comenzar:

```
>>> palabra.find('na', 3)
4
```

Este es un ejemplo de **argumento opcional**; `find` puede tomar también un tercer argumento, el índice donde debería detenerse:

```
>>> nombre = 'bob'
>>> nombre.find('b', 1, 2)
-1
```

Esta búsqueda falla porque `b` no aparece en el rango de índices desde `1` hasta `2`, sin incluir el `2`. Buscar hasta el segundo índice, sin incluirlo, hace a `find` consistente con el operador de corte.

8.9. El operador `in`

La palabra `in` es un operador booleano que toma dos cadenas y devuelve `True` si la primera aparece como una subcadena en la segunda:

```
>>> 'a' in 'banana'
True
>>> 'semilla' in 'banana'
False
```

Por ejemplo, la siguiente función imprime todas las letras de `palabra1` que también aparecen en `palabra2`:

```
def en_ambas(palabra1, palabra2):
    for letra in palabra1:
        if letra in palabra2:
            print(letra)
```

Con nombres de variables bien escogidos, Python a veces se lee como el inglés. Podrías leer este bucle, "*para* (cada) letra *en* (la primera) palabra, *si* (la) letra (aparece) *en* (la segunda) palabra, *imprimir* (la) letra."

Esto es lo que obtienes si comparas uvas y tunas:

```
>>> en_ambas('uvas', 'tunas')
u
a
s
```

8.10. Comparación de cadenas

Los operadores relacionales funcionan en las cadenas. Para ver si dos cadenas son iguales:

```
if palabra == 'banana':
    print('Todo bien, bananas.')
```

Otras operaciones relacionales son útiles para poner palabras en orden alfabético:

```
if palabra < 'banana':
    print('Tu palabra, ' + palabra + ', viene antes de banana.')
elif palabra > 'banana':
    print('Tu palabra, ' + palabra + ', viene después de banana.')
else:
    print('Todo bien, bananas.')
```

Python no maneja letras mayúsculas y minúsculas de la misma manera en que lo hacen las personas. Todas las letras mayúsculas vienen antes de todas las letras minúsculas, entonces:

```
Tu palabra, Piña, viene antes de banana.
```

Una manera común de abordar este problema es convertir las cadenas a un formato estándar, por ejemplo todas minúsculas, antes de realizar la comparación. Ten eso en mente en caso de que tengas que defenderte de un hombre armado con una Piña.

8.11. Depuración

Cuando usas índices para recorrer los valores en una secuencia, es difícil obtener el comienzo y final de un recorrido de manera correcta. Aquí hay una función que se supone que compara dos palabras y devuelve `True` si una de las palabras es el reverso de la otra, pero contiene dos errores:

```
def es_reverso(palabra1, palabra2):
    if len(palabra1) != len(palabra2):
        return False

    i = 0
    j = len(palabra2)

    while j > 0:
        if palabra1[i] != palabra2[j]:
```

```
            return False
        i = i+1
        j = j-1

    return True
```

La primera sentencia `if` verifica si las palabras tienen la misma longitud. Si no, podemos devolver `False` inmediatamente. De lo contrario, para el resto de la función, podemos suponer que las palabras tienen la misma longitud. Este es un ejemplo del patrón guardián de la Sección 6.8.

`i` y `j` son índices: `i` recorre a `palabra1` hacia adelante mientras `j` recorre a `palabra2` hacia atrás. Si encontramos dos letras que no coinciden, podemos devolver `False` inmediatamente. Si terminamos todo el bucle y todas las letras coinciden, devolvemos `True`.

Si probamos esta función con las palabras "pots" y "stop", esperamos el valor de retorno `True`, pero obtenemos un IndexError:

```
>>> es_reverso('pots', 'stop')
...
  File "reverso.py", line 15, in es_reverso
    if palabra1[i] != palabra2[j]:
IndexError: string index out of range
```

Para depurar este tipo de error, mi primer movimiento es imprimir los valores de los índices inmediatamente antes de la línea donde aparece el error.

```
    while j > 0:
        print(i, j)        # imprimir aquí

        if palabra1[i] != palabra2[j]:
            return False
        i = i+1
        j = j-1
```

Ahora cuando ejecuto el programa de nuevo, obtengo más información:

```
>>> es_reverso('pots', 'stop')
0 4
...
IndexError: string index out of range
```

En el primer paso por el bucle, el valor de `j` es 4, lo cual está fuera de rango para la cadena `'pots'`. El índice del último carácter es 3, por lo que el valor inicial para `j` debería ser `len(palabra2)-1`.

Si arreglo este error y ejecuto el programa de nuevo, obtengo:

```
>>> es_reverso('pots', 'stop')
0 3
1 2
2 1
True
```

Esta vez obtenemos la respuesta correcta, pero se ve como si el bucle solo se ejecutara tres veces, lo cual es sospechoso. Para obtener una mejor idea de lo que está ocurriendo, es útil dibujar un diagrama de estado. Durante la primera iteración, el marco para `es_reverso` se muestra en la Figura 8.2.

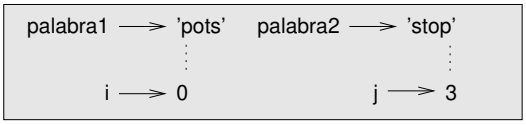

Figura 8.2: Diagrama de estado.

Me tomé la licencia de organizar las variables en el marco y agregar líneas punteadas para mostrar que los valores de `i` y `j` indican caracteres en `palabra1` y `palabra2`.

Comenzando con este diagrama, ejecuta el programa en papel, cambiando los valores de `i` y `j` durante cada iteración. Encuentra y arregla el segundo error en esta función.

8.12. Glosario

objeto: Algo a lo cual una variable puede referirse. Por ahora, puedes usar "objeto" y "valor" indistintamente.

secuencia: Una colección ordenada de valores donde cada valor se identifica por un índice entero.

ítem: Uno de los valores en una secuencia.

índice: Un valor entero usado para seleccionar un ítem en una secuencia, tal como un carácter en una cadena. En Python, los índices parten desde 0.

corte (*slice*): Una parte de una cadena especificada por un rango de índices.

cadena vacía: Una cadena sin caracteres y con longitud 0, representada por dos comillas.

inmutable: La propiedad de una secuencia cuyos ítems no pueden cambiarse.

recorrer: Iterar a través de los ítems en una secuencia, realizando una operación similar en cada uno de estos.

búsqueda: Un patrón de un recorrido que se detiene cuando encuentra lo que busca.

contador: Una variable usada para contar algo, usualmente inicializada en cero y luego incrementada.

invocación: Una sentencia que llama a un método.

argumento opcional: Un argumento de función o de método que no es obligatorio.

8.13. Ejercicios

Ejercicio 8.1. *Lee la documentación de los métodos de cadena en* `http://docs.python.org/3/library/stdtypes.html#string-methods`. *Tal vez quieras experimentar con algunos para asegurarte de que entiendes cómo funcionan.* `strip` *y* `replace` *son particularmente útiles.*

La documentación usa una sintaxis que podría confundir. Por ejemplo, en `find(sub[, start[, end]])`*, los corchetes indican argumentos opcionales. Entonces* `sub` *es obligatorio, pero* `start` *es opcional, y si incluyes* `start`*, entonces* `end` *es opcional.*

Ejercicio 8.2. *Hay un método de cadena llamado* `count` *que es similar a la función de la Sección 8.7. Lee la documentación de este método y escribe una invocación que cuente el número de letras* `a` *en* `'banana'`.

Ejercicio 8.3. *Un corte de cadena puede tomar un tercer índice que especifique el "tamaño de paso"; es decir, el número de espacios entre caracteres sucesivos. Un tamaño de paso de 2 significa cada dos caracteres; 3 significa cada tres, etc.*

```
>>> fruta = 'banana'
>>> fruta[0:5:2]
'bnn'
```

Un tamaño de paso de -1 pasa a través de la palabra hacia atrás, por lo que el corte `[::-1]` *genera una cadena invertida.*

Usa esta notación para escribir una versión de una línea de `es_palindromo` *del Ejercicio 6.3.*

Ejercicio 8.4. *Las siguientes funciones tienen la* intención *de verificar si una cadena contiene al menos una letra minúscula, pero algunas están equivocadas. Para cada función, describe qué hace realmente la función (suponiendo que el parámetro es una cadena).*

```
def contiene_minuscula1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def contiene_minuscula2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def contiene_minuscula3(s):
    for c in s:
        flag = c.islower()
    return flag

def contiene_minuscula4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def contiene_minuscula5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Ejercicio 8.5. *Un cifrado César es una forma débil de encriptación que implica la "rotación" de cada letra en un número fijo de lugares. Rotar una letra significa desplazarla a través del alfabeto, volviendo al comienzo si es necesario, por lo que 'A' rotada en 3 es 'D' y 'Z' rotada en 1 es 'A'.*

Para rotar una palabra, rota cada letra en la misma cantidad. Por ejemplo, "cheer" rotada en 7 es "jolly" y "melon" rotada en -10 es "cubed". En la película 2001: Odisea del espacio, *el computador de la nave se llama HAL, que es IBM rotada en -1.*

Escribe una función llamada `rotar_palabra` *que tome una cadena y un entero como parámetros y devuelva una nueva cadena que contenga las letras de la cadena original rotadas en la cantidad entregada.*

Tal vez quieras usasr la función incorporada `ord`*, que convierte un carácter en un código numérico, y* `chr`*, que convierte códigos numéricos en caracteres. Las letras del alfabeto están codificadas en orden alfabético, así por ejemplo:*

```
>>> ord('c') - ord('a')
2
```

Debido a que `'c'` *es la dos-ésima letra del alfabeto. Pero ten cuidado: los códigos numéricos para las letras mayúsculas son diferentes.*

Los chistes potencialmente ofensivos en internet a veces están codificados en ROT13, que es un cifrado César con rotación 13. Si no te ofendes fácilmente, encuentra y decodifica algunos. Solución: `http://thinkpython2.com/code/rotate.py`.

Capítulo 9

Estudio de caso: juego de palabras

Este capítulo presenta el segundo estudio de caso, el cual involucra resolver puzles de palabras buscando palabras que tengan ciertas propiedades. Por ejemplo, encontraremos los palíndromos más largos en inglés y buscaremos palabras cuyas letras aparezcan en orden alfabético. Y presentaré otro plan de desarrollo de programa: reducción a un problema previamente resuelto.

9.1. Leer listas de palabras

Para los ejercicios de este capítulo necesitamos una lista de palabras en inglés. Hay muchas listas de palabras disponibles en la web, pero la más adecuada para nuestro propósito es una de las listas de palabras recopiladas y contribuidas al dominio público por Grady Ward como parte del proyecto léxico Moby (ver `http://wikipedia.org/wiki/Moby_Project`). Es una lista de 113,809 palabras de crucigrama oficiales; es decir, palabras que se consideran válidas en crucigramas y otros juegos de palabras. En la colección Moby, el nombre del archivo es `113809of.fic`; puedes descargar una copia, con el nombre más simple `words.txt`, en `http://thinkpython2.com/code/words.txt`.

Este archivo está en texto plano, por lo que puedes abrirlo con un editor de texto, pero también puedes leerlo desde Python. La función incorporada `open` toma el nombre del archivo como parámetro y devuelve un **objeto de archivo** que puedes usar para leer el archivo.

```
>>> fin = open('words.txt')
```

`fin` es un nombre común para un objeto de archivo usado para la entrada (*file input*). El objeto de archivo proporciona varios métodos para la lectura, incluyendo `readline`, que lee caracteres desde un archivo hasta que llega a una nueva línea y devuelve el resultado como una cadena:

```
>>> fin.readline()
'aa\n'
```

La primera palabra en esta lista particular es "aa", que es un tipo de lava. La secuencia `\n` representa el carácter nueva línea que separa esta palabra de la siguiente.

El objeto de archivo hace un seguimiento de dónde está en el archivo, por lo que si llamas a `readline` de nuevo, obtienes la palabra siguiente:

```
>>> fin.readline()
'aah\n'
```

La palabra siguiente es "aah", que es una palabra perfectamente legítima, así que deja de mirarme así. O bien, si es el carácter nueva línea lo que te molesta, podemos deshacernos de este con el método de cadena `strip`:

```
>>> linea = fin.readline()
>>> palabra = linea.strip()
>>> palabra
'aahed'
```

Puedes usar también un objeto de archivo como parte de un bucle `for`. Este programa lee `words.txt` e imprime cada palabra, una por línea:

```
fin = open('words.txt')
for linea in fin:
    palabra = linea.strip()
    print(palabra)
```

9.2. Ejercicios

Hay soluciones a estos ejercicios en la siguiente sección. Deberías al menos intentar cada uno antes de leer las soluciones.

Ejercicio 9.1. *Escribe un programa que lea* `words.txt` *e imprima solo las palabras con más de 20 caracteres (sin contar espacios en blanco).*

Ejercicio 9.2. *En 1939, Ernest Vincent Wright publicó una novela de 50,000 palabras llamada* Gadsby, *la cual no contiene la letra "e". Dado que la "e" es la letra más común en el idioma inglés, eso no es fácil de hacer.*

Sin duda, solo imaginar una oración sin usar dicho símbolo más común implica una actividad difícil. Si tardas mucho al principio, hazlo con cuidado y trabaja horas para adquirir la habilidad poco a poco.

Basta, ya no sigo más.

Escribe una función llamada `no_tiene_e` *que devuelva* `True` *si la palabra dada no incluye la letra "e".*

Escribe un programa que lea `words.txt` *e imprima solo las palabras que no tienen "e". Calcula el porcentaje de palabras en la lista que no tienen "e".*

Ejercicio 9.3. *Escribe una función con nombre* `evita` *que tome una palabra y una cadena de letras prohibidas y devuelva* `True` *si la palabra no usa ninguna de las letras prohibidas.*

Escribe un programa que solicite al usuario ingresar una cadena de letras prohibidas y luego imprima el número de palabras que no contiene ninguna de estas. ¿Puedes encontrar una combinación de 5 letras prohibidas que excluya al menor número de palabras?

Ejercicio 9.4. *Escribe una función con nombre* `usa_solo` *que tome una palabra y una cadena de letras y devuelva* `True` *si la palabra contiene solo letras de la lista. ¿Puedes crear una oración en inglés usando solo las letras* `acefhlo`*? ¿Una distinta a "Hoe alfalfa"?*

Ejercicio 9.5. *Escribe una función con nombre* `usa_todas` *que tome una palabra y una cadena de letras requeridas y devuelva* `True` *si la palabra usa todas las letras requeridas al menos una vez. ¿Cuántas palabras que usan todas las vocales* `aeiou` *existen? ¿Qué hay de* `aeiouy`*?*

Ejercicio 9.6. *Escribe una función llamada* `es_abecedario` *que devuelva* `True` *si las letras en una palabra aparecen en orden alfabético (las letras dobles están permitidas). ¿Cuántas palabras abecedarias existen?*

9.3. Búsqueda

Todos los ejercicios de la sección anterior tienen algo en común; pueden ser resueltos con el patrón de búsqueda que vimos en la Sección 8.6. El ejemplo más simple es:

```
def no_tiene_e(palabra):
    for letra in palabra:
        if letra == 'e':
            return False
    return True
```

El bucle `for` recorre los caracteres en `palabra`. Si encontramos la letra "e", podemos devolver `False` inmediatamente; de lo contrario tenemos que ir a la siguiente letra. Si terminamos el bucle de manera normal, significa que no encontramos una "e", por lo cual devolvemos `True`.

Podrías haber escrito esta función de manera más concisa usando el operador `in`, pero comencé con esta versión porque demuestra la lógica del patrón de búsqueda.

`evita` es una versión más general de `no_tiene_e` pero tiene la misma estructura:

```
def evita(palabra, prohibidas):
    for letra in palabra:
        if letra in prohibidas:
            return False
    return True
```

Podemos devolver `False` apenas encontremos una letra prohibida; si llegamos al final del bucle, devolvemos `True`.

`usa_solo` es similar, excepto que el sentido de la condición es inverso:

```
def usa_solo(palabra, disponibles):
    for letra in palabra:
        if letra not in disponibles:
            return False
    return True
```

En lugar de una lista de letras prohibidas, tenemos una lista de letras disponibles. Si encontramos una letra en `palabra` que no está en `disponibles`, podemos devolver `False`.

`usa_todas` es similar, excepto que invertimos el rol de la palabra y la cadena de letras:

```
def usa_todas(palabra, requeridas):
    for letra in requeridas:
        if letra not in palabra:
            return False
    return True
```

En lugar de recorrer las letras en `palabra`, el bucle recorre las letras requeridas. Si alguna de las letras requeridas no aparece en la palabra, podemos devolver `False`.

Si realmente estuvieras pensando como un informático, habrías reconocido que `usa_todas` era una instancia de un problema previamente resuelto, y habrías escrito:

```
def usa_todas(palabra, requeridas):
    return usa_solo(requeridas, palabra)
```

Este es un ejemplo de un plan de desarrollo de programa llamado **reducción a un problema previamente resuelto**, lo cual significa que reconoces el problema en el que estás trabajando como una instancia de un problema resuelto y aplicas una solución existente.

9.4. Bucles con índices

Escribí las funciones de la sección anterior con bucles `for` porque solo necesitaba los caracteres en las cadenas; No tenía que hacer nada con los índices.

Para `es_abecedario` tenemos que comparar letras adyacentes, lo cual es un poco difícil con un bucle `for`:

```
def es_abecedario(palabra):
    anterior = palabra[0]
    for c in palabra:
        if c < anterior:
            return False
        anterior = c
    return True
```

Una alternativa es usar recursividad:

```
def es_abecedario(palabra):
    if len(palabra) <= 1:
        return True
    if palabra[0] > palabra[1]:
        return False
    return es_abecedario(palabra[1:])
```

Otra opción es usar un bucle `while`:

```
def es_abecedario(palabra):
    i = 0
    while i < len(palabra)-1:
        if palabra[i+1] < palabra[i]:
            return False
        i = i+1
    return True
```

Este bucle comienza con `i=0` y termina cuando `i=len(palabra)-1`. En cada paso por el bucle, compara al i-ésimo carácter (que puedes pensarlo como el carácter actual) con el $i+1$-ésimo carácter (que puedes pensarlo como el siguiente).

Si el siguiente carácter es menor (alfabéticamente anterior) que el actual, entonces hemos descubierto que se rompe la tendencia abecedaria y devolvemos `False`.

Si llegamos al final del bucle sin encontrar una falla, entonces la palabra pasa la prueba. Para convencerte de que el bucle termina de manera correcta, considera un ejemplo como `'flossy'`. La longitud de la palabra es 6, por lo que la última vez que el bucle se ejecuta es cuando `i` es 4, que es el índice del penúltimo carácter. En la última iteración, compara el penúltimo carácter con el último, que es lo que queremos.

Aquí hay una versión de `es_palindromo` (ver Ejercicio 6.3) que usa dos índices; uno comienza al principio y aumenta; el otro comienza al final y disminuye.

```
def es_palindromo(palabra):
    i = 0
    j = len(palabra)-1

    while i<j:
        if palabra[i] != palabra[j]:
```

```
            return False
        i = i+1
        j = j-1

    return True
```

O bien podríamos reducir a un problema previamente resuelto y escribir:

```
def es_palindromo(palabra):
    return es_reverso(palabra, palabra)
```

Usando `es_reverso` de la Sección 8.11.

9.5. Depuración

Probar programas es difícil. Las funciones de este capítulo son relativamente fáciles de probar porque puedes verificar los resultados a mano. Aún así, escoger un conjunto de palabras que pruebe todos los errores posibles está en algún lugar entre difícil e imposible.

Tomando a `no_tiene_e` como ejemplo, hay dos casos obvios para verificar: las palabras que tienen una 'e' deberían devolver `False` y las palabras que no la tienen deberían devolver `True`. No deberías tener problemas para proponer una palabra de cada caso.

Dentro de cada caso, hay algunos subcasos menos obvios. Entre las palabras que tienen una "e", deberías probar palabras con una "e" al principio, al final y en algún lugar del medio. Deberías probar palabras largas, palabras cortas y palabras muy cortas, como la cadena vacía. La cadena vacía es un ejemplo de un **caso especial**, que es uno de los casos no obvios donde los errores a menudo acechan.

Además de los casos de prueba que generes, puedes también probar tu programa con una lista de palabras como `words.txt`. Escudriñando la salida, podrías ser capaz de captar los errores, pero ten cuidado: podrías captar un tipo de error (palabras que no deberían estar incluidas, pero lo están) y no otro (palabras que deberían estar incluidas, pero no lo están).

En general, las pruebas pueden ayudarte a encontrar errores, pero no es fácil generar un buen conjunto de casos de prueba, e incluso si lo haces, no puedes estar seguro de que tu programa está correcto. De acuerdo al legendario informático:

> Se pueden probar programas para mostrar la presencia de errores, ¡pero nunca para mostrar su ausencia!
>
> — Edsger W. Dijkstra

9.6. Glosario

objeto de archivo: Un valor que representa un archivo abierto.

reducción a un problema previamente resuelto: Una manera de resolver un problema expresándolo como una instancia de un problema previamente resuelto.

caso especial: Un caso de prueba que es atípico o no obvio (y menos probable de abordar correctamente).

9.7. Ejercicios

Ejercicio 9.7. *Esta pregunta está basada en un Puzzler que fue transmitido en el programa de radio* Car Talk *(http://www.cartalk.com/content/puzzlers):*

> *Dame una palabra con tres letras dobles consecutivas. Te daré un par de palabras que casi califican, pero no. Por ejemplo, la palabra committee, c-o-m-m-i-t-t-e-e. Estaría excelente, si no fuera por la 'i' que se cuela allí. O Mississippi: M-i-s-s-i-s-s-i-p-p-i. Si pudieras sacar esas 'i' funcionaría. Pero hay una palabra que tiene tres pares de letras consecutivos y, por lo que sé, puede ser la única palabra. Por supuesto que probablemente hay 500 más pero solo puedo pensar en una. ¿Cuál es la palabra?*

Escribe un programa que la encuentre. Solución: `http://thinkpython2.com/code/cartalk1.py`.

Ejercicio 9.8. *Aquí hay otro Puzzler de* Car Talk *(http://www.cartalk.com/content/puzzlers):*

> *"Estaba conduciendo en la carretera el otro día y me fijé en mi odómetro. Al igual que la mayoría de los odómetros, muestra seis dígitos, solo en millas enteras. Entonces, si mi automóvil tenía 300,000 millas, por ejemplo, veía 3-0-0-0-0-0.*
>
> *"Ahora, lo que vi ese día fue muy interesante. Me di cuenta de que los últimos 4 dígitos eran palíndromo; es decir, se leían igual hacia adelante y hacia atrás. Por ejemplo, 5-4-4-5 es un palíndromo, por lo que mi odómetro podría haber leído 3-1-5-4-4-5.*
>
> *"Una milla más adelante, los últimos 5 números eran palíndromo. Por ejemplo, podría haber leído 3-6-5-4-5-6. Una milla después de eso, los 4 números que están al medio de los 6 eran palíndromo. ¿Y estás listo para esto? Una milla más adelante, ¡los 6 eran palíndromo!*
>
> *"La pregunta es, ¿qué había en mi odómetro cuando miré por primera vez?"*

Escribe un programa en Python que pruebe todos los números de seis dígitos e imprima aquellos números que satisfagan estos requisitos. Solución: `http://thinkpython2.com/code/cartalk2.py`.

Ejercicio 9.9. *Aquí hay otro Puzzler de* Car Talk *que puedes resolver con una búsqueda (http://www.cartalk.com/content/puzzlers):*

> *"Recientemente tuve una visita con mi mamá y me di cuenta de que los dos dígitos que componen mi edad cuando se invierten resulta en su edad. Por ejemplo, si ella tiene 73, yo tengo 37. Nos preguntamos cuán a menudo ha ocurrido esto a través de los años pero nos desviamos a otros temas y nunca dimos con una respuesta.*
>
> *"Cuando llegué a casa descubrí que los dígitos de nuestras edades han sido reversibles seis veces hasta ahora. También descubrí que, si tenemos suerte, ocurriría de nuevo en unos años, y si realmente tenemos suerte ocurriría una vez más después de eso. En otras palabras, habría ocurrido 8 veces en total. Entonces la pregunta es, ¿qué edad tengo ahora?"*

Escribe un programa en Python que busque soluciones a este Puzzler. Pista: podrías encontrar útil el método de cadena `zfill`.

Solución: `http://thinkpython2.com/code/cartalk3.py`.

Capítulo 10

Listas

Este capítulo presenta uno de los tipos incorporados más útiles de Python: las listas. También aprenderás más sobre objetos y lo que puede ocurrir cuando tienes más de un nombre para el mismo objeto.

10.1. Una lista es una secuencia

Al igual que una cadena, una **lista** es una secuencia de valores. En una cadena, los valores son caracteres; en una lista, pueden ser de cualquier tipo. Los valores en una lista se llaman **elementos** o a veces **ítems**.

Hay varias maneras de crear una nueva lista; la más simple es encerrar los elementos en corchetes (`[` y `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

El primer ejemplo es una lista de cuatro enteros. El segundo es una lista de tres cadenas. Los elementos de una lista no tienen que ser del mismo tipo. La siguiente lista contiene una cadena, un número de coma flotante, un entero y (¡atención!) otra lista:

```
['spam', 2.0, 5, [10, 20]]
```

Una lista dentro de otra lista está **anidada**.

Una lista que no contiene elementos se llama lista vacía; puedes crear una con corchetes vacíos, `[]`.

Como podrías esperar, puedes asignar valores de lista a variables:

```
>>> quesos = ['Cheddar', 'Edam', 'Gouda']
>>> numeros = [42, 123]
>>> vacio = []
>>> print(quesos, numeros, vacio)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

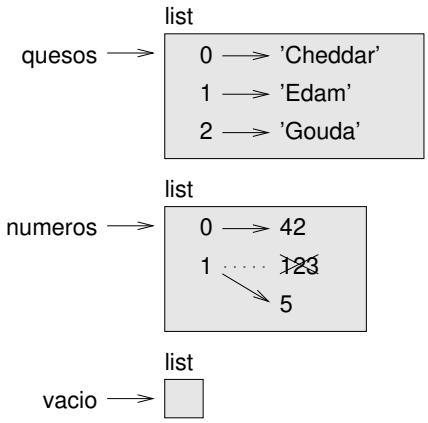

Figura 10.1: Diagrama de estado.

10.2. Las listas son mutables

La sintaxis para acceder a los elementos de una lista es la misma que para acceder a los caracteres de una cadena: el operador de corchetes. La expresión dentro de los corchetes especifica el índice. Recuerda que los índices comienzan en 0:

```
>>> quesos[0]
'Cheddar'
```

A diferencia de las cadenas, las listas son mutables. Cuando el operador de corchetes aparece en el lado izquierdo de una asignación, este identifica el elemento de la lista que será asignado.

```
>>> numeros = [42, 123]
>>> numeros[1] = 5
>>> numeros
[42, 5]
```

El uno-ésimo elemento de `numeros`, que solía ser 123, ahora es 5.

La Figura 10.1 muestra el diagrama de estado para `quesos`, `numeros` y `vacio`.

Las listas se representan por cajas con la palabra "list" por fuera y los elementos de la lista por dentro. `quesos` se refiere a una lista con tres elementos con índices 0, 1 y 2. `numeros` contiene dos elementos; el diagrama muestra que el valor del segundo elemento ha sido reasignado de 123 a 5. `vacio` se refiere a una lista sin elementos.

Los índices de las listas funcionan de la misma manera que los índices de las cadenas:

- Cualquier expresión entera puede usarse como índice.
- Si intentas leer o escribir un elemento que no existe, obtienes un `IndexError`.
- Si un índice tiene un valor negativo, se cuenta hacia atrás desde el final de la lista.

El operador `in` también funciona en las listas.

```
>>> quesos = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in quesos
True
>>> 'Brie' in quesos
False
```

10.3. Recorrer una lista

La manera más común de recorrer los elementos de una lista es con un ciclo `for`. La sintaxis es la misma que para las cadenas:

```
for queso in quesos:
    print(queso)
```

Esto funciona bien si solo necesitas leer los elementos de la lista. Pero si quieres escribir o actualizar los elementos, necesitas los índices. Una manera común de hacer eso es combinar las funciones incorporadas `range` y `len`:

```
for i in range(len(numeros)):
    numeros[i] = numeros[i] * 2
```

Este bucle recorre la lista y actualiza cada elemento. `len` devuelve el número de elementos en la lista. `range` devuelve una lista de índices de 0 a $n - 1$, donde n es el largo de la lista. En cada paso por el bucle, `i` obtiene el índice del siguiente elemento. La sentencia de asignación en el cuerpo usa `i` para leer el valor antiguo del elemento y asignar el nuevo valor.

Un bucle `for` a través de una lista vacía nunca ejecuta el cuerpo:

```
for x in []:
    print('Esto nunca ocurre.')
```

A pesar de que una lista puede contener otra lista, la lista anidada aún cuenta como un solo elemento. La longitud de esta lista es cuatro:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

10.4. Operaciones de lista

El operador + concatena listas:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

El operador `*` repite una lista un número dado de veces:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

El primer ejemplo repite `[0]` cuatro veces. El segundo ejemplo repite la lista `[1, 2, 3]` tres veces.

10.5. Cortes de lista

El operador de corte también funciona en las listas:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Si omites el primer índice, el corte comienza al principio. Si omites el segundo, el corte llega al final. Entonces, si omites ambos, el corte es una copia de la lista completa.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Dado que las listas son mutables, a menudo es útil crear una copia antes de realizar operacioines que modifiquen listas.

Un operador de corte en el lado izquierdo de una asignación puede actualizar múltiples elementos:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

10.6. Métodos de lista

Python proporciona métodos que operan en listas. Por ejemplo, `append` agrega un nuevo elemento al final de la lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

`extend` toma una lista como argumento y anexa todos los elementos:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

Este ejemplo deja a `t2` sin modificar.

`sort` ordena los ejementos de la lista de menor a mayor:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

La mayoría de los métodos de lista son nulos; modifican la lista y devuelven `None`. Si por casualidad escribes `t = t.sort()`, te decepcionará el resultado.

10.7. Mapa, filtro y reducción

Para sumar todos los números de una lista, puedes usar un bucle como este:

```
def sumar_todos(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` se inicializa en 0. En cada paso por el bucle, `x` obtiene un elemento de la lista. El operador `+=` proporciona una manera corta de actualizar una variable. Esta **sentencia de asignación aumentada**,

```
    total += x
```

es equivalente a

```
    total = total + x
```

A medida que el bucle se ejecuta, `total` acumula la suma de los elementos; una variable que se usa de esta manera a veces es llamada **acumulador**.

Sumar los elementos de una lista es una operación tan común que Python la facilita como función incorporada, `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Una operación como esta que combina una secuencia de elementos en un solo valor a veces es llamada **reducción**.

A veces quieres recorrer una lista mientras construyes otra. Por ejemplo, la siguiente función toma una lista de cadenas y devuelve una nueva lista que contiene cadenas que comienzan con mayúscula:

```
def todas_con_mayuscula(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` se inicializa con una lista vacía; en cada paso por el bucle, anexamos el elemento siguiente. Entonces `res` es otro tipo de acumulador.

Una operación como `todas_con_mayuscula` a veces es llamada **mapa** porque "mapea" una función (en este caso el método `capitalize`) sobre cada uno de los elementos en una secuencia.

Otra operación común es seleccionar algunos de los elementos de una lista y devolver una sublista. Por ejemplo, la siguiente función toma una lista de cadenas y devuelve una lista que contiene solo las cadenas escritas con mayúsculas:

```
def solo_mayusculas(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` es un método de cadena que devuelve `True` si la cadena solo contiene letras mayúsculas.

Una operación como `solo_mayusculas` se llama **filtro** porque selecciona algunos de los elementos y filtra los otros.

La mayoría de las operaciones de lista se pueden expresar como una combinación de mapa, filtro y reducción.

10.8. Eliminar elementos

Hay varias maneras de eliminar elementos de una lista. Si conoces el índice del elemento que quieres, puedes usar `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

`pop` modifica la lista y devuelve el elemento que se eliminó. Si no entregas un índice, elimina y devuelve el último elemento.

Si no necesitas el valor eliminado, puedes usar el operador `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

Si conoces el elemento que quieres eliminar (pero no el índice), puedes usar `remove`:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
['a', 'c']
```

El valor de retorno de `remove` es `None`.

Para eliminar más de un elemento, puedes usar `del` con índices de corte:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

Como siempre, el corte selecciona todos los elementos hasta el segundo índice pero sin incluirlo.

10.9. Listas y cadenas

Una cadena es una secuencia de caracteres y una lista es una secuencia de valores, pero una lista de caracteres no es lo mismo que una cadena. Para convertir una cadena en una lista de caracteres, puedes usar `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Dado que `list` es el nombre de una función incorporada, deberías evitar usarlo como nombre de variable. Yo además evito `l` porque se parece mucho a `1`. Entonces por eso uso `t`.

La función `list` separa la cadena en letras individuales. Si quieres separar una cadena en palabras, puedes usar el método `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

Un argumento opcional llamado **delimitador** especifica qué caracteres usar como separador de palabras. El siguiente ejemplo usa un guión como delimitador:

```
>>> s = 'spam-spam-spam'
>>> delimitador = '-'
>>> t = s.split(delimitador)
>>> t
['spam', 'spam', 'spam']
```

`join` es el inverso de `split`. Toma una lista de cadenas y concatena los elementos. `join` es un método de cadena, por lo que tienes que invocarlo en el delimitador y pasarle la lista como parámetro:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimitador = ' '
>>> s = delimitador.join(t)
>>> s
'pining for the fjords'
```

En este caso el delimitador es un carácter de espacio, por lo que `join` pone un espacio entre las palabras. Para concatenar cadenas sin espacios, puedes usar la cadena vacía, `''`, como delimitador.

10.10. Objetos y valores

Si ejecutamos estas sentencias de asignación:

```
a = 'banana'
b = 'banana'
```

Sabemos que `a` y `b` se refieren a una cadena, pero no sabemos si se refieren a la *misma* cadena. Hay dos estados posibles, mostrados en la Figura 10.2.

En el primer caso, `a` y `b` se refieren a dos objetos diferentes que tienen el mismo valor. En el segundo caso, se refieren al mismo objeto.

Para verificar si dos variables se refieren al mismo objeto, puedes usar el operador `is`.

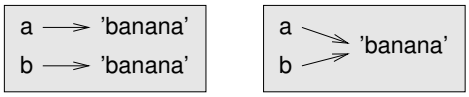

Figura 10.2: Diagrama de estado.

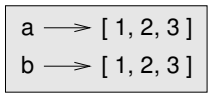

Figura 10.3: Diagrama de estado.

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

En este ejemplo, Python solo crea un objeto de cadena y tanto `a` como `b` se refieren a este. Pero cuando creas dos listas, obtienes dos objetos:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Entonces el diagrama de estado se ve como la Figura 10.3.

En este caso diríamos que las dos listas son **equivalentes**, porque tienen los mismos elementos, pero no **idénticos**, porque no son el mismo objeto. Si dos objetos son idénticos, son también equivalentes, pero si son equivalentes, no necesariamente son idénticos.

Hasta ahora, hemos estado usando "objeto" y "valor" indistintamente, pero es más preciso decir que un objeto tiene un valor. Si evalúas `[1, 2, 3]`, obtienes un objeto de lista cuyo valor es una secuencia de enteros. Si otra lista tiene los mismos elementos, decimos que tiene el mismo valor, pero no es el mismo objeto.

10.11. Alias

Si `a` se refiere a un objeto y asignas `b = a`, entonces ambas variables se refieren al mismo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

El diagrama de estado se ve como la Figura 10.4.

La asociación de una variable con un objeto se llama **referencia**. En este ejemplo, hay dos referencias al mismo objeto.

Un objeto con más de una referencia tiene más de un nombre, por lo que decimos que el objeto tiene **alias**.

Si el objeto con alias es mutable, los cambios realizados con un alias afectan al otro:

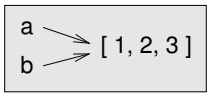

Figura 10.4: Diagrama de estado.

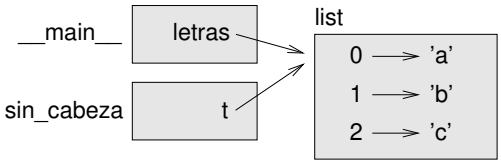

Figura 10.5: Diagrama de pila.

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

Aunque este comportamiento puede ser útil, es propenso a errores. En general, es más seguro evitar los alias cuando trabajes con objetos mutables.

Para los objetos inmutables como las cadenas, los alias no son tan problemáticos. En este ejemplo:

```
a = 'banana'
b = 'banana'
```

Casi nunca hace diferencia si `a` y `b` se refieren a la misma cadena o no.

10.12. Argumentos de lista

Cuando pasas una lista a una función, la función obtiene una referencia a la lista. Si la función modifica la lista, la sentencia llamadora ve el cambio. Por ejemplo, `sin_cabeza` quita el primer elemento de una lista:

```
def sin_cabeza(t):
    del t[0]
```

Se usa de la siguiente manera:

```
>>> letras = ['a', 'b', 'c']
>>> sin_cabeza(letras)
>>> letras
['b', 'c']
```

El parámetro `t` y la variable `letras` son alias para el mismo objeto. El diagrama de pila se ve como la Figura 10.5.

Dado que la lista es compartida por dos marcos, la dibujé entre estos.

Es importante distinguir entre operaciones que modifican listas y operaciones que crean nuevas listas. Por ejemplo, el método `append` modifica una lista, pero el operador `+` crea una nueva lista.

Aquí hay un ejemplo que usa `append`:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

El valor de retorno de `append` es `None`.

Aquí hay un ejemplo que usa el operador `+`:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

El resultado del operador es una nueva lista y la lista original no ha cambiado.

Esta diferencia es importante cuando escribes funciones que se supone que modifican listas. Por ejemplo, esta función *no* elimina la cabeza de una lista:

```
def sin_cabeza_mal(t):
    t = t[1:]              # ¡INCORRECTO!
```

El operador de corte crea una nueva lista y la asignación hace que `t` se refiera a esta, pero eso no afecta a la llamadora.

```
>>> t4 = [1, 2, 3]
>>> sin_cabeza_mal(t4)
>>> t4
[1, 2, 3]
```

Al principio de `sin_cabeza_mal`, `t` y `t4` se refieren a la misma lista. Al final, `t` se refiere a una nueva lista, pero `t4` aún se refiere a la original, la lista sin modificar.

Una alternativa es escribir una función que cree y devuelva una nueva lista. Por ejemplo, `cola` devuelve todos los elementos de una lista excepto el primero:

```
def cola(t):
    return t[1:]
```

Esta función deja a la lista original sin modificar. Se usa de la siguiente manera:

```
>>> letra = ['a', 'b', 'c']
>>> resto = cola(letras)
>>> resto
['b', 'c']
```

10.13. Depuración

El uso descuidado de las listas (y otros objetos mutables) puede llevar a largas horas de depuración. Aquí hay algunas trampas comunes y maneras de evitarlas:

1. La mayoría de los métodos de lista modifican el argumento y devuelven `None`. Esto es lo opuesto a los métodos de cadena, que devuelven una nueva cadena y dejan sola a la original.

   Si te acostumbraste a escribir código de cadena como este:

   ```
   palabra = palabra.strip()
   ```

   Es tentador escribir código de lista como este:

   ```
   t = t.sort()            # ¡INCORRECTO!
   ```

   Dado que `sort` devuelve `None`, es probable que la siguiente operación que realices con `t` falle.

   Antes de usar métodos y operadores de lista, deberías leer la documentación cuidadosamente y luego probarlos en modo interactivo.

2. Escoge una forma y quédate con esa.

   Parte del problema con las listas es que hay muchas maneras de hacer las cosas. Por ejemplo, para eliminar un elemento de una lista, puedes usar `pop`, `remove`, `del`, o incluso una asignación de corte.

   Para agregar un elemento, puedes usar el método `append` o el operador `+`. Suponiendo que `t` es una lista y `x` es un elemento de lista, estas líneas son correctas:

   ```
   t.append(x)
   t = t + [x]
   t += [x]
   ```

   Y estas son incorrectas:

   ```
   t.append([x])           # ¡INCORRECTO!
   t = t.append(x)         # ¡INCORRECTO!
   t + [x]                 # ¡INCORRECTO!
   t = t + x               # ¡INCORRECTO!
   ```

   Prueba cada uno de estos ejemplos en modo interactivo para asegurarte de que entiendes lo que haces. Nota que solo el último provoca un error de tiempo de ejecución; los otros tres son legales, pero hacen lo incorrecto.

3. Crea copias para evitar los alias.

   Si quieres usar un método como `sort` que modifique el argumento, pero necesitas mantener la lista original también, puedes crear una copia.

   ```
   >>> t = [3, 1, 2]
   >>> t2 = t[:]
   >>> t2.sort()
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

   En este ejemplo podrías usar también la función incorporada `sorted`, que devuelve una nueva lista ordenada y deja sola a la original.

   ```
   >>> t2 = sorted(t)
   >>> t
   [3, 1, 2]
   >>> t2
   [1, 2, 3]
   ```

10.14. Glosario

lista: Una secuencia de valores.

elemento: Uno de los valores en una lista (u otra secuencia), también llamados ítems.

lista anidada: Una lista que es un elemento de otra lista.

acumulador: Una variable usada en un bucle para sumar o acumular un resultado.

asignación aumentada: Una sentencia que actualiza el valor de una variable usando un operador como +=.

reducción: Un patrón de procesamiento que recorre una secuencia y acumula los elementos en un solo resultado.

mapa: Un patrón de procesamiento que recorre una secuencia y realiza una operación en cada elemento.

filtro: Un patrón de procesamiento que recorre una lista y selecciona los elementos que satisfacen algún criterio.

objeto: Algo a lo cual una variable puede referirse. Un objeto tiene un tipo y un valor.

equivalente: Que tiene el mismo valor.

idéntico: Que es el mismo objeto (lo cual implica equivalencia).

referencia: La asociación entre una variable y su valor.

alias: Una circunstancia donde dos o más variables se refieren al mismo objeto.

delimitador: Un carácter o cadena usado para indicar dónde debería separarse una cadena.

10.15. Ejercicios

Puedes descargar las soluciones a estos ejercicios en `http://thinkpython2.com/code/list_exercises.py`.

Ejercicio 10.1. *Escribe una función llamada* `suma_anidada` *que tome una lista de listas de enteros y sume los elementos de todas las listas anidadas. Por ejemplo:*

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> suma_anidada(t)
21
```

Ejercicio 10.2. *Escribe una función llamada* `cumsum` *que tome una lista de números y devuelva la suma acumulativa; es decir, una nueva lista donde el i-ésimo elemento es la suma de los primeros* $i + 1$ *elementos de la lista original. Por ejemplo:*

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

Ejercicio 10.3. *Escribe una función llamada* `medio` *que tome una lista y devuelva una nueva lista que contenga todos los elementos excepto el primero y el último. Por ejemplo:*

```
>>> t = [1, 2, 3, 4]
>>> medio(t)
[2, 3]
```

Ejercicio 10.4. *Escribe una función llamada* `acortar` *que tome una lista, la modifique eliminando el primer y último elemento, y devuelva* `None`*. Por ejemplo:*

```
>>> t = [1, 2, 3, 4]
>>> acortar(t)
>>> t
[2, 3]
```

Ejercicio 10.5. *Escribe una función llamada* `esta_ordenada` *que tome una lista como parámetro y devuelva* `True` *si la lista está ordenada de manera ascendente y* `False` *si no. Por ejemplo:*

```
>>> esta_ordenada([1, 2, 2])
True
>>> esta_ordenada(['b', 'a'])
False
```

Ejercicio 10.6. *Dos palabras son anagramas si puedes reordenar las letras de una para escribir la otra. Escribe una función llamada* `es_anagrama` *que tome dos cadenas y devuelva* `True` *si son anagramas.*

Ejercicio 10.7. *Escribe una función llamada* `tiene_duplicados` *que tome una lista y devuelva* `True` *si hay algún elemento que aparece más de una vez. No debería modificar la lista original.*

Ejercicio 10.8. *Este ejercicio está relacionado con la denominada "Paradoja del cumpleaños", de la cual puedes leer en* `http://es.wikipedia.org/wiki/Paradoja_del_cumplea~nos`*.*

Si hay 23 estudiantes en tu clase, ¿cuáles son las posibilidades de que dos de ustedes estén de cumpleaños el mismo día? Puedes estimar esta probabilidad generando muestras al azar de 23 cumpleaños y verificar coincidencias. Pista: puedes generar cumpleaños aleatorio con la función `randint` *del módulo* `random`*.*

Puedes descargar mi solución en `http://thinkpython2.com/code/birthday.py`*.*

Ejercicio 10.9. *Escribe una función que lea el archivo* `words.txt` *y construya una lista con un elemento por palabra. Escribe dos versiones de esta función, una usando el método* `append` *y la otra usando la notación* `t = t + [x]`*. ¿Cuál toma más tiempo en ejecutar? ¿Por qué?*

Solución: `http://thinkpython2.com/code/wordlist.py`*.*

Ejercicio 10.10. *Para verificar si una palabra está en la lista de palabras, podrías usar el operador* `in`*, pero sería lento porque busca en orden a través de las palabras.*

Debido a que las palabras están en orden alfabético, podemos acelerar las cosas con una búsqueda de bisección (también conocida como búsqueda binaria), que es similar a lo que haces cuando buscas una palabra en el diccionario (el libro, no la estructura de datos). Comienzas en el medio y verificas si la palabra que buscas viene antes de la palabra en el medio de la lista. Si es así, buscas en la primera mitad de la lista de la misma manera. De lo contrario, buscas la segunda mitad.

De cualquier manera, cortas por la mitad el espacio de búsqueda que queda. Si la lista de palabras tiene 113,809 palabras, tomará alrededor de 17 pasos para encontrar la palabra o concluir que no está.

Escribe una función llamada `in_bisect` *que tome una lista ordenada y un valor objetivo, y devuelva* `True` *si la palabra está en la lista y* `False` *si no está.*

¡O bien podrías leer la documentación del módulo `bisect` *y usar eso! Solución:* `http://thinkpython2.com/code/inlist.py`*.*

Ejercicio 10.11. *Dos palabras son un "par reverso" si cada una es el reverso de la otra. Escribe un programa que encuentre todos los pares reversos en la lista de palabras. Solución:* `http://thinkpython2.com/code/reverse_pair.py`.

Ejercicio 10.12. *Dos palabras se "entrelazan" si tomando letras alternas de cada una se forma una nueva palabra. Por ejemplo, "shoe" y "cold" se entrelazan para formar "schooled". Solución:* `http://thinkpython2.com/code/interlock.py`. *Crédito: Este ejercicio está inspirado en un ejemplo de* `http://puzzlers.org`.

1. *Escribe un programa que encuentre todos los pares de palabras que se entrelazan. Pista: ¡no revises todos los pares!*

2. *¿Puedes encontrar alguna palabra que sea un triple entrelazado; es decir, leyendo cada tres letras se forma una palabra, comenzando con la primera, la segunda o la tercera letra?*

Capítulo 11

Diccionarios

Este capítulo presenta otro tipo incorporado llamado diccionario. Los diccionarios son una de las mejores características de Python; son los bloques de construcción de muchos algoritmos eficientes y elegantes.

11.1. Un diccionario es un mapeo

Un **diccionario** es como una lista, pero más general. En una lista, los índices tienen que ser enteros; en un diccionario pueden ser (casi) de cualquier tipo.

Un diccionario contiene una colección de índices, que se llaman **claves**, y una colección de valores. Cada clave está asociada a un valor único. La asociación de una clave y un valor se llama **par clave-valor**, o a veces **ítem**.

En lenguaje matemático, un diccionario representa un **mapeo** de las claves a los valores, por tanto puedes decir también que cada clave "mapea a" un valor. Como ejemplo, construiremos un diccionario que mapea de palabras en inglés a palabras en español, así las claves y los valores son todas cadenas.

La función `dict` crea un nuevo diccionario sin ítems. Como `dict` es el nombre de una función incorporada, deberías evitar usarla como nombre de variable.

```
>>> ing_esp = dict()
>>> ing_esp
{}
```

Las llaves, `{}`, representan un diccionario vacío. Para agregar ítems al diccionario, puedes usar los corchetes:

```
>>> ing_esp['one'] = 'uno'
```

Esta línea crea un ítem que mapea de la clave `'one'` al valor `'uno'`. Si imprimimos el diccionario de nuevo, vemos un par clave-valor con un signo de dos puntos entre la clave y el valor:

```
>>> ing_esp
{'one': 'uno'}
```

Este formato de salida es también un formato de entrada. Por ejemplo, puedes crear un nuevo diccionario con tres ítems:

```
>>> ing_esp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Pero si imprimes `ing_esp`, podrías sorprenderte:

```
>>> ing_esp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

El orden de los pares clave-valor podría no ser el mismo. Si escribiste el mismo ejemplo en tu computador, podrías obtener un resultado diferente. En general, el orden de los ítems en un diccionario es impredecible.

Pero eso no es un problema porque los elementos de un diccionario nunca se indexan con índices enteros. En cambio, usas las claves para buscar los valores correspondientes:

```
>>> ing_esp['two']
'dos'
```

La clave `'two'` siempre mapea al valor `'dos'`, entonces el orden de los ítems no importa.

Si la clave no está en el diccionario, obtienes una excepción:

```
>>> ing_esp['four']
KeyError: 'four'
```

La función `len` funciona en diccionarios; devuelve el número de pares clave-valor:

```
>>> len(ing_esp)
3
```

El operador `in` también funciona en diccionarios; te dice si algo aparece como una *clave* en el diccionario (aparecer como un valor no basta).

```
>>> 'one' in ing_esp
True
>>> 'uno' in ing_esp
False
```

Para ver si algo aparece como un valor en un diccionario, puedes usar el método `values`, el cual devuelve una colección de valores, y entonces usar el operador `in`:

```
>>> valores = ing_esp.values()
>>> 'uno' in valores
True
```

El operador `in` usa diferentes algoritmos para las listas y los diccionarios. Para las listas, busca los elementos de la lista en orden, como en la Sección 8.6. A medida que la lista se vuelve más larga, el tiempo de búsqueda se hace más largo en proporción directa.

Los diccionarios de Python usan una estructura de datos llamada **tabla hash** que tiene una propiedad notable: el operador `in` toma casi la misma cantidad de tiempo sin importar cuántos ítems hay en el diccionario. Explico cómo es eso posible en la Sección B.4, pero la explicación podría no tener sentido hasta que hayas leído algunos capítulos más.

11.2. El diccionario como colección de contadores

Supongamos que te dan una cadena y quieres contar cuántas veces aparece cada letra. Hay varias maneras en que podrías hacerlo:

1. Podrías crear 26 variables, una para cada letra del alfabeto. Luego podrías recorrer la cadena y, para cada carácter, incrementar el contador correspondiente, probablemente usando un condicional encadenado.

2. Podrías crear una lista de 26 elementos. Luego podrías convertir cada carácter a un número (usando la función incorporada `ord`), usar el número como un índice dentro de la lista e incrementar el contador apropiado.

3. Podrías crear un diccionario con caracteres como claves y contadores como los valores correspondientes. La primera vez que veas un carácter, añadirías un ítem al diccionario. Después de eso incrementarías el valor de un ítem existente.

Cada una de estas opciones realiza la misma computación, pero cada una de ellas implementa esa computación de una manera diferente.

Una **implementación** es una manera de realizar una computación; algunas implementaciones son mejores que otras. Por ejemplo, una ventaja de la implementación con diccionario es que no tenemos que saber de antemano qué letras aparecen en la cadena y solo tenemos que hacer espacio para las letras que sí aparecen.

Así es como se vería el código:

```
def histograma(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

El nombre de la función es `histograma`, el cual es un término estadístico para una colección de contadores (o frecuencias).

La primera línea de la función crea un diccionario vacío. El bucle `for` recorre la cadena. En cada paso por el bucle, si el carácter `c` no está en el diccionario, creamos un nuevo ítem con clave `c` y valor inicial 1 (dado que hemos visto esta letra una vez). Si `c` ya está en el diccionario, incrementamos `d[c]`.

Funciona así:

```
>>> h = histograma('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

El histograma indica que las letras `'a'` y `'b'` aparecen una vez; `'o'` aparece dos veces, y así sucesivamente.

Los diccionarios tienen un método llamado `get` que toma una clave y un valor por defecto. Si la clave aparece en el diccionario, `get` devuelve el valor correspondiente; de lo contrario, devuelve el valor por defecto. Por ejemplo:

```
>>> h = histograma('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
```

```
1
>>> h.get('c', 0)
0
```

Como ejercicio, usa `get` para escribir `histograma` de manera más concisa. Deberías ser capaz de eliminar la sentencia `if`.

11.3. Bucles y diccionarios

Si usas un diccionario en una sentencia `for`, recorre las claves del diccionario. Por ejemplo, `imprimir_hist` imprime cada clave y el valor correspondiente:

```
def imprimir_hist(h):
    for c in h:
        print(c, h[c])
```

La salida se ve así:

```
>>> h = histograma('parrot')
>>> imprimir_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Nuevamente, las claves no estan en un orden particular. Para recorrer las claves en orden, puedes usar la función incorporada `sorted`:

```
>>> for clave in sorted(h):
...     print(clave, h[clave])
a 1
o 1
p 1
r 2
t 1
```

11.4. Consulta inversa

Dado un diccionario `d` y una clave `k`, es fácil encontrar el valor correspondiente `v = d[k]`. Esta operación se llama **consulta** (en inglés, *lookup*).

¿Y qué pasa si tienes `v` y quieres encontrar `k`? Tienes dos problemas: primero, podría haber más de una clave que mapee al valor `v`. Dependiendo de la aplicación, quizás puedas elegir una o tengas que hacer una lista que las contenga a todas. Segundo, no hay una sintaxis sencilla para hacer una **consulta inversa**; tienes que buscar.

Aquí hay una función que toma un valor y devuelve la primera clave que mapea a ese valor:

```
def consulta_inversa(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

Esta función es otro ejemplo del patrón de búsqueda pero que usa una característica que no hemos visto antes, `raise`. La **sentencia raise** causa una excepción; en este caso causa un `LookupError`, que es una excepción incorporada que se usa para indicar que falló una operación de consulta.

Si se llega al final del bucle, significa que `v` no aparece en el diccionario como valor, por lo que plantea una excepción.

Aquí hay un ejemplo de una consulta inversa eficaz:

```
>>> h = histograma('parrot')
>>> clave = consulta_inversa(h, 2)
>>> clave
'r'
```

Y una ineficaz:

```
>>> clave = consulta_inversa(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in consulta_inversa
LookupError
```

El efecto de cuando levantas una excepción mediante `raise` es el mismo de cuando Python levanta una: imprime un rastreo y un mensaje de error.

Cuando levantes una excepción, puedes proporcionar un mensaje de error detallado como argumento opcional. Por ejemplo:

```
>>> raise LookupError('el valor no aparece en el diccionario')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: el valor no aparece en el diccionario
```

Una consulta inversa es mucho más lenta que una consulta directa; si tienes que hacerlo a menudo, o si el diccionario se vuelve grande, el rendimiento de tu progama se verá afectado.

11.5. Diccionarios y listas

Las listas pueden aparecer como valores en un diccionario. Por ejemplo, si te dan un diccionario que mapea de letras a frecuencias, quizás quieras invertirlo; es decir, crear un diccionario que mapee de frecuencias a letras. Dado que podría haber muchas letras con la misma frecuencia, cada valor en el diccionario invertido debería ser una lista de letras.

Aquí hay una función que invierte un diccionario:

```
def invertir_dict(d):
    inverso = dict()
    for clave in d:
        valor = d[clave]
        if valor not in inverso:
            inverso[valor] = [clave]
        else:
            inverso[valor].append(clave)
    return inverso
```

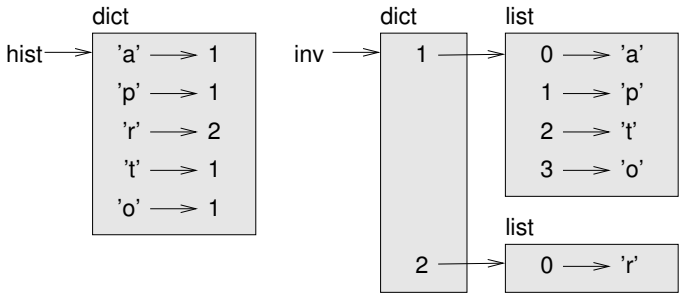

Figura 11.1: Diagrama de estado.

En cada paso por el bucle, `clave` obtiene una clave de `d` y `valor` obtiene el valor correspondiente. Si `valor` no está en `inverso`, lo cual significa que no lo hemos visto antes, entonces creamos un nuevo ítem y lo inicializamos con un **singleton** (una lista que contiene un único elemento). De lo contrario, hemos visto este valor antes, por lo cual anexamos a la lista la clave correspondiente.

Aquí hay un ejemplo:

```
>>> hist = histograma('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invertir_dict(hist)
>>> inv
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

La Figura 11.1 es un diagrama de estado que muestra a `hist` e `inv`. Un diccionario se representa como una caja con el tipo `dict` arriba suyo y el par clave-valor adentro. Si los valores son enteros, números de coma flotante o cadenas, los dibujo adentro de la caja, pero usualmente dibujo las listas afuera de la caja, solo para mantener simple al diagrama.

Las listas pueden ser valores en un diccionario, tal como muestra este ejemplo simple, pero no pueden ser claves. Aquí está lo que ocurre si lo intentas:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'ups'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Anteriormente mencioné que un diccionario se implementa usando una tabla hash y eso significa que las claves tienen que ser **hashables**.

Un **hash** es una función que toma un valor (de cualquier tipo) y devuelve un entero. Los diccionarios usan estos enteros, llamados valores hash, para almacenar y consultar pares clave-valor.

Este sistema funciona bien si las claves son inmutables. Pero si las claves son mutables, como las listas, ocurren cosas malas. Por ejemplo, cuando creas un par clave-valor, Python hashea la clave y la almacena en la ubicación correspondiente. Si modificas la clave y luego la hasheas de nuevo, iría a una ubicación diferente. En ese caso podrías tener dos entradas

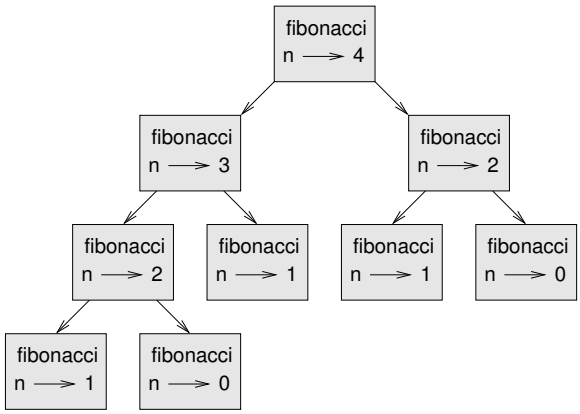

Figura 11.2: Gráfico de llamadas.

para la misma clave, o quizás no seas capaz de encontrar una clave. De cualquier manera, el diccionario no funcionaría de manera correcta.

Por eso es que las claves tienen que ser hashables y, por la misma razón, los tipos mutables como las listas no lo son. La manera más simple de evitar esta limitación es usar tuplas, las cuales veremos en el capítulo siguiente.

Dado que los diccionarios son mutables, estos no pueden usarse como claves, pero *pueden* usarse como valores.

11.6. Memos

Si jugaste con la función `fibonacci` de la Sección 6.7, quizás has notado que mientras más grande es el argumento que entregues, más tarda la función en ejecutarse. Además, el tiempo de ejecución aumenta rápidamente.

Para entender el por qué, considera la Figura 11.2, la cual muestra el **gráfico de llamadas** para `fibonacci` con `n=4`.

Un gráfico de llamadas muestra un conjunto de marcos de funciones, con líneas que conectan cada marco a los marcos de las funciones llamadas. En la parte de arriba del gráfico, `fibonacci` con `n=4` llama `fibonacci` con `n=3` y `n=2`. A su vez, `fibonacci` con `n=3` llama a `fibonacci` con `n=2` y `n=1`. Y así sucesivamente.

Cuenta cuántas veces se llama a `fibonacci(0)` y `fibonacci(1)`. Esta es una solución ineficiente para el probema y se pone peor a medida que el argumento se hace más grande.

Una solución es hacer un seguimiento de los valores que ya han sido calculados almacenándolos en un diccionario. Un valor previamente calculado que se almacena para un uso posterior se llama **memo**. Aquí hay una versión "memoizada" de `fibonacci`:

```
conocidos = {0:0, 1:1}

def fibonacci(n):
    if n in conocidos:
```

```
        return conocidos[n]

    resultado = fibonacci(n-1) + fibonacci(n-2)
    conocidos[n] = resultado
    return resultado
```

`conocidos` es un diccionario que hace un seguimiento de los números de Fibonacci que ya conocemos. Comienza con dos ítems: 0 mapea a 0 y 1 mapea a 1.

Cada vez que se llama a `fibonacci`, revisa a `conocidos`. Si el resultado ya está ahí, puede devolverlo inmediatamente. De lo contrario, tiene que calcular el nuevo valor, agregarlo al diccionario y devolverlo.

Si ejecutas esta versión de `fibonacci` y la comparas con la original, encontrarás que esta es mucho más rápida.

11.7. Variables globales

En los ejemplos anteriores, `conocidos` se crea fuera de la función, por lo que pertenece al marco especial llamado `__main__`. Las variables en `__main__` a veces se llaman **globales** porque se puede acceder a ellos desde cualquier función. A diferencia de las variables locales, que desaparecen cuando su función termina, las variables globales persisten de una llamada a función a la siguiente.

Es común usar variables globales para las **banderas** (en inglés, *flags*); es decir, variables booleanas que indican si una condición es verdadera. Por ejemplo, algunos programas usan una bandera llamada `verbose` para controlar el nivel de detalle en la salida:

```
verbose = True

def ejemplo1():
    if verbose:
        print('Ejecutando ejemplo1')
```

Si intentas reasignar una variable global, quizás te sorprendas. El siguiente ejemplo se supone que hace seguimiento de si la función ha sido llamada:

```
fue_llamada = False

def ejemplo2():
    fue_llamada = True          # INCORRECTO
```

Pero si lo ejecutas verás que el valor de `fue_llamada` no cambia. El problema es que `ejemplo2` crea una nueva variable local con nombre `fue_llamada`. La variable local se va cuando la función termina y no tiene efecto en la variable global.

Para reasignar una variable global dentro de una función tienes que **declarar** la variable global antes de usarla:

```
fue_llamada = False

def ejemplo2():
    global fue_llamada
    fue_llamada = True
```

La **sentencia global** le dice al intérprete algo como "En esta función, cuando digo `fue_llamada`, quiero decir la variable global; no crees una local."

Aquí hay un ejemplo que intenta actualizar una variable global:

```
contar = 0

def ejemplo3():
    contar = contar + 1          # INCORRECTO
```

Si lo ejecutas obtienes:

```
UnboundLocalError: local variable 'contar' referenced before assignment
```

Python supone que `contar` es local, y bajo ese supuesto lo estás leyendo antes de escribirlo. La solución, nuevamente, es declarar `contar` como global.

```
def ejemplo3():
    global contar
    contar += 1
```

Si una variable global se refiere a un valor mutable, puedes modificar el valor sin declarar la variable:

```
conocidos = {0:0, 1:1}

def ejemplo4():
    conocidos[2] = 1
```

Entonces puedes agregar, eliminar y reemplazar elementos de una lista global o diccionario global, pero si quieres reasignar la variable, tienes que declararla:

```
def ejemplo5():
    global conocidos
    conocidos = dict()
```

Las variables globales pueden ser útiles, pero si tienes muchas, y las modificas frecuentemente, pueden hacer que los programas sean difíciles de depurar.

11.8. Depuración

A medida que trabajes con conjuntos de datos más grandes, depurar imprimiendo y verificando la salida a mano puede volverse algo difícil de manejar. Aquí hay algunas sugerencias para depurar conjuntos grandes de datos.

Reduce la escala: Si es posible, reduce el tamaño del conjunto de datos. Por ejemplo, si el programa lee un archivo de texto, comienza solo con las primeras 10 líneas, o con el ejemplo más pequeño que puedas encontrar. Puedes editar aquellos archivos o (mejor) modificar el programa de manera que este lea solo las primeras `n` líneas.

Si hay un error, puedes reducir `n` al valor más pequeño que muestre el error y luego incrementarlo gradualmente mientras encuentras y corriges los errores.

Revisa resúmenes y tipos: En lugar de imprimir y verificar el conjunto de datos completo, considera imprimir resúmenes de los datos: por ejemplo, el número de ítems en el diccionario o el total de una lista de números.

Una causa común de errores de tiempo de ejecución es un valor que no es del tipo correcto. Para depurar esta clase de errores, a menudo es suficiente imprimir el tipo del valor.

Escribe verificaciones automáticas: A veces puedes escribir código para verificar errores de manera automática. Por ejemplo, si estás calculando el promedio de una lista de números, podrías verificar que el resultado no es mayor que el elemento más grande de la lista ni menor que el más pequeño. A esto se le llama "prueba de cordura" (en inglés, *sanity check*) porque detecta resultados que son "locos".

Otro tipo de prueba compara los resultados de dos computaciones diferentes para ver si son consistentes. A esto se le llama "prueba de consistencia".

Dale formato a la salida: Dar formato a la salida de la depuración puede hacer más fácil detectar un error. Vimos un ejemplo en la Sección 6.9. Otra herramienta que quizás encuentres útil es el módulo `pprint`, el cual proporciona una función `pprint` que muestra tipos incorporados en un formato más legible por humanos (`pprint` significa "pretty print").

Nuevamente, el tiempo que pasas construyendo andamiaje puede reducir el tiempo que pasas depurando.

11.9. Glosario

mapeo: Una relación en la cual cada elemento de un conjunto corresponde a un elemento de otro conjunto.

diccionario: Un mapeo de las claves a sus valores correspondienes.

par clave-valor: La representación del mapeo de una clave a un valor.

ítem: En un diccionario, otro nombre para un par clave-valor.

clave: Un objeto que aparece en un diccionario como la primera parte de un par clave-valor.

valor: Un objeto que aparece en un diccionario como la segunda parte de un par clave-valor. Esto es más específico que nuestro uso anterior de la palabra "valor".

implementación: Una manera de realizar una computación.

tabla hash: El algoritmo usado para implementar los diccionarios de Python.

función hash: Una función usada por una tabla hash para calcular la ubicación de una clave.

hashable: Un tipo que tiene una función hash. Los tipos inmutables como los enteros, números de coma flotante y cadenas son hashables; los tipos mutables como las listas y diccionarios no lo son.

consulta: Una operación de diccionario que toma una clave y encuentra el valor correspondiente.

consulta inversa: Una operación de diccionario que toma un valor y encuentra una o más claves que mapean a este.

sentencia raise: Una sentencia que (deliberadamente) levanta una excepción.

singleton: Una lista (u otra secuencia) con un solo elemento.

gráfico de llamadas: Un diagrama que muestra cada marco creado durante la ejecución de un programa, con una flecha desde cada llamador hacia cada llamado.

memo: Un valor calculado que se almacena para evitar una futura computación innecesaria.

variable global: Una variable definida fuera de la función. Las variables globales pueden ser accesibles desde cualquier función.

sentencia global: Una sentencia que declara global a un nombre de variable.

bandera: Una variable booleana que se usa para indicar si una condición es verdadera.

declaración: Una sentencia como `global` que le dice al intérprete algo sobre una variable.

11.10. Ejercicios

Ejercicio 11.1. *Escribe una función que lea las palabras en* `words.txt` *y las almacene como claves en un diccionario. No importa cuáles sean los valores. Luego puedes usar el operador* `in` *como una manera rápida de verificar si una cadena está en el diccionario.*

Si hiciste el Ejercicio 10.10, puedes comparar la velocidad de esta implementación con el operador `in` *de lista y la búsqueda de bisección.*

Ejercicio 11.2. *Lee la documentación del método de diccionario* `setdefault` *y úsalo para escribir una versión más concisa de* `invertir_dict`*. Solución: http://thinkpython2.com/code/invert_dict.py.*

Ejercicio 11.3. *Memoiza la función de Ackermann del Ejercicio 6.2 y ve si la memoización permite evaluar la función con argumentos más grandes. Pista: no. Solución: http://thinkpython2.com/code/ackermann_memo.py.*

Ejercicio 11.4. *Si hiciste el Ejercicio 10.7, ya tienes una función con nombre* `tiene_duplicados` *que toma una lista como parámetro y devuelve* `True` *si hay algún objeto que aparece más de una vez en la lista.*

Usa un diccionario para escribir una versión más rápida y simple de `tiene_duplicados`*. Solución: http://thinkpython2.com/code/has_duplicates.py.*

Ejercicio 11.5. *Dos palabras son "pares rotativos" si puedes rotar una de ellas y obtener la otra (ver* `rotar_palabra` *en el Ejercicio 8.5).*

Escribe un programa que lea una lista de palabras y encuentre todos los pares rotativos. Solución: http://thinkpython2.com/code/rotate_pairs.py.

Ejercicio 11.6. *Aquí hay otro Puzzler de* Car Talk *(http://www.cartalk.com/content/puzzlers):*

> *Este fue enviado por un compañero llamado Dan O'Leary. Se encontró hace poco con una palabra monosílaba de cinco letras común, que tiene la siguiente propiedad única. Cuando quitas la primera letra, las demás letras forman un homófono de la palabra original, es decir, una palabra que suena exactamente igual. Reemplaza la primera letra, es decir, ponla de nuevo y quita la segunda letra y el resultado es otro homófono de la palabra original. Y la pregunta es, ¿cuál es la palabra?*
>
> *Ahora voy a darte un ejemplo que no funciona. Veamos la palabra de cinco letras, 'wrack.' W-R-A-C-K, ya sabes, como 'wrack with pain.' Si quito la primero letra, me*

> *quedo con una palabra de cuatro letras, 'R-A-C-K.' Como en 'Holy cow, did you see the rack on that buck! It must have been a nine-pointer!' Es un homófono perfecto. Si vuelves a poner la 'w' y quitas la 'r' en su lugar, te quedas con la palabra 'wack' que es una palabra real, que simplemente no es un homófono de las otras dos palabras.*
>
> *Pero hay, de todas maneras, al menos una palabra de la cual Dan y nosotros sabemos, que entregará dos homófonos, si quitas cualquiera de las primeras dos letras para hacer dos nuevas palabras de cuatro letras. La pregunta es, ¿cuál es la palabra?*

Puedes usar el diccionario del Ejercicio 11.1 para verificar si una cadena está en la lista de palabras.

Para verificar si dos palabras son homófonas, puedes usar el Diccionario de Pronunciación de la CMU. Lo puedes descargar en `http://www.speech.cs.cmu.edu/cgi-bin/cmudict` *o en* `http://thinkpython2.com/code/c06d` *y también puedes descargar* `http://thinkpython2.com/code/pronounce.py`*, que proporciona una función con nombre* `read_dictionary` *que lee el diccionario de pronunciación y devuelve un diccionario de Python que mapea de cada palabra a una cadena que describe su pronunciación primaria.*

Escribe un programa que haga una lista de todas las palabras que resuelven el Puzzler. Solución: `http://thinkpython2.com/code/homophone.py`*.*

Capítulo 12

Tuplas

Este capítulo presenta un tipo incorporado más, la tupla, y luego muestra cómo las listas, los diccionarios y las tuplas trabajan juntos. Además, presento una característica útil para listas de argumento de longitud variable: los operadores de reunión y dispersión.

12.1. Las tuplas son inmutables

Una tupla es una secuencia de valores. Los valores pueden ser de cualquier tipo y están indexados por enteros, por lo que en ese sentido las tuplas se parecen mucho a las listas. La diferencia importante es que las tuplas son inmutables.

Sintácticamente, una tupla es una lista de valores separados por comas:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Aunque no es necesario, es común encerrar las tuplas en paréntesis:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un solo elemento, tienes que incluir una coma final:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

Un valor en paréntesis no es una tupla:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Otra manera de crear una tupla es la función incorporada `tuple`. Sin argumentos, esta crea una tupla vacía:

```
>>> t = tuple()
>>> t
()
```

Si el argumento es una secuencia (cadena, lista o tupla), el resultado es una tupla con los elementos de la secuencia:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

Dado que `tuple` es el nombre de una función incorporada, deberías evitar usarlo como nombre de una variable.

La mayoría de los operadores de lista también funcionan en tuplas. El operador de corchetes indexa un elemento:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

Y el operador de corte selecciona un rango de elementos.

```
>>> t[1:3]
('b', 'c')
```

Pero si intentas modificar uno de los elementos de la tupla, obtienes un error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Dado que las tuplas son inmutables, no puedes modificar los elementos. Sin embargo, puedes reemplazar una tupla con otra:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

Esta sentencia crea una nueva tupla y luego hace que `t` se refiera a esta.

Los operadores relacionales funcionan con las tuplas y otras secuencias; Python comienza comparando el primer elemento de cada secuencia. Si son iguales, avanza a los siguientes elementos, y así sucesivamente, hasta que encuentre elementos que sean diferentes. Los elementos posteriores no se consideran (incluso si son realmente grandes).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

12.2. Asignación de tupla

A menudo es útil intercambiar los valores de dos variables. Con asignaciones convencionales, tienes que usar una variable temporal. Por ejemplo, para intercambiar `a` y `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Esta solución es incómoda; la **asignación de tupla** es más elegante:

```
>>> a, b = b, a
```

El lado izquierdo es una tupla de variables; el lado derecho es una tupla de expresiones. Cada valor es asignado a su respectiva variable. Todas las expresiones en el lado derecho son evaluadas antes de cualquiera de las asignaciones.

El número de variables en el lazo izquierdo y el número de valores del lado derecho tienen que ser el mismo:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

De manera más general, el lado derecho puede ser cualquier tipo de secuencia (cadena, lista o tupla). Por ejemplo, para separar una direccion de email en nombre de usuario y dominio, podrías intentar:

```
>>> direccion = 'monty@python.org'
>>> nombre_usuario, dominio = direccion.split('@')
```

El valor de retorno de `split` es una lista con dos elementos; el primer elemento se asigna a `nombre_usuario`, el segundo a `dominio`.

```
>>> nombre_usuario
'monty'
>>> dominio
'python.org'
```

12.3. Tuplas como valores de retorno

Estrictamente hablando, una función solo puede devolver un valor de retorno, pero si el valor es una tupla, el efecto es el mismo que devolver múltiples valores. Por ejemplo, si quieres dividir dos enteros y calcular el cociente y el resto, es ineficiente calcular `x//y` y luego `x%y`. Es mejor calcular ambos al mismo tiempo.

La función incorporada `divmod` toma dos argumentos y devuelve una tupla de dos valores, el cociente y el resto. Puedes almacenar el resultado como una tupla:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

O bien usar asignación de tupla para almacenar los elementos por separado:

```
>>> cociente, resto = divmod(7, 3)
>>> cociente
2
>>> resto
1
```

Aquí hay un ejemplo de una función que devuelve una tupla:

```
def min_max(t):
    return min(t), max(t)
```

`max` y `min` son funciones incorporadas que encuentran los elementos más grande y más pequeño de una secuencia. `min_max` calcula ambos y devuelve una tupla de dos valores.

12.4. Tuplas de argumentos de longitud variable

Las funciones pueden tomar un número variable de argumentos. Un nombre de parámetro que comienza con `*` hace una **reunión** de argumentos en una tupla. Por ejemplo, `printall` toma cualquier número de argumentos y los imprime:

```
def printall(*args):
    print(args)
```

El parámetro de reunión puede tener cualquier nombre que te guste, pero `args` es convencional. Aquí se muestra cómo opera la función:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

El complemento de la reunión es la **dispersión**. Si tienes una secuencia de valores y quieres pasarlo a una función como multiples argumentos, puedes usar el operador `*`. Por ejemplo, `divmod` toma exactamente dos argumentos; no funciona con una tupla:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Pero si dispersas la tupla, funciona:

```
>>> divmod(*t)
(2, 1)
```

Muchas de las funciones incorporadas usan tuplas de argumento de longitud variable. Por ejemplo, `max` y `min` pueden tomar cualquier número de argumentos:

```
>>> max(1, 2, 3)
3
```

Pero `sum` no puede.

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

Como ejercicio, escribe una función llamada `sumar_todos` que tome cualquier número de argumentos y devuelva su suma.

12.5. Listas y tuplas

`zip` es una función incorporada que toma dos o más secuencias y las intercala. El nombre de la función se refiere a una cremallera (*zipper*), ya que esta intercala dos filas de dientes.

Este ejemplo hace zip a una cadena y una lista:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

El resultado es un **objeto zip** que sabe cómo iterar a través de los pares. El uso más común de `zip` es en un bucle `for`:

```
>>> for par in zip(s, t):
...     print(par)
...
('a', 0)
('b', 1)
('c', 2)
```

Un objeto zip es una especie de **iterador**, que es cualquier objeto que itera a través de una secuencia. Los iteradores son similares a las listas de alguna manera, pero a diferencia de las listas, no puedes usar un índice para seleccionar un elemento de un iterador.

Si quieres usar operadores de lista y métodos, puedes usar un objeto zip para hacer una lista:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

El resultado es una lista de tuplas; en este ejemplo, cada tupla contiene un carácter de la cadena y el elemento correspondiente de la lista.

Si las secuencias no tienen la misma longitud, el resultado tiene la longitud de la secuencia más corta.

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

Puedes usar asignación de tupla en un bucle `for` para recorrer una lista de tuplas:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letra, numero in t:
    print(numero, letra)
```

En cada paso por el bucle, Python selecciona la siguiente tupla en la lista y asigna los elementos a `letra` y `numero`. La salida de este bucle es:

```
0 a
1 b
2 c
```

Si combinas `zip`, `for` y asignación de tupla, obtienes una manera útil para recorrer dos (o más) secuencias al mismo tiempo. Por ejemplo, `tiene_coincidencia` toma dos secuencias, `t1` y `t2`, y devuelve `True` si hay un índice `i` tal que `t1[i] == t2[i]`:

```
def tiene_coincidencia(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Si necesitas recorrer los elementos de una secuencia y sus índices, puedes usar la función incorporada `enumerate`:

```
for indice, elemento in enumerate('abc'):
    print(indice, elemento)
```

El resultado de `enumerate` es un objeto enumerate, que itera una secuencia de pares; cada par contiene un índice (comenzando desde 0) y un elemento de la secuencia dada. En este ejemplo, la salida es

```
0 a
1 b
2 c
```

Nuevamente.

12.6. Diccionarios y tuplas

Los diccionarios tienen un método llamado `items` que devuelve una secuencia de tuplas, donde cada tupla es un par clave-valor.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

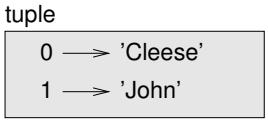

Figura 12.1: Diagrama de estado.

El resultado es un objeto `dict_items`, el cual es un iterador que itera los pares clave-valor. Puedes usarlo en un bucle `for` así:

```
>>> for clave, valor in d.items():
...     print(clave, valor)
...
c 2
a 0
b 1
```

Tal como esperarías de un diccionario, los ítems no están en un orden particular.

En sentido contrario, puedes usar una lista de tuplas para inicializar un nuevo diccionario:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Combinando `dict` con `zip` se produce una manera concisa de crear un diccionario:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

El método de diccionario `update` además toma una lista de tuplas y las agrega, como pares clave-valor, a un diccionario existente.

Es común usar tuplas como claves en diccionarios (principalmente porque no puedes usar listas). Por ejemplo, un directorio telefónico podría mapear de pares apellido, nombre a números telefónicos. Suponiendo que hemos definido `apellido`, `nombre` y `numero`, podríamos escribir:

```
directorio[apellido, nombre] = numero
```

La expresión en corchetes es una tupla. Podríamos usar asignación de tupla para recorrer este diccionario.

```
for apellido, nombre in directorio:
    print(nombre, apellido, directorio[apellido,nombre])
```

Este bucle recorre las claves en `directorio`, que son tuplas. Asigna los elementos de cada tupla a `apellido` y `nombre`, luego imprime el nombre completo y el número telefónico correspondiente.

Hay dos maneras de representar tuplas en diagrama de estado. La versión más detallada muestra los índices y elementos tal como aparecen en una lista. Por ejemplo, la tupla `('Cleese', 'John')` se mostraría como en la Figura 12.1.

Pero en un diagrama más grande quizás quieras omitir los detalles. Por ejemplo, un diagrama del directorio telefónico podría mostrarse como en la Figura 12.2.

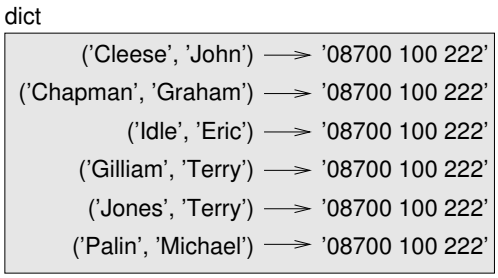

Figura 12.2: Diagrama de estado.

Aquí las tuplas se muestran usando la notación de la sintáxis de Python. El número de teléfono del diagrama es la línea de reclamos de la BBC, así que por favor no lo llames.

12.7. Secuencias de secuencias

Me he enfocado en listas de tuplas, pero casi todos los ejemplos de este capítulo funcionan también con listas de listas, tuplas de tuplas y tuplas de listas. Para evitar enumerar las posibles combinaciones, a veces es más fácil hablar de secuencias de secuencias.

En muchos contextos, los diferentes tipos de secuencias (cadenas, listas y tuplas) se pueden usar indistintamente. Entonces, ¿cómo deberías escoger uno por sobre los otros?

Para comenzar con lo obvio, las cadenas son más limitadas que otras secuencias porque los elementos tienen que ser caracteres. Además, son inmutables. Si necesitas la posibilidad de cambiar los caracteres en una cadena (en contraposición a crear una nueva cadena), quizás quieras usar una lista de caracteres en su lugar.

Las listas son más comunes que las tuplas, principalmente porque son mutables. Sin embargo, hay algunos casos donde podrías preferir tuplas:

1. En algunos contextos, como una sentencia `return`, es sintácticamente más simple crear una tupla que una lista.
2. Si quieres usar una secuencia como clave de diccionario, tienes que usar un tipo inmutable como una tupla o una cadena.
3. Si pasas una secuencia como argumento a una función, usar tuplas reduce las posibilidades de comportamiento inesperado debido a los alias.

Dado que las tuplas son inmutables, no proporcionan métodos como `sort` y `reverse`, los cuales modifican listas existentes. Sin embargo, Python proporciona la función incorporada `sorted`, la cual toma cualquier secuencia y devuelve una nueva lista con los mismos elementos en orden, y `reversed`, que toma una secuencia y devuelve un iterador que recorre la lista en orden invertido.

12.8. Depuración

Las listas, diccionarios y tuplas son ejemplos de **estructuras de datos**; en este capítulo comenzamos a ver estructuras de datos combinadas, como listas de tuplas, o diccionarios

que contienen tuplas como claves y listas como valores. Las estructuras de datos combinadas son útiles, pero son propensas a lo que yo llamo **errores de forma**; es decir, errores causados cuando una estructura de datos tiene el tipo, tamaño o estructura incorrecta. Por ejemplo, si estás esperando una lista que contiene un entero y yo te doy un simple y viejo entero (no en una lista), no funcionará.

Para ayudar a depurar esta clase de errores, he escrito un módulo llamado `structshape` que proporciona una función, también llamada `structshape`, que toma cualquier tipo de estructura de datos como argumento y devuelve una cadena que resume su forma. Puedes descargarlo en `http://thinkpython2.com/code/structshape.py`

Aquí hay un resultado para una lista simple:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

Un programa más elegante podría escribir "list of 3 ints", pero fue más fácil no lidiar con los plurales. Aquí hay una lista de listas:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
'list of 3 list of 2 int'
```

Si los elementos de la lista no son del mismo tipo, `structshape` los agrupa, en orden, por tipo:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

Aquí hay una lista de tuplas:

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

Y aquí hay un diccionario con 3 ítems que mapean enteros a cadenas.

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

Si tienes problemas al hacer seguimiento de tus estructuras de datos, `structshape` puede ayudar.

12.9. Glosario

tupla: Una secuencia inmutable de elementos.

asignación de tupla: Una asignación con una secuencia en el lado derecho y una tupla de variables en el lado izquierdo. El lado derecho es evaluado y luego sus elementos son asignados a las variables en el lado izquierdo.

reunión: Una operación que junta múltiples argumentos en una tupla.

dispersión: Una operación que hace que una secuencia se comporte como múltiples argumentos.

objeto zip: El resultado de llamar a la función incorporada `zip`; un objeto que itera a través de una secuencia de tuplas.

iterador: Un objeto que itera a través de una secuencia, pero que no proporciona operadores ni métodos de lista.

estructura de datos: Una colección de valores relacionados, a menudo organizada en listas, diccionarios, tuplas, etc.

error de forma: Un error causado porque un valor tiene la forma incorrecta; es decir, el tipo o tamaño incorrecto.

12.10. Ejercicios

Ejercicio 12.1. *Escribe una función llamada* `mas_frecuente` *que tome una cadena e imprima las letras en orden de frecuencia descendente. Encuentra ejemplos de texto en varios idiomas diferentes y ve cómo varía la frecuencia de letras entre los idiomas. Compara tus resultados con las tablas en* `http://en.wikipedia.org/wiki/Letter_frequencies`*. Solución:* `http://thinkpython2.com/code/most_frequent.py`*.*

Ejercicio 12.2. *¡Más anagramas!*

1. *Escribe un programa que lea una lista de palabras desde un archivo (ver Sección 9.1) e imprima todos los conjuntos de palabras que son anagramas.*

   Aquí hay un ejemplo de cómo se vería la salida:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Pista: quizás quieras construir un diccionario que mapee de una colección de letras a una lista de palabras que puedan escribir con esas letras. La pregunta es, ¿cómo puedes representar la colección de letras de manera que se pueda usar como clave?

2. *Modifica el programa anterior para que imprima la lista de anagramas más grande primero, seguido de la segunda más grande, y así sucesivamente.*

3. *En Scrabble, un "bingo" es cuando juegas todas las siete fichas de tu atril, junto con una letra en el tablero, para formar una palabra de ocho letras. ¿Qué colección de 8 letras forma la mayor cantidad de bingos posible?*

   Solución: `http://thinkpython2.com/code/anagram_sets.py`*.*

Ejercicio 12.3. *Dos palabras forman un "par de metátesis" si puedes transformar una en la otra intercambiando dos letras; por ejemplo, "converse" y "conserve". Escribe un programa que encuentre todos los pares de metátesis en el diccionario. Pista: no pruebes todos los pares de palabras ni pruebes todos los posibles intercambios. Solución:* `http://thinkpython2.com/code/metathesis.py`*. Crédito: Este ejercicio está inspirado por un ejemplo de* `http://puzzlers.org`*.*

Ejercicio 12.4. *Aquí hay otro Puzzler de Car Talk (*`http://www.cartalk.com/content/puzzlers`*):*

> *¿Cuál es la palabra en inglés más larga que, a medida que eliminas sus letras una a la vez, sigue siendo una palabra en inglés válida?*
>
> *A ver, las letras se pueden eliminar desde cualquier extremo, o del medio, pero no puedes reorganizar ninguna de las letras. Cada vez que retiras una letra, terminas con otra palabra en inglés. Si haces eso, eventualmente vas a terminar con una letra y esa también va a ser una palabra en inglés, una que se encuentra en el diccionario. Quiero saber: ¿cuál es la palabra más larga y cuantas letras tiene?*
>
> *Voy a darte un pequeño ejemplo modesto: Sprite. ¿De acuerdo? Inicias con sprite, quitas una letra, una del interior de la palabra, te llevas la r, y nos quedamos con la palabra spite, luego quitamos la e del final, nos quedamos con spit, quitamos la s, nos quedamos con pit, it, y por último I.*

Escribe un programa que encuentre todas las palabras que se pueden reducir de esta manera, y luego encuentra la más larga.

Este ejercicio es más desafiante que la mayoría, así que aquí hay algunas sugerencias:

1. *Quizás quieras escribir una función que tome una palabra y obtenga una lista de todas las palabras que se pueden formar eliminando una letra. Estas son las "hijas" de la palabra.*
2. *De manera recursiva, una palabra es reducible si cualquiera de sus hijas son reducibles. Como caso base, puedes considerar la cadena vacía reducible.*
3. *La lista de palabras que proporciono,* `words.txt`*, no contiene palabras de una sola letra. Entonces quizás quieras agregar "I", "a" y la cadena vacía.*
4. *Para mejorar el desempeño de tu programa, quizás quieras memoizar las palabras que se sabe que son reducibles.*

Solución: `http://thinkpython2.com/code/reducible.py`.

Capítulo 13

Estudio de caso: selección de estructura de datos

En este punto has aprendido sobre las estructuras de datos de Python esenciales, y has visto algunos de los algoritmos que los usan. Si te gustaría saber más sobre algoritmos, este podría ser un buen momento para leer el Apéndice B. Sin embargo, no tienes que leerlo antes de continuar; puedes leerlo cuando te interese.

Este capítulo presenta un estudio de caso con ejercicios que te hacen pensar sobre escoger estructuras de datos y practicar usándolos.

13.1. Análisis de frecuencia de palabras

Como siempre, deberías al menos intentar los ejercicios antes de que leas mis soluciones.
Ejercicio 13.1. *Escribe un programa que lea un archivo, separe cada línea en palabras, quite los espacios en blanco y la puntuación de las palabras, y las convierta minúsculas.*

Pista: El módulo `string` *proporciona una cadena con nombre* `whitespace`*, que contiene los caracteres de espacio, tabulación, nueva línea, etc., y* `puntuación` *que contiene los caracteres de puntuación. Veamos si podemos hacer que Python diga groserías:*

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Además, podrías considerar el uso de los métodos de cadena `strip`, `replace` *y* `translate`.
Ejercicio 13.2. *Ve a Project Gutenberg (*`http://gutenberg.org`*) y descarga tu libro sin derechos de autor favorito en formato de texto plano.*

Modifica tu programa del ejercicio anterior para que lea el libro que descargaste, se salte la información del encabezado al principio del archivo y procese el resto de las palabras como antes.

Luego, modifica el programa para que cuente el número total de palabras en el libro y el número de veces que se usa cada palabra.

Imprime el número de palabras diferentes usadas en el libro. Compara libros diferentes de autores diferentes, escritos en épocas diferentes.. ¿Qué autor usa el vocabulario más extenso?

Ejercicio 13.3. *Modifica el programa del ejercicio anterior para que imprima las 20 palabras usadas con mayor frecuencia en el libro.*

Ejercicio 13.4. *Modifica el programa anterior para que lea una lista de palabras (ver Sección 9.1) y luego imprima todas las palabras del libro que no están en la lista de palabras. ¿Cuántas de ellas son errores tipográficos? ¿Cuántas de ellas son palabras comunes que* deberían *estar en la lista de palabras y cuántas de ellas son realmente oscuras?*

13.2. Números aleatorios

Dadas las mismas entradas, la mayoría de los programas de computador generan las mismas salidas cada vez, por lo que se dice que son **deterministas**. El determinismo es usualmente algo bueno, ya que esperamos que los mismos cálculos produzcan el mismo resultado. Para algunas aplicaciones, sin embargo, queremos que el computador sea impredecible. Los juegos son un ejemplo obvio, pero hay más.

Hacer un programa verdaderamente no determinista resulta difícil, pero hay maneras de hacer que al menos parezca no determinista. Una de ellas es usar algoritmos que generen números **pseudoaleatorios**. Los números pseudoaleatorios no son verdaderamente aleatorios porque se generan por una computación determinista, pero solo mirando a los números es casi imposible distinguirlos de los aleatorios.

El módulo `random` proporciona funciones que generan números pseudoaleatorios (que simplemente llamaré "aleatorios" desde aquí en adelante).

La función `random` devuelve un número de coma flotante aleatorio entre 0.0 y 1.0 (incluyendo 0.0 pero no 1.0). Cada vez que llamas a `random`, obtienes el siguiente número de una larga serie. Para ver una muestra, ejecuta este bucle:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

La función `randint` toma los parámetros `low` y `high` y devuelve un entero entre `low` y `high` (incluyendo a ambos).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para escoger un elemento de una secuencia de manera aleatoria, puedes usar `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

El módulo `random` también proporciona funciones para generar valores aleatorios a partir de distribuciones continuas incluyendo la gaussiana, exponencial, gamma y algunas más.

Ejercicio 13.5. *Escribe una función con nombre* `escoger_de_hist` *que tome un histograma como se definió en la Sección 11.2 y devuelva un valor aleatorio del histograma, esogido con probabilidad proporcional a la frecuencia. Por ejemplo, para este histograma:*

```
>>> t = ['a', 'a', 'b']
>>> hist = histograma(t)
>>> hist
{'a': 2, 'b': 1}
```

tu función debería devolver `'a'` *con probabilidad* 2/3 *y* `'b'` *con probabilidad* 1/3.

13.3. Histograma de palabras

Deberías intentar los ejemplos anteriores antes de continuar. Puedes descargar mi solución en `http://thinkpython2.com/code/analyze_book1.py`. Además, necesitarás `http://thinkpython2.com/code/emma.txt`.

Aquí hay un programa que lee un archivo y construye un histograma de las palabras en el archivo:

```
import string

def procesar_archivo(nombre_archivo):
    hist = dict()
    fp = open(nombre_archivo)
    for linea in fp:
        procesar_linea(linea, hist)
    return hist

def procesar_linea(linea, hist):
    linea = linea.replace('-', ' ')

    for palabra in linea.split():
        palabra = palabra.strip(string.punctuation + string.whitespace)
        palabra = palabra.lower()
        hist[palabra] = hist.get(palabra, 0) + 1

hist = procesar_archivo('emma.txt')
```

Este programa lee `emma.txt`, el cual contiene el texto de *Emma* de Jane Austen.

`procesar_archivo` recorre las líneas del archivo, pasándolas una a la vez a `procesar_linea`. El histograma `hist` se usa como acumulador.

`procesar_linea` usa el método de cadena `replace` para reemplazar los guiones con espacios antes de usar `split` para separar la línea en una lista de cadenas. Recorre la lista de palabras y usa `strip` y `lower` para eliminar la puntuación y convertir a minúsculas. (Es una abreviatura decir que las cadenas se "convierten"; recuerda que las cadenas son inmutables, por lo que los métodos como `strip` y `lower` devuelven cadenas nuevas.)

Finalmente, `procesar_linea` actualiza el histograma creando un nuevo ítem o incrementando uno existente.

Para contar el número total de palabras en el archivo, podemos sumar las frecuencias del histograma:

```
def total_palabras(hist):
    return sum(hist.values())
```

El número de palabras diferentes es solo el número de ítems en el diccionario:

```
def palabras_diferentes(hist):
    return len(hist)
```

Aquí hay algo de código para imprimir los resultados:

```
print('Número total de palabras:', total_palabras(hist))
print('Número de palabras diferentes:', palabras_diferentes(hist))
```

Y los resultados:

```
Número total de palabras: 161080
Número de palabras diferentes: 7214
```

13.4. Palabras más comunes

Para encontrar las palabras más comunes, podemos hacer una lista de tuplas, donde cada tupla contenga una palabra y su frecuencia, y ordenarla.

La siguiente función toma un histograma y devuelve una lista de tuplas frecuencia-palabra:

```
def mas_comunes(hist):
    t = []
    for clave, valor in hist.items():
        t.append((valor, clave))

    t.sort(reverse=True)
    return t
```

En cada tupla, la frecuencia aparece primero, por lo que la lista resultante es ordenada por frecuencia. Aquí hay un bucle que imprime las diez palabras más comunes:

```
t = mas_comunes(hist)
print('Las palabras más comunes son:')
for frec, palabra in t[:10]:
    print(palabra, frec, sep='\t')
```

Yo uso el argumento de palabra clave `sep` para decirle a `print` que use un carácter de tabulación como "separador", en lugar de un espacio, así la segunda columna está alineada. Aquí están los resultados de *Emma*:

```
Las palabras más comunes son:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

Este código se puede simplificar usando el parámetro `key` de la función `sort`. Si tienes curiosidad, puedes leer sobre este en `https://wiki.python.org/moin/HowTo/Sorting`.

13.5. Parámetros opcionales

Hemos visto funciones y métodos que toman argumentos opcionales. Es posible, además, escribir funciones definidas por el programador con argumentos opcionales. Por ejemplo, aquí hay una función que imprime las palabras más comunes en un histograma:

```
def imprime_mas_comunes(hist, num=10):
    t = mas_comunes(hist)
    print('Las palabras más comunes son:')
    for frec, palabra in t[:num]:
        print(palabra, frec, sep='\t')
```

El primer parámetro es obligatorio; el segundo es opcional. El **valor por defecto** de `num` es 10.

Si solo entregas un argumento:

```
imprimir_mas_comunes(hist)
```

`num` obtiene el valor por defecto. Si entregas dos argumentos:

```
imprimir_mas_comunes(hist, 20)
```

`num` obtiene el valor del argumento en su lugar. En otras palabras, el argumento opcional **anula** al valor por defecto.

Si una función tiene parámetros tanto obligatorios como opcionales, todos los parámetros obligatorios tienen que ir primero, seguido por los opcionales.

13.6. Diferencia de diccionarios

Encontrar las palabras del libro que no están en la lista de palabras de `words.txt` es un problema que podrías reconocer como diferencia de conjuntos; es decir, queremos encontrar todas las palabras de un conjunto (las palabras en el libro) que no están en el otro (las palabras en la lista).

`diferencia` toma dos diccionarios, `d1` y `d2`, y devuelve un nuevo diccionario que contiene todas las claves de `d1` que no están en `d2`. Dado que en realidad no nos importan los valores, los ponemos todos como None.

```
def diferencia(d1, d2):
    res = dict()
    for clave in d1:
        if clave not in d2:
            res[clave] = None
    return res
```

Para encontrar las palabras en el libro que no están en `words.txt`, podemos usar `procesar_archivo` para construir un histograma para `words.txt`, y luego obtener la diferencia:

```
palabras = procesar_archivo('words.txt')
dif = diferencia(hist, palabras)

print("Palabras en el libro que no están en la lista de palabras:")
for palabra in dif:
    print(palabra, end=' ')
```

Aquí hay algunos de los resultados para *Emma*:

```
Palabras en el libro que no están en la lista de palabras:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Algunas de esas palabras son nombres y posesivos. Otras, como "rencontre", ya no son de uso común. Sin embargo, ¡algunas son palabras comunes que realmente deberían estar en la lista!

Ejercicio 13.6. *Python proporciona una estructura de datos llamada* `set` *que provee muchas operaciones de conjunto comunes. Puedes leer sobre estas en la Sección 19.5, o leer la documentación en* `http://docs.python.org/3/library/stdtypes.html#types-set`.

Escribe un programa que use diferencia de conjuntos para encontrar palabras en el libro que no están en la lista. Solución: `http://thinkpython2.com/code/analyze_book2.py`.

13.7. Palabras aleatorias

Para escoger una palabra de forma aleatoria del histograma, el algoritmo más simple es construir una lista con múltiples copias de cada palabra, según la frecuencia observada, y luego escoger de la lista:

```
def palabra_aleatoria(h):
    t = []
    for palabra, frec in h.items():
        t.extend([palabra] * frec)

    return random.choice(t)
```

La expresión `[palabra] * frec` crea una lista con `frec` copias de la cadena `palabra`. El método `extend` es similar a `append`, excepto que el argumento es una secuencia.

Este algoritmo funciona, pero no es muy eficiente; cada vez que escoges una palabra aleatoria, reconstruye la lista, que es tan grande como el libro original. Una mejora obvia es construir la lista una vez y luego hacer múltiples selecciones, pero la lista es grande aún.

Una alternativa es:

1. Usar `keys` para obtener una lista de las palabras del libro.

2. Construir una lista que contenga la suma acumulativa de las frecuencias de las palabras (ver Ejercicio 10.2). El último ítem en esta lista es el número total de palabras en el libro: n.

3. Escoger un número de 1 a n. Usar búsqueda de bisección (ver Ejercicio 10.10) para encontrar el índice donde el número aleatorio sería insertado en la suma acumulativa.

4. Usar el índice para encontrar la palabra correspondiente en la lista de palabras.

Ejercicio 13.7. *Escribe un programa que use este algoritmo para escoger una palabra aleatoria del libro. Solución:* `http://thinkpython2.com/code/analyze_book3.py`.

13.8. Análisis de Markov

Si escoges palabras del libro de manera aleatoria, puedes obtener una idea del vocabulario, pero probablemente no obtendrás una oración:

```
this the small regard harriet which knightley's it most things
```

Una serie de palabras aleatorias rara vez tiene sentido porque no hay relación entre palabras sucesivas. Por ejemplo, en una oración real esperarías que un artículo como "the" esté seguida por un adjetivo o un sustantivo, y probablemente no por un verbo o un adverbio.

Una manera de medir estas relaciones es el análisis de Markov, que caracteriza, para una secuencia de palabras dada, la probabilidad de las palabras que podrían venir después. Por ejemplo, la canción *Eric, the Half a Bee* comienza:

> Half a bee, philosophically,
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. D'you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

En este texto, la frase "half the" está siempre seguida por la palabra "bee", pero la frase "the bee" podría estar seguida por "has" o "is".

El resultado del análisis de Markov es un mapeo de cada prefijo (como "half the" y "the bee") a todos los posibles sufijos (como "has" e "is").

Dado este mapeo, puedes generar un texto aleatorio comenzando con cualquier prefijo y escogiendo de manera aleatoria en los posibles sufijos. Después, puedes combinar el final del prefijo y el nuevo sufijo para formar el nuevo prefijo, y repetir.

Por ejemplo, si comienzas con el prefijo "Half a", entonces la siguiente palabra tiene que ser "bee", porque el prefijo solo aparece una vez en el texto. El siguiente prefijo es "a bee", por lo que el siguiente sufijo podría ser "philosophically", "be" o "due".

En este ejemplo, la longitud del prefijo es siempre dos, pero puedes hacer análisis de Markov con cualquier longitud de prefijo.

Ejercicio 13.8. *Análisis de Markov:*

1. *Escribe un programa que lea un texto desde un archivo y realice análisis de Markov. El resultado podría ser un diccionario que mapee de prefijos a una colección de posibles sufijos. La colección podría ser una lista, tupla o diccionario; depende de ti hacer una elección apropiada. Puedes probar tu programa con prefijo de largo dos, pero podrías escribir el programa de una manera en que resulte fácil intentar otras longitudes.*

2. *Agrega una función al programa anterior que genere texto aleatorio basado en el análisis de Markov. Aquí hay un ejemplo de* Emma *con prefijo de largo 2:*

   > *He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?I cannot make speeches, Emma:"he soon cut it all himself.*

Para este ejemplo, dejé la puntuación unida a las palabras. El resultado es casi sintácticamente correcto, pero no del todo. Semánticamente, casi tiene sentido, pero no del todo.

¿Qué ocurre si aumentas la longitud del prefijo? ¿Tiene más sentido el texto aleatorio?

3. *Una vez que tu programa funcione, quizás quieras intentar un mash-up: si combinas texto de dos o más libros, el texto aleatorio que generes mezclará el vocabulario y las frases de las fuentes de maneras interesantes.*

Crédito: Este estudio de caso está basado en un ejemplo de Kernighan and Pike, The Practice of Programming*, Addison-Wesley, 1999.*

Deberías intentar este ejercicio antes de continuar; luego puedes descargar mi solución en `http://thinkpython2.com/code/markov.py`. Además, necesitarás `http://thinkpython2.com/code/emma.txt`.

13.9. Estructuras de datos

Usar análisis de Markov para generar texto aleatorio es divertido, pero también hay un punto en este ejercicio: la selección de la estructura de datos. En tu solución al ejercicio anterior, tuviste que escoger:

- Cómo representar los prefijos.
- Cómo representar la colección de posibles sufijos.
- Cómo representar el mapeo de cada prefijo a la colección de posibles sufijos.

El último es fácil: un diccionario es la elección obvia para mapear de claves a valores correspondientes.

Para los prefijos, las opciones más obvias son cadenas, lista de cadenas o tupla de cadenas.

Para los sufijos, una opción es una lista; otra es un histograma (diccionario).

¿Cómo deberías escoger? El primero paso es pensar en las operaciones que necesitarás implementar para cada esturcura de datos. Para prefijos, necesitamos poder eliminar palabras del principio y agregar al final. Por ejemplo, si el prefijo actual es "Half a", y la siguiente palabra es "bee", necesitas poder formar el siguiente prefijo, "a bee".

Tu primera elección podría ser una lista, dado que es fácil agregar y eliminar elementos, pero también necesitamos poder usar los prefijos y claves en un diccionario, así que eso descarta las listas. Con las tuplas, no puedes anexar o eliminar, pero puedes usar el operador suma para formar una nueva tupla:

```
def cambiar(prefijo, palabra):
    return prefijo[1:] + (palabra,)
```

`cambiar` toma una tupla de palabras, `prefijo`, y una cadena, `palabra`, y forma una nueva tupla que tiene todas las palabras en `prefijo` excepto la primera, y `palabra` agregada al final.

Para la colección de sufijos, las operaciones que necesitamos realizar incluyen agregar un nuevo sufijo (o aumentar la frecuencia de uno existente) y escoger un sufijo aleatorio.

Agregar un nuevo sufijo es igual de fácil para la implementación de lista o el histograma. Escoger un elemento aleatorio de una lista es fácil; escoger del histograma es más díficil de hacer de manera eficiente (ver Ejercicio 13.7).

Hasta ahora hemos estado hablando principalmente sobre la facilidad de la implementación, pero hay otros factores a considerar al escoger estructuras de datos. Uno es el tiempo de ejecución. A veces hay una razón teórica para esperar que una estructura de datos sea más rápida que otra; por ejemplo, mencioné que el operador `in` es más rápido para diccionarios que para listas, al menos cuando el número de elementos es grande.

Sin embargo, a menudo no sabes de antemano cuál implementación será más rápida. Una opción es implementar ambas y ver cuál es mejor. Este enfoque se llama **evaluación comparativa** (en inglés, *benchmarking*). Una alternativa práctica es escoger la estructura de datos que sea la más fácil de implementar, y luego ver si es lo suficientemente rápida para la aplicación en cuestión. Si es así, no hay necesidad de seguir. Si no, hay herramientas, como el módulo `profile`, que puede identificar los lugares en un programa que toman la mayor parte del tiempo.

El otro factor a considerar es el espacio de almacenamiento. Por ejemplo, usar un histograma para la colección de sufijos podría ocupar menos espacio porque solo tienes que almacenar cada palabra una vez, sin importar cuántas veces aparezca en el texto. En algunos casos, ahorrar espacio puede también hacer que tu programa se ejecute más rápido, y en el caso extremo, tu programa podría no ejecutarse en absoluto si te quedas sin memoria. Sin embargo, para muchas aplicaciones el espacio es una consideración secundaria después del tiempo de ejecución.

Una útlima reflexión: en esta discusión, he insinuado que deberíamos usar una estructura de datos tanto para el análisis como para la generación. Pero dado que estas son fases separadas, sería posible también usar una estructura para el análisis y luego convertirla a otra estructura para la generación. Esto sería una ganancia neta si el tiempo ahorrado durante la generación excediera al tiempo ocupado en la conversión.

13.10. Depuración

Cuando estés depurando un programa, y especialmente si estás trabajando en un error de programación difícil, hay cinco cosas para probar:

Lectura: Examina tu código, léelo de nuevo a ti mismo y verifica que dice lo que querías decir.

Ejecución: Experimenta haciendo cambios y ejecutando versiones diferentes. A menudo, si muestras en pantalla lo correcto en el lugar correcto del programa, el problema se vuelve obvio, pero a veces tienes que construir andamiaje.

Rumiación: ¡Tómate un tiempo para pensar! ¿Qué tipo de error es: de sintaxis, de tiempo de ejecución o semántico? ¿Qué información puedes obtener a partir de los mensajes de error o de la salida del programa? ¿Qué tipo de error podría causar el problema que estás viendo? ¿Qué cambiaste últimamente, antes de que el problema apareciera?

Patito de goma: Si le explicas el problema a alguien más, a veces encuentras la respuesta antes de terminar la pregunta. A menudo no necesitas a la otra persona; podrías simplemente hablarle a un patito de goma. Y ese es el origen de la conocida estrategia

llamada **método de depuración del patito de goma** (en inglés, *rubber duck debugging*). No estoy inventando; ver `https://en.wikipedia.org/wiki/Rubber_duck_debugging`.

Retroceso: En algún punto, lo mejor que se puede hacer es retroceder, deshacer los cambios recientes, hasta que regreses a un programa que funcione y que entiendas. Luego puedes comenzar a reconstruir.

Los programadores principiantes a veces se atascan en una de estas actividades y olvidan las otras. Cada actividad viene con su propio modo de fallo.

Por ejemplo, leer tu código podría ayudar si el problema es un error tipográfico, pero no si el problema es un malentendido conceptual. Si no entiendes lo que tu programa hace, puedes leerlo 100 veces y nunca ver el error, porque el error está en tu cabeza.

Ejecutar experimentos puede ayudar, especialmente si ejecutas pruebas pequeñas y simples. Pero si ejecutas experimentos sin pensar ni leer tu código, podrías caer en un patrón que yo llamo "programación de camino aleatorio", que es el proceso de hacer cambios aleatorios hasta que el programa haga lo correcto. No hace falta decir que la programación de camino aleatorio puede tomar mucho tiempo.

Tienes que tomarte el tiempo de pensar. La depuración es como una ciencia experimental. Deberías tener al menos una hipótesis acerca de cuál es el problema. Si hay dos o más posibilidades, intenta pensar en una prueba que eliminaría una de ellas.

Sin embargo, incluso las mejores técnicas de depuración fallarán si hay muchos errores, o si el código que intentas arreglar es muy grande y complicado. A veces la mejor opción es retroceder, simplificando el programa hasta que obtengas algo que funcione y que entiendas.

Los programadores principiantes son a menudo reacios a retroceder porque no pueden soportar eliminar una línea de códiigo (incluso si es incorrecta). Si te hace sentir mejor, copia tu programa en otro archivo antes de comenzar a recortarlo. Luego puedes volver a copiar los pedazos uno a la vez.

Encontrar un error de programación difícil requiere lectura, ejecución, rumiación y a veces retroceso. Si te atascas en una de estas actividades, intenta las otras.

13.11. Glosario

determinista: Dicho de un programa que hace lo mismo cada vez que se ejecuta, dadas las mismas entradas.

pseudoaleatorio: Dicho de una secuencia de números que aparenta ser aleatorio, pero se genera por un programa determinista.

valor por defecto: El valor dado a un parámetro opcional si no se le entrega un argumento.

anular: Reemplazar un valor por defecto con un argumento.

evaluación comparativa: El proceso de escoger entre estructuras de datos implementando alternativas y probándolas en una muestra de posibles entradas.

método de depuración del patito de goma: Depurar explicando tu problema a un objeto inanimado tal como un patito de goma. Articular el problema puede ayudarte a resolverlo, incluso si el patito de goma no sabe Python.

13.12. Ejercicios

Ejercicio 13.9. *El "rango" de una palabra es su posición en una lista de palabras ordenadas por frecuencia: la palabra más común tiene rango 1, la segunda más común tiene rango 2, etc.*

La ley de Zipf describe una relación entre los rangos y frecuencias de palabras en lenguajes naturales (`http://en.wikipedia.org/wiki/Zipf's_law`). Específicamente, predice que la frecuencia, f, de cada palabra con rango r es:

$$f = cr^{-s}$$

donde s y c son parámetros que dependen del lenguaje y el texto. Si tomas el logaritmo de ambos lados de esta ecuación, obtienes:

$$\log f = \log c - s \log r$$

Entonces, si graficas $\log f$ versus $\log r$, deberías obtener una línea recta con pendiente $-s$ e intercepto $\log c$.

Escribe un programa que lea un texto de un archivo, cuente las frecuencias de palabras e imprima una línea para cada palabra, en orden de frecuencia descendiente, con $\log f$ y $\log r$. Usa el progmara de gráficos de tu elección para graficar los resultados y verificar si forman una línea recta. ¿Puedes estimar el valor de s?

Solución: `http://thinkpython2.com/code/zipf.py`. Para ejecutar mi solución, necesitas el módulo de gráficos `matplotlib`*. Si instalaste Anaconda, ya tienes* `matplotlib`*; de lo contrario, quizás tengas que instalarlo.*

Capítulo 14

Files

This chapter introduces the idea of "persistent" programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

14.1. Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off.

Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, `pickle`, that makes it easy to store program data.

14.2. Reading and writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM. We saw how to open and read a file in Section 9.1.

To write a file, you have to open it with mode `'w'` as a second parameter:

```
>>> fout = open('output.txt', 'w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created.

`open` returns a file object that provides methods for working with the file. The `write` method puts data into the file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call `write` again, it adds the new data to the end of the file.

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

When you are done writing, you should close the file.

```
>>> fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

14.3. Format operator

The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with `str`:

```
>>> x = 52
>>> fout.write(str(x))
```

An alternative is to use the **format operator**, `%`. When applied to integers, `%` is the modulus operator. But when the first operand is a string, `%` is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string.

For example, the format sequence `'%d'` means that the second operand should be formatted as a decimal integer:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string `'42'`, which is not to be confused with the integer value `42`.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.

The following example uses `'%d'` to format an integer, `'%g'` to format a floating-point number, and `'%s'` to format a string:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

In the first example, there aren't enough elements; in the second, the element is the wrong type.

For more information on the format operator, see `https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting`. A more powerful alternative is the string format method, which you can read about at `https://docs.python.org/3/library/stdtypes.html#str.format`.

14.4. Filenames and paths

Files are organized into **directories** (also called "folders"). Every running program has a "current directory", which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories ("os" stands for "operating system"). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` stands for "current working directory". The result in this example is `/home/dinsdale`, which is the home directory of a user named `dinsdale`.

A string like `'/home/dinsdale'` that identifies a file or directory is called a **path**.

A simple filename, like `memo.txt` is also considered a path, but it is a **relative path** because it relates to the current directory. If the current directory is `/home/dinsdale`, the filename `memo.txt` would refer to `/home/dinsdale/memo.txt`.

A path that begins with `/` does not depend on the current directory; it is called an **absolute path**. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` provides other functions for working with filenames and paths. For example, `os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
            walk(path)
```

`os.path.join` takes a directory and a file name and joins them into a complete path.

The `os` module provides a function called `walk` that is similar to this one but more versatile. As an exercise, read the documentation and use it to print the names of the files in a given directory and its subdirectories. You can download my solution from `http://thinkpython2.com/code/walk.py`.

14.5. Catching exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like `os.path.exists` and `os.path.isfile`, but it would take a lot of time and code to check all the possibilities (if "`Errno 21`" is any indication, there are at least 21 things that can go wrong).

It is better to go ahead and try—and deal with problems if they happen—which is exactly what the `try` statement does. The syntax is similar to an `if...else` statement:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python starts by executing the `try` clause. If all goes well, it skips the `except` clause and proceeds. If an exception occurs, it jumps out of the `try` clause and runs the `except` clause.

Handling an exception with a `try` statement is called **catching** an exception. In this example, the `except` clause prints an error message that is not very helpful. In general, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully.

14.6. Databases

A **database** is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.

The module `dbm` provides an interface for creating and updating database files. As an example, I'll create a database that contains captions for image files.

Opening a database is similar to opening other files:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

The mode `'c'` means that the database should be created if it doesn't already exist. The result is a database object that can be used (for most operations) like a dictionary.

When you create a new item, `dbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, `dbm` reads the file:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

The result is a **bytes object**, which is why it begins with `b`. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it.

If you make another assignment to an existing key, `dbm` replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like `keys` and `items`, don't work with database objects. But iteration with a `for` loop works:

```
for key in db:
    print(key, db[key])
```

As with other files, you should close the database when you are done:

```
>>> db.close()
```

14.7. Pickling

A limitation of `dbm` is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The `pickle` module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

`pickle.dumps` takes an object as a parameter and returns a string representation (`dumps` is short for "dump string"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for `pickle` to interpret. `pickle.loads` ("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object.

You can use `pickle` to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called `shelve`.

14.8. Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command `ls -l` normally displays the contents of the current directory in long format. You can launch `ls` with `os.popen`[1]:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

[1] `popen` is deprecated now, which means we are supposed to stop using it and start using the `subprocess` module. But for simple cases, I find `subprocess` more complicated than necessary. So I am going to keep using `popen` until they take it away.

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print(stat)
None
```

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

For example, most Unix systems provide a command called `md5sum` that reads the contents of a file and computes a "checksum". You can read about MD5 at `http://en.wikipedia.org/wiki/Md5`. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run `md5sum` from Python and get the result:

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print(res)
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

14.9. Writing modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named `wc.py` with the following code:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

The module object provides `linecount`:

```
>>> wc.linecount('wc.py')
7
```

So that's how you write modules in Python.

The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn't run them.

Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` is a built-in variable that is set when the program starts. If the program is running as a script, `__name__` has the value `'__main__'`; in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named `wc.py` and run it as a script. Then run the Python interpreter and `import wc`. What is the value of `__name__` when the module is being imported?

Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function `reload`, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

14.10. Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2  3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>> print(repr(s))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies can cause problems.

For most systems, there are applications to convert from one format to another. You can find them (and read more about this issue) at `http://en.wikipedia.org/wiki/Newline`. Or, of course, you could write one yourself.

14.11. Glossary

persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.

format operator: An operator, `%`, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

format string: A string, used with the format operator, that contains format sequences.

format sequence: A sequence of characters in a format string, like `%d`, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the `try` and `except` statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

bytes object: An object similar to a string.

shell: A program that allows users to type commands and then executes them by starting other programs.

pipe object: An object that represents a running program, allowing a Python program to run commands and read the results.

14.12. Exercises

Ejercicio 14.1. *Write a function called* `sed` *that takes as arguments a pattern string, a replacement string, and two filenames; it should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the file, it should be replaced with the replacement string.*

If an error occurs while opening, reading, writing or closing files, your program should catch the exception, print an error message, and exit. Solution: `http://thinkpython2.com/code/sed.py`.

Ejercicio 14.2. *If you download my solution to Exercise* **??** *from* `http://thinkpython2.com/code/anagram_sets.py`, *you'll see that it creates a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example,* `'opst'` *maps to the list* `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Write a module that imports `anagram_sets` *and provides two new functions:* `store_anagrams` *should store the anagram dictionary in a "shelf";* `read_anagrams` *should look up a word and return a list of its anagrams. Solution:* `http://thinkpython2.com/code/anagram_db.py`.

Ejercicio 14.3. *In a large collection of MP3 files, there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for duplicates.*

1. *Write a program that searches a directory and all of its subdirectories, recursively, and returns a list of complete paths for all files with a given suffix (like* `.mp3`*). Hint:* `os.path` *provides several useful functions for manipulating file and path names.*

2. *To recognize duplicates, you can use* `md5sum` *to compute a "checksum" for each files. If two files have the same checksum, they probably have the same contents.*

3. *To double-check, you can use the Unix command* `diff`.

Solution: `http://thinkpython2.com/code/find_duplicates.py`.

Capítulo 15

Classes and objects

At this point you know how to use functions to organize code and built-in types to organize data. The next step is to learn "object-oriented programming", which uses programmer-defined types to organize both code and data. Object-oriented programming is a big topic; it will take a few chapters to get there.

Code examples from this chapter are available from `http://thinkpython2.com/code/Point1.py`; solutions to the exercises are available from `http://thinkpython2.com/code/Point1_soln.py`.

15.1. Programmer-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0, 0)$ represents the origin, and (x, y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

A programmer-defined type is also called a **class**. A class definition looks like this:

```
class Point:
    """Represents a point in 2-D space."""
```

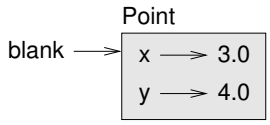

Figura 15.1: Object diagram.

The header indicates that the new class is called `Point`. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a **class object**.

```
>>> Point
<class '__main__.Point'>
```

Because `Point` is defined at the top level, its "full name" is `__main__.Point`.

The class object is like a factory for creating objects. To create a Point, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to `blank`.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix `0x` means that the following number is in hexadecimal).

Every object is an instance of some class, so "object" and "instance" are interchangeable. But in this chapter I use "instance" to indicate that I am talking about a programmer-defined type.

15.2. Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.whitespace`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb.

Figure 15.1 is a state diagram that shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**.

The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

You can read the value of an attribute using the same syntax:

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

The expression `blank.x` means, "Go to the object `blank` refers to and get the value of `x`." In the example, we assign that value to a variable named `x`. There is no conflict between the variable `x` and the attribute `x`.

You can use dot notation as part of any expression. For example:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

As an exercise, write a function called `distance_between_points` that takes two Points as arguments and returns the distance between them.

15.3. Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition:

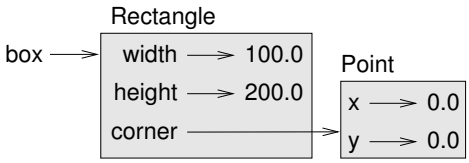

Figura 15.2: Object diagram.

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

The docstring lists the attributes: `width` and `height` are numbers; `corner` is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression `box.corner.x` means, "Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`."

Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

15.4. Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Here is an example that passes `box` as an argument and assigns the resulting Point to `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5. Objects are mutable

You can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Inside the function, `rect` is an alias for `box`, so when the function modifies `rect`, `box` changes.

As an exercise, write a function named `move_rectangle` that takes a Rectangle and two numbers named `dx` and `dy`. It should change the location of the rectangle by adding `dx` to the `x` coordinate of `corner` and adding `dy` to the `y` coordinate of `corner`.

15.6. Copying

Aliasing can make a program difficult to read because changes in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The `copy` module contains a function called `copy` that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` and `p2` contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
```

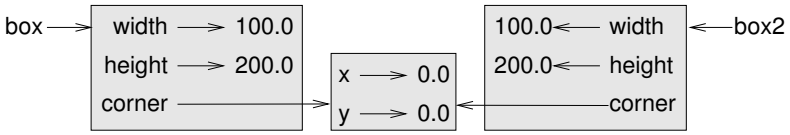

Figura 15.3: Object diagram.

```
>>> p1 == p2
False
```

The `is` operator indicates that `p1` and `p2` are not the same object, which is what we expected. But you might have expected `==` to yield `True` because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the `==` operator is the same as the `is` operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

If you use `copy.copy` to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Figure 15.3 shows what the object diagram looks like. This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

For most applications, this is not what you want. In this example, invoking `grow_rectangle` on one of the Rectangles would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module provides a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` and `box` are completely separate objects.

As an exercise, write a version of `move_rectangle` that creates and returns a new Rectangle instead of modifying the old one.

15.7. Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<class '__main__.Point'>
```

You can also use `isinstance` to check whether an object is an instance of a class:

```
>>> isinstance(p, Point)
True
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

You can also use a `try` statement to see if the object has the attributes you need:

```
try:
    x = p.x
except AttributeError:
    x = 0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in Section 17.9.

15.8. Glossary

class: A programmer-defined type. A class definition creates a new class object.

class object: An object that contains information about a programmer-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

instantiate: To create a new object.

attribute: One of the named values associated with an object.

embedded object: An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

15.9. Exercises

Ejercicio 15.1. *Write a definition for a class named* `Circle` *with attributes* `center` *and* `radius`*, where* `center` *is a Point object and radius is a number.*

Instantiate a Circle object that represents a circle with its center at $(150, 100)$ *and radius 75.*

Write a function named `point_in_circle` *that takes a Circle and a Point and returns True if the Point lies in or on the boundary of the circle.*

Write a function named `rect_in_circle` *that takes a Circle and a Rectangle and returns True if the Rectangle lies entirely in or on the boundary of the circle.*

Write a function named `rect_circle_overlap` *that takes a Circle and a Rectangle and returns True if any of the corners of the Rectangle fall inside the circle. Or as a more challenging version, return True if any part of the Rectangle falls inside the circle.*

Solution: `http://thinkpython2.com/code/Circle.py`.

Ejercicio 15.2. *Write a function called* `draw_rect` *that takes a Turtle object and a Rectangle and uses the Turtle to draw the Rectangle. See Chapter 4 for examples using Turtle objects.*

Write a function called `draw_circle` *that takes a Turtle and a Circle and draws the Circle.*

Solution: `http://thinkpython2.com/code/draw.py`.

Capítulo 16

Classes and functions

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results. In this chapter I also present "functional programming style" and two new program development plans.

Code examples from this chapter are available from `http://thinkpython2.com/code/Time1.py`. Solutions to the exercises are at `http://thinkpython2.com/code/Time1_soln.py`.

16.1. Time

As another example of a programmer-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

The state diagram for the `Time` object looks like Figure 16.1.

As an exercise, write a function called `print_time` that takes a Time object and prints it in the form `hour:minute:second`. Hint: the format sequence `'%.2d'` prints an integer using at least two digits, including a leading zero if necessary.

Write a boolean function called `is_after` that takes two Time objects, `t1` and `t2`, and returns `True` if `t1` follows `t2` chronologically and `False` otherwise. Challenge: don't use an `if` statement.

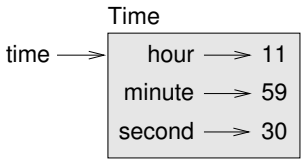

Figura 16.1: Object diagram.

16.2. Pure functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of `add_time`:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

The function creates a new `Time` object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value.

To test this function, I'll create two Time objects: `start` contains the start time of a movie, like *Monty Python and the Holy Grail*, and `duration` contains the run time of the movie, which is one hour 35 minutes.

`add_time` figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, `10:80:00` might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

16.3. Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

`increment`, which adds a given number of seconds to a `Time` object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if `seconds` is much greater than sixty?

In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the `if` statements with `while` statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of `increment` that doesn't contain any loops.

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

As an exercise, write a "pure" version of `increment` that creates and returns a new Time object rather than modifying the parameter.

16.4. Prototyping versus planning

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60 (see `http://en.wikipedia.org/wiki/Sexagesimal`.)! The `second` attribute is the "ones column", the `minute` attribute is the "sixties column", and the `hour` attribute is the "thirty-six hundreds column".

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is a function that converts an integer to a Time (recall that `divmod` divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that `time_to_int(int_to_time(x)) == x` for many values of `x`. This is an example of a consistency check.

Once you are convinced they are correct, you can use them to rewrite `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite `increment` using `time_to_int` and `int_to_time`.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`time_to_int` and `int_to_time`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

16.5. Debugging

A Time object is well-formed if the values of `minute` and `second` are between 0 and 60 (including 0 but not 60) and if `hour` is positive. `hour` and `minute` should be integral values, but we might allow `second` to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like `valid_time` that takes a Time object and returns `False` if it violates an invariant:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Or you could use an **assert statement**, which checks a given invariant and raises an exception if it fails:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

`assert` statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

16.6. Glossary

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

designed development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are void; that is, they return `None`.

functional programming style: A style of program design in which the majority of functions are pure.

invariant: A condition that should always be true during the execution of a program.

assert statement: A statement that check a condition and raises an exception if it fails.

16.7. Exercises

Code examples from this chapter are available from `http://thinkpython2.com/code/Time1.py`; solutions to the exercises are available from `http://thinkpython2.com/code/Time1_soln.py`.

Ejercicio 16.1. *Write a function called* `mul_time` *that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.*

Then use `mul_time` *to write a function that takes a Time object that represents the finishing time in a race, and a number that represents the distance, and returns a Time object that represents the average pace (time per mile).*

Ejercicio 16.2. *The* `datetime` *module provides* `time` *objects that are similar to the Time objects in this chapter, but they provide a rich set of methods and operators. Read the documentation at http://docs.python.org/3/library/datetime.html.*

1. *Use the* `datetime` *module to write a program that gets the current date and prints the day of the week.*

2. *Write a program that takes a birthday as input and prints the user's age and the number of days, hours, minutes and seconds until their next birthday.*

3. *For two people born on different days, there is a day when one is twice as old as the other. That's their Double Day. Write a program that takes two birth dates and computes their Double Day.*

4. *For a little more challenge, write the more general version that computes the day when one person is n times older than the other.*

Solution: http://thinkpython2.com/code/double.py

Capítulo 17

Classes and methods

Although we are using some of Python's object-oriented features, the programs from the last two chapters are not really object-oriented because they don't represent the relationships between programmer-defined types and the functions that operate on them. The next step is to transform those functions into methods that make the relationships explicit.

Code examples from this chapter are available from `http://thinkpython2.com/code/Time2.py`, and solutions to the exercises are in `http://thinkpython2.com/code/Point2_soln.py`.

17.1. Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support object-oriented programming, which has these defining characteristics:

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.

For example, the `Time` class defined in Chapter 16 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes in Chapter 15 correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in `Time1.py` there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument.

This observation is the motivation for **methods**; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

Methods are semantically the same as functions, but there are two syntactic differences:

- Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.
- The syntax for invoking a method is different from the syntax for calling a function.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

17.2. Printing objects

In Chapter 16, we defined a class named `Time` and in Section 16.1, you wrote a function named `print_time`:

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a `Time` object as an argument:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)
09:45:00
```

In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method. `start` is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, `print_time(start)`, suggests that the function is the active agent. It says something like, "Hey `print_time`! Here's an object for you to print."
- In object-oriented programming, the objects are the active agents. A method invocation like `start.print_time()` says "Hey `start`! Please print yourself."

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite `time_to_int` (from Section 16.4) as a method. You might be tempted to rewrite `int_to_time` as a method, too, but that doesn't really make sense because there would be no object to invoke it on.

17.3. Another example

Here's a version of `increment` (from Section 16.3) rewritten as a method:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

This version assumes that `time_to_int` is written as a method. Also, note that it is a pure function, not a modifier.

Here's how you would invoke `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

The subject, `start`, gets assigned to the first parameter, `self`. The argument, `1337`, gets assigned to the second parameter, `seconds`.

This mechanism can be confusing, especially if you make an error. For example, if you invoke `increment` with two arguments, you get:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

By the way, a **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:

```
sketch(parrot, cage, dead=True)
```

`parrot` and `cage` are positional, and `dead` is a keyword argument.

17.4. A more complicated example

Rewriting `is_after` (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter `self` and the second parameter `other`:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

17.5. The init method

The init method (short for "initialization") is a special method that gets invoked when an object is instantiated. Its full name is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An init method for the `Time` class might look like this:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

It is common for the parameters of `__init__` to have the same names as the attributes. The statement

```
        self.hour = hour
```

stores the value of the parameter hour as an attribute of self.

The parameters are optional, so if you call Time with no arguments, you get the default values.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

If you provide two arguments, they override hour and minute.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.

As an exercise, write an init method for the Point class that takes x and y as optional parameters and assigns them to the corresponding attributes.

17.6. The __str__ method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object.

For example, here is a str method for Time objects:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you print an object, Python invokes the str method:

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

As an exercise, write a str method for the Point class. Create a Point object and print it.

17.7. Operator overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named __add__ for the Time class, you can use the + operator on Time objects.

Here is what the definition might look like:

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

And here is how you could use it:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

When you apply the + operator to Time objects, Python invokes `__add__`. When you print the result, Python invokes `__str__`. So there is a lot happening behind the scenes!

Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like `__add__`. For more details, see `http://docs.python.org/3/reference/datamodel.html#specialnames`.

As an exercise, write an `add` method for the Point class.

17.8. Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of `__add__` that checks the type of `other` and invokes either `add_time` or `increment`:

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

The built-in function `isinstance` takes a value and a class object, and returns `True` if the value is an instance of the class.

If `other` is a Time object, `__add__` invokes `add_time`. Otherwise it assumes that the parameter is a number and invokes `increment`. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments.

Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method `__radd__`, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

As an exercise, write an `add` method for Points that works with either a Point object or a tuple:

- If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.

17.9. Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.

Many of the functions we wrote for strings also work for other sequence types. For example, in Section 11.2 we used `histogram` to count the number of times each letter appears in a word.

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
```

```
        else:
            d[c] = d[c]+1
    return d
```

This function also works for lists, tuples, and even dictionaries, as long as the elements of `s` are hashable, so they can be used as keys in `d`.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that work with several types are called **polymorphic**. Polymorphism can facilitate code reuse. For example, the built-in function `sum`, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

Since Time objects provide an `add` method, they work with `sum`:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the function works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

17.10. Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr` (see Section 15.7).

Another way to access attributes is the built-in function `vars`, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

`print_attributes` traverses the dictionary and prints each attribute name and its corresponding value.

The built-in function `getattr` takes an object and an attribute name (as a string) and returns the attribute's value.

17.11. Interface and implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented.

For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include `time_to_int`, `is_after`, and `add_time`.

We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a `Time` object are `hour`, `minute`, and `second`.

As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like `is_after`, easier to write, but it makes other methods harder.

After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface.

But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don't have to change.

17.12. Glossary

object-oriented language: A language that provides features, such as programmer-defined types and methods, that facilitate object-oriented programming.

object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

method: A function that is defined inside a class definition and is invoked on instances of that class.

subject: The object a method is invoked on.

positional argument: An argument that does not include a parameter name, so it is not a keyword argument.

operator overloading: Changing the behavior of an operator like + so it works with a programmer-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.

17.13. Exercises

Ejercicio 17.1. *Download the code from this chapter from* http://thinkpython2.com/code/Time2.py. *Change the attributes of* `Time` *to be a single integer representing seconds since midnight. Then modify the methods (and the function* `int_to_time`*) to work with the new implementation. You should not have to modify the test code in* `main`*. When you are done, the output should be the same as before. Solution:* http://thinkpython2.com/code/Time2_soln.py.

Ejercicio 17.2. *This exercise is a cautionary tale about one of the most common, and difficult to find, errors in Python. Write a definition for a class named* `Kangaroo` *with the following methods:*

1. *An* `__init__` *method that initializes an attribute named* `pouch_contents` *to an empty list.*
2. *A method named* `put_in_pouch` *that takes an object of any type and adds it to* `pouch_contents`*.*
3. *A* `__str__` *method that returns a string representation of the Kangaroo object and the contents of the pouch.*

Test your code by creating two `Kangaroo` *objects, assigning them to variables named* `kanga` *and* `roo`*, and then adding* `roo` *to the contents of* `kanga`*'s pouch.*

Download http://thinkpython2.com/code/BadKangaroo.py*. It contains a solution to the previous problem with one big, nasty bug. Find and fix the bug.*

If you get stuck, you can download http://thinkpython2.com/code/GoodKangaroo.py*, which explains the problem and demonstrates a solution.*

Capítulo 18

Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that represent playing cards, decks of cards, and poker hands.

If you don't play poker, you can read about it at `http://en.wikipedia.org/wiki/Poker`, but you don't have to; I'll tell you what you need to know for the exercises.

Code examples from this chapter are available from `http://thinkpython2.com/code/Card.py`.

18.1. Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like `'Spade'` for suits and `'Queen'` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. In this context, "encode" means that we are going to define a mapping between numbers and suits, or between numbers and ranks. This kind of encoding is not meant to be a secret (that would be "encryption").

For example, this table shows the suits and the corresponding integer codes:

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

This code makes it easy to compare cards; because higher suits map to higher numbers, we can compare suits by comparing their codes.

The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

I am using the $\mapsto$ symbol to make it clear that these mappings are not part of the Python program. They are part of the program design, but they don't appear explicitly in the code.

The class definition for `Card` looks like this:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

As usual, the init method takes an optional parameter for each attribute. The default card is the 2 of Clubs.

To create a Card, you call `Card` with the suit and rank of the card you want.

```
queen_of_diamonds = Card(1, 12)
```

18.2. Class attributes

In order to print Card objects in a way that people can easily read, we need a mapping from the integer codes to the corresponding ranks and suits. A natural way to do that is with lists of strings. We assign these lists to **class attributes**:

```
# inside class Card:

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'Jack', 'Queen', 'King']

    def __str__(self):
        return '%s of %s' % (Card.rank_names[self.rank],
                             Card.suit_names[self.suit])
```

Variables like `suit_names` and `rank_names`, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object `Card`.

This term distinguishes them from variables like `suit` and `rank`, which are called **instance attributes** because they are associated with a particular instance.

Both kinds of attribute are accessed using dot notation. For example, in `__str__`, `self` is a Card object, and `self.rank` is its rank. Similarly, `Card` is a class object, and `Card.rank_names` is a list of strings associated with the class.

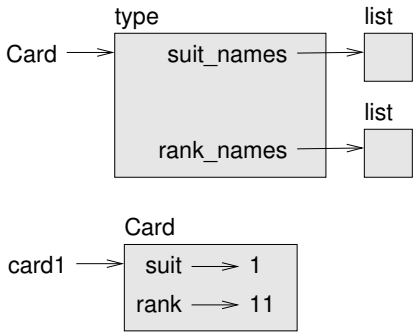

Figura 18.1: Object diagram.

Every card has its own `suit` and `rank`, but there is only one copy of `suit_names` and `rank_names`.

Putting it all together, the expression `Card.rank_names[self.rank]` means "use the attribute `rank` from the object `self` as an index into the list `rank_names` from the class `Card`, and select the appropriate string."

The first element of `rank_names` is `None` because there is no card with rank zero. By including `None` as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string `'2'`, and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

Figure 18.1 is a diagram of the `Card` class object and one Card instance. `Card` is a class object; its type is `type`. `card1` is an instance of `Card`, so its type is `Card`. To save space, I didn't draw the contents of `suit_names` and `rank_names`.

18.3. Comparing cards

For built-in types, there are relational operators (`<`, `>`, `==`, etc.) that compare values and determine when one is greater than, less than, or equal to another. For programmer-defined types, we can override the behavior of the built-in operators by providing a method named `__lt__`, which stands for "less than".

`__lt__` takes two parameters, `self` and `other`, and returns `True` if `self` is strictly less than `other`.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

The answer might depend on what game you are playing, but to keep things simple, we'll make the arbitrary choice that suit is more important, so all of the Spades outrank all of the Diamonds, and so on.

With that decided, we can write `__lt__`:

```
# inside class Card:

    def __lt__(self, other):
        # check the suits
        if self.suit < other.suit: return True
        if self.suit > other.suit: return False

        # suits are the same... check ranks
        return self.rank < other.rank
```

You can write this more concisely using tuple comparison:

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

As an exercise, write an `__lt__` method for Time objects. You can use tuple comparison, but you also might consider comparing integers.

18.4. Decks

Now that we have Cards, the next step is to define Decks. Since a deck is made up of cards, it is natural for each Deck to contain a list of cards as an attribute.

The following is a class definition for `Deck`. The init method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new Card with the current suit and rank, and appends it to `self.cards`.

18.5. Printing the deck

Here is a `__str__` method for `Deck`:

```
#inside class Deck:

    def __str__(self):
        res = []
```

```
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using the string method `join`. The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

Even though the result appears on 52 lines, it is one long string that contains newlines.

18.6. Add, remove, shuffle and sort

To deal cards, we would like a method that removes a card from the deck and returns it. The list method `pop` provides a convenient way to do that:

```
#inside class Deck:

    def pop_card(self):
        return self.cards.pop()
```

Since `pop` removes the *last* card in the list, we are dealing from the bottom of the deck.

To add a card, we can use the list method `append`:

```
#inside class Deck:

    def add_card(self, card):
        self.cards.append(card)
```

A method like this that uses another method without doing much work is sometimes called a **veneer**. The metaphor comes from woodworking, where a veneer is a thin layer of good quality wood glued to the surface of a cheaper piece of wood to improve the appearance.

In this case `add_card` is a "thin" method that expresses a list operation in terms appropriate for decks. It improves the appearance, or interface, of the implementation.

As another example, we can write a Deck method named `shuffle` using the function `shuffle` from the `random` module:

```
# inside class Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Don't forget to import `random`.

As an exercise, write a Deck method named `sort` that uses the list method `sort` to sort the cards in a `Deck`. `sort` uses the `__lt__` method we defined to determine the order.

18.7. Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. To define a new class that inherits from an existing class, you put the name of the existing class in parentheses:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

This definition indicates that `Hand` inherits from `Deck`; that means we can use methods like `pop_card` and `add_card` for Hands as well as Decks.

When a new class inherits from an existing one, the existing one is called the **parent** and the new class is called the **child**.

In this example, `Hand` inherits `__init__` from `Deck`, but it doesn't really do what we want: instead of populating the hand with 52 new cards, the init method for Hands should initialize `cards` with an empty list.

If we provide an init method in the `Hand` class, it overrides the one in the `Deck` class:

```
# inside class Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

When you create a Hand, Python invokes this init method, not the one in `Deck`.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

The other methods are inherited from `Deck`, so we can use `pop_card` and `add_card` to deal a card:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

A natural next step is to encapsulate this code in a method called `move_cards`:

```
#inside class Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` takes two arguments, a Hand object and the number of cards to deal. It modifies both `self` and `hand`, and returns `None`.

In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use `move_cards` for any of these operations: `self` can be either a Deck or a Hand, and `hand`, despite the name, can also be a `Deck`.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the design easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be spread across several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

18.8. Class diagrams

So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each Rectangle contains a reference to a Point, and each Deck contains references to many Cards. This kind of relationship is called **HAS-A**, as in, "a Rectangle has a Point."
- One class might inherit from another. This relationship is called **IS-A**, as in, "a Hand is a kind of a Deck."
- One class might depend on another in the sense that objects in one class take objects in the second class as parameters, or use objects in the second class as part of a computation. This kind of relationship is called a **dependency**.

A **class diagram** is a graphical representation of these relationships. For example, Figure 18.2 shows the relationships between `Card`, `Deck` and `Hand`.

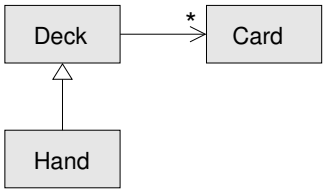

Figura 18.2: Class diagram.

The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck.

The standard arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a **multiplicity**; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like `52`, a range, like `5..7` or a star, which indicates that a Deck can have any number of Cards.

There are no dependencies in this diagram. They would normally be shown with a dashed arrow. Or if there are a lot of dependencies, they are sometimes omitted.

A more detailed diagram might show that a Deck actually contains a *list* of Cards, but built-in types like list and dict are usually not included in class diagrams.

18.9. Debugging

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked.

Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like `shuffle`, you might get the one defined in `Deck`, but if any of the subclasses override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If `Deck.shuffle` prints a message that says something like `Running Deck.shuffle`, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Here's an example:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class '__main__.Deck'>
```

So the `shuffle` method for this Hand is the one in `Deck`.

`find_defining_class` uses the `mro` method to get the list of class objects (types) that will be searched for methods. "MRO" stands for "method resolution order", which is the sequence of classes Python searches to "resolve" a method name.

Here's a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and PokerHand.

If you violate this rule, which is called the "Liskov substitution principle", your code will collapse like (sorry) a house of cards.

18.10. Data encapsulation

The previous chapters demonstrate a development plan we might call "object-oriented design". We identified objects we needed—like `Point`, `Rectangle` and `Time`—and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from Section 13.8, provides a good example. If you download my code from `http://thinkpython2.com/code/markov.py`, you'll see that it uses two global variables—`suffix_map` and `prefix`—that are read and written from several functions.

```
suffix_map = {}
prefix = ()
```

Because these variables are global, we can only run one analysis at a time. If we read two texts, their prefixes and suffixes would be added to the same data structures (which makes for some interesting generated text).

To run multiple analyses, and keep them separate, we can encapsulate the state of each analysis in an object. Here's what that looks like:

```
class Markov:

    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

Next, we transform the functions into methods. For example, here's `process_word`:

```
    def process_word(self, word, order=2):
        if len(self.prefix) < order:
            self.prefix += (word,)
            return
```

```
        try:
            self.suffix_map[self.prefix].append(word)
        except KeyError:
            # if there is no entry for this prefix, make one
            self.suffix_map[self.prefix] = [word]

        self.prefix = shift(self.prefix, word)
```

Transforming a program like this—changing the design without changing the behavior—is another example of refactoring (see Section 4.7).

This example suggests a development plan for designing objects and methods:

1. Start by writing functions that read and write global variables (when necessary).
2. Once you get the program working, look for associations between global variables and the functions that use them.
3. Encapsulate related variables as attributes of an object.
4. Transform the associated functions into methods of the new class.

As an exercise, download my Markov code from `http://thinkpython2.com/code/markov.py`, and follow the steps described above to encapsulate the global variables as attributes of a new class called `Markov`. Solution: `http://thinkpython2.com/code/markov2.py`.

18.11. Glossary

encode: To represent one set of values using another set of values by constructing a mapping between them.

class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.

instance attribute: An attribute associated with an instance of a class.

veneer: A method or function that provides a different interface to another function without doing much computation.

inheritance: The ability to define a new class that is a modified version of a previously defined class.

parent class: The class from which a child class inherits.

child class: A new class created by inheriting from an existing class; also called a "subclass".

IS-A relationship: A relationship between a child class and its parent class.

HAS-A relationship: A relationship between two classes where instances of one class contain references to instances of the other.

dependency: A relationship between two classes where instances of one class use instances of the other class, but do not store them as attributes.

class diagram: A diagram that shows the classes in a program and the relationships between them.

multiplicity: A notation in a class diagram that shows, for a HAS-A relationship, how many references there are to instances of another class.

data encapsulation: A program development plan that involves a prototype using global variables and a final version that makes the global variables into instance attributes.

18.12. Exercises

Ejercicio 18.1. *For the following program, draw a UML class diagram that shows these classes and the relationships among them.*

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong


class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
            self.pings = []
        else:
            self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

Ejercicio 18.2. *Write a Deck method called* `deal_hands` *that takes two parameters, the number of hands and the number of cards per hand. It should create the appropriate number of Hand objects, deal the appropriate number of cards per hand, and return a list of Hands.*

Ejercicio 18.3. *The following are the possible hands in poker, in increasing order of value and decreasing order of probability:*

pair: *two cards with the same rank*

two pair: *two pairs of cards with the same rank*

three of a kind: *three cards with the same rank*

straight: *five cards with ranks in sequence (aces can be high or low, so* `Ace-2-3-4-5` *is a straight and so is* `10-Jack-Queen-King-Ace`*, but* `Queen-King-Ace-2-3` *is not.)*

flush: *five cards with the same suit*

full house: *three cards with one rank, two cards with another*

four of a kind: *four cards with the same rank*

straight flush: *five cards in sequence (as defined above) and with the same suit*

The goal of these exercises is to estimate the probability of drawing these various hands.

1. *Download the following files from* `http://thinkpython2.com/code`*:*

   `Card.py` *: A complete version of the* `Card`, `Deck` *and* `Hand` *classes in this chapter.*

   `PokerHand.py` *: An incomplete implementation of a class that represents a poker hand, and some code that tests it.*

2. *If you run* `PokerHand.py`*, it deals seven 7-card poker hands and checks to see if any of them contains a flush. Read this code carefully before you go on.*

3. *Add methods to* `PokerHand.py` *named* `has_pair`, `has_twopair`*, etc. that return True or False according to whether or not the hand meets the relevant criteria. Your code should work correctly for "hands" that contain any number of cards (although 5 and 7 are the most common sizes).*

4. *Write a method named* `classify` *that figures out the highest-value classification for a hand and sets the* `label` *attribute accordingly. For example, a 7-card hand might contain a flush and a pair; it should be labeled "flush".*

5. *When you are convinced that your classification methods are working, the next step is to estimate the probabilities of the various hands. Write a function in* `PokerHand.py` *that shuffles a deck of cards, divides it into hands, classifies the hands, and counts the number of times various classifications appear.*

6. *Print a table of the classifications and their probabilities. Run your program with larger and larger numbers of hands until the output values converge to a reasonable degree of accuracy. Compare your results to the values at* `http://en.wikipedia.org/wiki/Hand_rankings`*.*

Solution: `http://thinkpython2.com/code/PokerHandSoln.py`*.*

Capítulo 19

The Goodies

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise.

Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that's more concise, readable or efficient, and sometimes all three.

19.1. Conditional expressions

We saw conditional statements in Section 5.4. Conditional statements are often used to choose one of two values; for example:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

This statement checks whether `x` is positive. If so, it computes `math.log`. If not, `math.log` would raise a ValueError. To avoid stopping the program, we generate a "NaN", which is a special floating-point value that represents "Not a Number".

We can write this statement more concisely using a **conditional expression**:

```
y = math.log(x) if x > 0 else float('nan')
```

You can almost read this line like English: "`y` gets log-`x` if `x` is greater than 0; otherwise it gets NaN".

Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of `factorial`:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We can rewrite it like this:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is the init method from `GoodKangaroo` (see Exercise 17.2):

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

We can rewrite this one like this:

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

19.2. List comprehensions

In Section 10.7 we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method `capitalize` to the elements, and returns a new list of strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a **list comprehension**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the `for` clause indicates what sequence we are traversing.

The syntax of a list comprehension is a little awkward because the loop variable, `s` in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of `t` that are upper case, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand.

But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

19.3. Generator expressions

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function `next` gets the next value from the generator:

```
>>> next(g)
0
>>> next(g)
1
```

When you get to the end of the sequence, `next` raises a StopIteration exception. You can also use a `for` loop to iterate through the values:

```
>>> for val in g:
...     print(val)
4
9
16
```

The generator object keeps track of where it is in the sequence, so the `for` loop picks up where `next` left off. Once the generator is exhausted, it continues to raise `StopIteration`:

```
>>> next(g)
StopIteration
```

Generator expressions are often used with functions like `sum`, `max`, and `min`:

```
>>> sum(x**2 for x in range(5))
30
```

19.4. `any` and `all`

Python provides a built-in function, `any`, that takes a sequence of boolean values and returns `True` if any of the values are `True`. It works on lists:

```
>>> any([False, False, True])
True
```

But it is often used with generator expressions:

```
>>> any(letter == 't' for letter in 'monty')
True
```

That example isn't very useful because it does the same thing as the `in` operator. But we could use `any` to rewrite some of the search functions we wrote in Section 9.3. For example, we could write `avoids` like this:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

The function almost reads like English, "`word` avoids `forbidden` if there are not any forbidden letters in `word`."

Using `any` with a generator expression is efficient because it stops immediately if it finds a `True` value, so it doesn't have to evaluate the whole sequence.

Python provides another built-in function, `all`, that returns `True` if every element of the sequence is `True`. As an exercise, use `all` to re-write `uses_all` from Section 9.3.

19.5. Sets

In Section 13.6 I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes `d1`, which contains the words from the document as keys, and `d2`, which contains the list of words. It returns a dictionary that contains the keys from `d1` that are not in `d2`.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

In all of these dictionaries, the values are `None` because we never use them. As a result, we waste some storage space.

Python provides another built-in type, called a `set`, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.

For example, set subtraction is available as a method called `difference` or as an operator, `-`. So we can rewrite `subtract` like this:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same.

Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to `has_duplicates`, from Exercise 10.7, that uses a dictionary:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns `True`.

Using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in `t` appears more than once, the set will be smaller than `t`. If there are no duplicates, the set will be the same size as `t`.

We can also use sets to do some of the exercises in Chapter 9. For example, here's a version of `uses_only` with a loop:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` checks whether all letters in `word` are in `available`. We can rewrite it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The `<=` operator checks whether one set is a subset of another, including the possibility that they are equal, which is true if all the letters in `word` appear in `available`.

As an exercise, rewrite `avoids` using sets.

19.6. Counters

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a **multiset**, a Counter is a natural way to represent a multiset.

Counter is defined in a standard module called `collections`, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable.

Unlike dictionaries, Counters don't raise an exception if you access an element that doesn't appear. Instead, they return 0:

```
>>> count['d']
0
```

We can use Counters to rewrite `is_anagram` from Exercise 10.6:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent.

Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, `most_common`, which returns a list of value-frequency pairs, sorted from most common to least:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

19.7. defaultdict

The `collections` module also provides `defaultdict`, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly.

When you create a defaultdict, you provide a function that's used to create new values. A function used to create objects is sometimes called a **factory**. The built-in functions that create lists, sets, and other types can be used as factories:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notice that the argument is `list`, which is a class object, not `list()`, which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist.

```
>>> t = d['new key']
>>> t
[]
```

The new list, which we're calling `t`, is also added to the dictionary. So if we modify `t`, the change appears in `d`:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using `defaultdict`. In my solution to Exercise **??**, which you can get from `http://thinkpython2.com/code/anagram_sets.py`, I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, `'opst'` maps to the list `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Here's the original code:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

This can be simplified using `setdefault`, which you might have used in Exercise 11.2:

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that's no big deal, but if the factory function is complicated, it might be.

We can avoid this problem and simplify the code using a `defaultdict`:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

My solution to Exercise 18.3, which you can download from `http://thinkpython2.com/code/PokerHandSoln.py`, uses `setdefault` in the function `has_straightflush`. This solution has the drawback of creating a `Hand` object every time through the loop, whether it is needed or not. As an exercise, rewrite it using a defaultdict.

19.8. Named tuples

Many simple objects are basically collections of related values. For example, the Point object defined in Chapter 15 contains two numbers, `x` and `y`. When you define a class like this, you usually start with an init method and a str method:

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from `namedtuple` is a class object:

```
>>> Point
<class '__main__.Point'>
```

`Point` automatically provides methods like `__init__` and `__str__` so you don't have to write them.

To create a Point object, you use the Point class as a function:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

The init method assigns the arguments to attributes using the names you provided. The str method prints a representation of the Point object and its attributes.

You can access the elements of the named tuple by name:

```
>>> p.x, p.y
(1, 2)
```

But you can also treat a named tuple as a tuple:

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that simple classes don't always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you could define a new class that inherits from the named tuple:

```
class Pointier(Point):
    # add more methods here
```

Or you could switch to a conventional class definition.

19.9. Gathering keyword args

In Section 12.4, we saw how to write a function that gathers its arguments into a tuple:

```
def printall(*args):
    print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don't have keywords):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

But the * operator doesn't gather keyword arguments:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but `kwargs` is a common choice. The result is a dictionary that maps keywords to values:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, ** to call a function:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Without the scatter operator, the function would treat `d` as a single positional argument, so it would assign `d` to `x` and complain because there's nothing to assign to `y`:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently used options.

19.10. Glossary

conditional expression: An expression that has one of two values, depending on a condition.

list comprehension: An expression with a `for` loop in square brackets that yields a new list.

generator expression: An expression with a `for` loop in parentheses that yields a generator object.

multiset: A mathematical entity that represents a mapping between the elements of a set and the number of times they appear.

factory: A function, usually passed as a parameter, used to create objects.

19.11. Exercises

Ejercicio 19.1. *The following is a function computes the binomial coefficient recursively.*

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1
    if n == 0:
        return 0

    res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
    return res
```

Rewrite the body of the function using nested conditional expressions.

One note: this function is not very efficient because it ends up computing the same values over and over. You could make it more efficient by memoizing (see Section 11.6). But you will find that it's harder to memoize if you write it using conditional expressions.

Apéndice A

Depuración

Cuando estés depurando, deberías distinguir entre los diferentes tipos de errores con el fin de rastrearlos de manera más rápida:

- Los errores de sintaxis son descubiertos por el intérprete cuando está traduciendo el código fuente a código byte. Indican que hay algo mal en la estructura del programa. Ejemplo: omitir el signo de dos puntos al final de una sentencia `def` genera el mensaje algo redundante `SyntaxError: invalid syntax`.
- Los errores de tiempo de ejecución son producidos por el intérprete si algo va mal mientras el programa se está ejecutando. La mayoría de los mensajes de error de tiempo de ejecución incluyen información acerca de dónde ocurrió el error y qué funciones se estaban ejecutando. Ejemplo: una recursividad infinita eventualmente causa el error de tiempo de ejecución "maximum recursion depth exceeded".
- Los errores semánticos son problemas que tiene un programa que se ejecuta sin producir mensajes de error pero sin hacer lo correcto. Ejemplo: una expresión puede que no sea evaluada en el orden que esperas, entregando un resultado incorrecto.

El primer paso en la depuración es averiguar con qué tipo de error estás lidiando. Aunque las siguientes secciones están organizadas por tipo de error, algunas técnicas son aplicables en más de una situación.

A.1. Errores de sintaxis

Los errores de sintaxis son generalmente fáciles de arreglar una vez que averiguas cuáles son. Desafortunadamente, los mensajes de error a menudo no son útiles. Los mensajes más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, de los cuales ninguno es muy informativo.

Por otra parte, el mensaje sí te dice dónde en el programa ocurrió el problema. En realidad, te dice dónde Python notó un problema, que no necesariamente es donde el error está. A veces el error está antes de la ubicación del mensaje de error, a menudo en la línea precedente.

Si estás construyendo el programa de manera incremental, deberías tener una buena idea acerca de dónde está el error. Estará en la última línea que agregaste.

Si estás copiando código de un libro, comienza comparando tu código con el código del libro muy cuidadosamente. Revisa cada carácter. Al mismo tiempo, recuerda que el libro podría estar mal, por tanto si ves algo que parece un error de sintaxis, puede serlo.

Aquí hay algunas maneras de evitar los errores de sintaxis más comunes:

1. Asegúrate de que no estás usando una palabra clave de Python para un nombre de variable.
2. Verifica que tienes un signo de dos puntos al final del encabezado de cada sentencia compuesta, incluyendo las sentencias `for`, `while`, `if` y `def`.
3. Asegúrate de que todas las cadenas en el código tengan comillas coincidentes. Asegúrate de que todas las comillas son "comillas rectas", no "comillas tipográficas".
4. Si tienes cadenas multilínea con comillas triples (simples o dobles), asegúrate de que has terminado la cadena de manera apropiada. Una cadena sin terminar puede causar un error `invalid token` al final de tu programa, o puede tratar la siguiente parte del programa como una cadena hasta que llega a la siguiente cadena. En el segundo caso, ¡podría no producir ningún mensaje de error!
5. Un operador de apertura no cerrado —`(`, `{` o `[`— hace que Python continúe con la línea siguiente como parte de la sentencia actual. Generalmente, ocurre un error casi inmediatamente en la línea siguient.
6. Revisa el clásico = en lugar de == dentro de un condicional.
7. Revisa la sangría para asegurarte de que esté alineada como se supone que debe. Python puede manejar el espacio y la tabulación, pero si los mezclas puede causar problemas. La mejor manera de evitar este problema es usar un editor de texto que sepa sobre Python y genere sangría consistente.
8. Si tienes caracteres no ASCII en el código (incluyendo cadenas y comentarios), podría causar un problema, aunque Python 3 generalmente maneja caracteres no ASCII. Ten cuidado si pegas texto de una página web u otra fuente.

Si nada funciona, pasa a la siguiente sección...

A.1.1. Sigo haciendo cambios y no hay diferencia.

Si el intérprete dice que hay un error y tú no lo ves, podría ser porque tú y el intérprete no están mirando el mismo código. Revisa tu entorno de programación para asegurarte de que el programa que estás editando es el que Python está intentando ejecutar.

Si no estás seguro, intenta poniendo un error de sintaxis obvio y deliberado al principio del programa. Ahora ejecútalo de nuevo. Si el intérprete no encuentra el nuevo error, no estás ejecutando el código nuevo.

Hay algunos posibles culpables:

- Editaste el archivo y olvidaste guardar los cambios antes de ejecutarlo de nuevo. Algunos entornos de programación hacen esto por ti, pero otros no.
- Cambiaste el nombre del archivo, pero todavía estás ejecutando el nombre antiguo.
- Algo en tu entorno de desarrollo está configurado de manera incorrecta.
- Si estás escribiendo un módulo y usando `import`, asegúrate de que no le das a tu módulo el mismo nombre que uno de los módulos estándar de Python.
- Si estás usando `import` para leer un módulo, recuerda que tienes que reiniciar el intérprete o usar `reload` para leer un archivo modificado. Si importas el módulo de nuevo, no hace nada.

Si te atascas y no puedes averiguar qué está pasando, una manera de abordarlo es comenzar de nuevo con un nuevo programa como "Hola, mundo" y asegurarte de que puedes obtener un programa conocido para ejecutar. Luego agrega gradualmente los pedazos del programa original al nuevo programa.

A.2. Errores de tiempo de ejecución

Una vez que tu programa está sintácticamente correcto, Python puede leerlo y al menos comenzar a ejecutarlo. ¿Qué podría salir mal?

A.2.1. Mi programa no hace absolutamente nada.

Este problema es más común cuando tu archivo se compone de funciones y clases pero en realidad no invoca una función para comenzar la ejecución. Esto puede ser intencional si solo planeas importar este módulo para proporcionar clases y funciones.

Si no es intentional, asegúrate de que hay una llamada a función en el programa, y asegúrate de que el flujo de ejecución lo alcanza (ver "Flujo de ejecución" más adelante).

A.2.2. Mi programa se congela.

Si un programa se detiene y parece estar haciendo nada, está "congelado". A menudo eso significa que está atrapado en un bucle infinito o una recursividad infinita.

- Si hay un bucle en particular del cual sospechas que es el problema, agrega una sentencia `print` inmediatamente antes del bucle que diga "entrando al bucle" y otro inmediatamente después que diga "saliendo del bucle".

  Ejecuta el programa. Si obtienes el primer mensaje y no el segundo, tienes un bucle infinito. Ve a la sección "Bucle infinito" de más adelante.

- La mayor parte del tiempo, una recursividad infinita causará que el programa se ejecute por un momento y luego produzca un error "RuntimeError: Maximum recursion depth exceeded". Si eso ocurre, ve a la sección "Recursividad infinita" de más adelante.

  Si no obtienes este error pero sospechas que hay un problema con una función recursiva o método recursivo, todavía puedes usar las térnicas de la sección "Recursividad infinita".

- Si ninguno de esos pasos funcioan, comienza a probar otros bucles y otras funciones y métodos recursivos.
- Si eso no funciona, entonces es posible que no entiendas el flujo de ejecución de tu programa. Ve a la sección "Flujo de ejecución" de más adelante.

Bucle infinito

Si crees que tienes un bicle infinito y crees que sabes qué bucle está causando el problema, agrega una sentencia `print` al final del bucle que imprima los valores de las variables en la condición y el valor de la condición.

Por ejemplo:

```
while x > 0 and y < 0 :
    # hacer algo a x
    # hacer algo a y

    print('x: ', x)
    print('y: ', y)
    print("condición: ", (x > 0 and y < 0))
```

Ahora cuando ejecutes el programa, verás tres líneas de salida para cada vez que se pase por el bucle. En el último paso por el bucle, la condición debería ser `False`. Si el bucle continúa, podrás ver los valores de `x` e `y`, y podrías averiguar por qué no se están actualizando correctamente.

Recursividad infinita

La mayor parte del tiempo, la recursividad infinita causa que el programa se ejecute por un momento y luego produzca un error `Maximum recursion depth exceeded`.

Si sospechas que una función está causando una recursividad infinita, asegúrate de que hay un caso base. Debería haber alguna condición que cause que la función devuelva sin hacer una invocación recursiva. Si no, necesitas volver a pensar el algoritmo e identificar un caso base.

Si hay un caso base pero el programa no parece estar alcanzándolo, agrega una sentencia `print` al principio de la función que imprima los parámetros. Ahora cuando ejecutes el programa, verás algunas líneas de salida cada vez que se invoca a la función, y verás los valores de los parámetros. Si los parámetros no se están moviendo hacia el caso base, obtendrás algunas ideas sobre por qué no ocurre.

Flujo de ejecución

Si no estás seguro de cómo se está moviendo el flujo de ejecución a través de tu programa, agrega sentencias `print` al principio de cada función con un mensaje como "entrando a la función `foo`", donde `foo` es el nombre de la función.

Ahora cuando ejecutes el programa, imprimirá una señal de cada función que se invoque.

A.2.3. Cuando ejecuto el programa obtengo una excepción.

Si algo va mal durante el tiempo de ejecución, Python imprime un mensaje que incluye el nombre de la excepción, la línea del programa donde ocurrió el problema y un rastreo.

El rastreo identifica la función que se está ejecutando actualmente y luego la función que la llamó, y luego la función que llamo a *aquella*, y así sucesivamente. En otras palabras, rastrea la secuencia de llamadas a función que te llevaron a donde estás, incluyendo el número de línea en tu archivo donde ocurrió cada llamada.

El primer paso es examinar el lugar en el programa donde ocurrió el error y ver si puedes averiguar lo que sucedió. Estos son algunos de los errores de tiempo de ejecución más comunes:

NameError: Estás intentando usar una variable que no existe en el entorno actual. Revisa si el nombre está bien escrito, o al menos de manera consistente. Y recuerda que las variables locales son locales; no puedes referirte a estas desde afuera de la función donde se definieron.

TypeError: Hay varias causas posibles:

- Estás intentando usar un valor de manera inapropiada. Ejemplo: indexar una cadena, lista o tupla con algo más que un entero.
- Hay una discordancia entre los ítems en una cadena de formato y los ítems pasados para una conversión. Eso puede ocurrir si el número de ítems no coincide o si se pidió una conversión no válida.
- Estás pasando el número equivocado de argumentos a una función. Para los métodos, mira la definición del método y verifica que el primer parámetro es `self`. Luego, mira la invocación del método; asegúrate de que estás invocando al método en un objeto con el tipo correcto y proporcionando los otros argumentos de manera correcta.

KeyError: Estás intentando acceder a un elemento de un diccionario usando una clave que el diccionario no contiene. Si las claves son cadenas, recuerda que las mayúsculas importan.

AttributeError: Estás intentando acceder a un atributo o método que no existe. ¡Revisa la ortografía! Puedes usar la función incorporada `vars` para hacer una lista de los atributos que sí existen.

Si un AttributeError indica que un objeto tiene `NoneType`, eso significa que es `None`. Entonces el problema no es el nombre de atributo, sino el objeto.

La razón por la cual el objeto es `None` podría ser que olvidaste devolver un valor desde una función; si llegas al final de una función ponienfo una sentencia `return`, devuelve `None`. Otra causa común es usar el resultado de un método de lista, como `sort`, que devuelve `None`.

IndexError: El índice que estás usando para acceder a una lista, cadena o tupla es mayor que su longitud menos uno. Inmediatamente antes del lugar del error, agrega una sentencia `print` para mostrar en pantalla el valor del índice y la longitud de la secuencia. ¿Tiene la secuencia el tamaño correcto? ¿Tiene el índice el valor correcto?

El depurador de Python (`pdb`, *Python debugger*) es útil para rastrear excepciones porque te permite examinar el estado del programa inmediatamente antes del error. Puedes leer sobre `pdb` en `https://docs.python.org/3/library/pdb.html`.

A.2.4. Agregué tantas sentencias `print` que me inundé con la salida.

Uno de los problemas al usar sentencias `print` para depurar es que puedes terminar enterrándote en la salida. Hay dos maneras de proceder: simplificar la salida o simplificar el programa.

Para simplificar la salida, puedes eliminar o poner como comentarios las sentencias `print` que no están ayudando, o combinarlas, o dar formato a la salida para que sea más fácil de entender.

Para simplificar el programa, hay varias cosas que puedes hacer. Primero, reduce la escala del problema en el cual está trabajando el programa. Por ejemplo, si estás buscando una lista, busca una lista *pequeña*. Si el programa toma entrada del usuario, dale la entrada más simple que cause el problema.

Segundo, limpia el programa. Elimina el código muerto y reorganiza el programa para hacerlo tan fácil de leer como sea posible. Por ejemplo, si sospechas que el problema está en una parte profundamente anidada del programa, intenta reescribir esa parte con una estructura más simple. Si sospechas de una función grande, intenta separarla en funciones más pequeñas y probarlas de manera separada.

A menudo el proceso de encontrar el caso de prueba mínimo te guía al error. Si encuentras que un programa funciona en una situación pero no en otra, eso te da una pisa sobre qué está pasando.

Del mismo modo, reescribir un pedazo de código puede ayudarte a encontrar errores sutiles. Si haces un cambio que crees que no debería afectar al programa, y sí afecta, eso te puede dar una pista.

A.3. Errores semánticos

De alguna manera, los errores semánticos son los más difíciles de depurar, porque el intérprete no proporciona información sobre qué está mal. Solo tú sabes lo que se supone que debe hacer el programa.

El primer paso es hacer una conexión entre el texto del programa y el comportamiento que ves. Necesitas una hipótesis sobre qué está haciendo realmente el programa. Una de las cosas que hace que eso sea difícil es que los computadores funcionan muy rápido.

A menudo desearás poder ralentizar el programa a velocidad humana, y con algunos depuradores puedes hacerlo. Pero el tiempo que toma insertar unas pocas sentencias `print` bien ubicadas es a menudo corto comparado con el de configurar el depurador, insertar y eliminar puntos de interrupción, y avanzar "paso a paso" en el programa hasta donde ocurre el error.

A.3.1. Mi programa no funciona.

Deberías hacerte estas preguntas:

- ¿Hay algo que se supone que el programa debe hacer pero que no parece estar ocurriendo? Encuentra la sección del código que realiza esa función y asegúrate de que se está ejecutando cuando crees que debería.

- ¿Ocurre algo que no debería? Encuentra código en tu programa que realiza esa función y ve si se está ejecutando cuando no debería.
- ¿Hay una sección de código produciendo un efecto que no es lo que esperabas? Asegúrate de que entiendes el código en cuestión, especialmente si involucra funciones o métodos de otros módulos de Python. Lee la documentación para las funciones que llamas. Pruébalas escribiendo casos de prueba simples y verificando los resultados.

Para programar, necesitas un modelo mental de cómo funcionan los programas. Si escribes un programa que no hace lo que esperas, muchas veces el problema no está en el programa; está en tu modelo mental.

La mejor manera de corregir tu modelo mental es separar el programa en sus componentes (generalmente las funciones y métodos) y probar cada componente de manera independiente. Una vez que encuentras la discrepancia entre tu modelo y la realidad, puedes resolver el problema.

Por supuesto, deberías estar construyendo y probando componentes a medida que desarrollas el programa. Si encuentras un problema, debería haber solo una pequeña cantidad de código nuevo que no se sabe que es correcto.

A.3.2. Tengo una expresión grande y fea, y no hace lo que yo espero.

Escribir expresiones complejas está bien mientras sean legibles, pero pueden ser difíciles de depurar. Muchas veces es una buena idea separar una expresión compleja en una serie de asignaciones a variables temporales.

Por ejemplo:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

Esto se puede reescribir como:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

La versión explícita es más fácil de leer porque los nombres de variable proporcionan documentación adicional, y es más fácil depurar porque puedes verificar los tipos de las variables intermedias y mostrar sus valores en pantalla.

Otro problema que puede ocurrir con las expresiones grandes es que el orden de evaluación puede que no sea lo que esperas. Por ejemplo, si estás traduciendo la expresión $\frac{x}{2\pi}$ a Python, podrías escribir:

```
y = x / 2 * math.pi
```

Eso no es correcto porque la multiplicación y la división tienen la misma prioridad y se evalúan de izquierda a derecha. Entonces esta expresión calcula $x\pi/2$.

Una buena manera de depurar expresiones es agregar paréntesis para hacer que el orden de evaluación sea explícito:

```
y = x / (2 * math.pi)
```

Siempre que no estés seguro del orden de evaluación, usa paréntesis. No solo estará correcto el programa (en el sentido de hacer lo que pretendías), también será más legible para otras personas que no han memorizado el orden de las operaciones.

A.3.3. Tengo una función que no devuelve lo que yo espero.

Si tienes una sentencia `return` con una expresión compleja, no tienes la posibilidad de imprimir el resultado antes de devolverlo. De nuevo, puedes usar una variable temporal. Por ejemplo, en lugar de:

```
return self.hands[i].removeMatches()
```

podrías escribir:

```
count = self.hands[i].removeMatches()
return count
```

Ahora tienes la oportunidad de mostrar en pantalla el valor de `count` antes de devolverlo.

A.3.4. De verdad me atasqué y necesito ayuda.

Primero, intenta alejarte del computador por algunos minutos. Los computadores emiten ondas que afectan al cerebro, causando estos síntomas:

- Frustración e ira.
- Creencias supersticiosas ("el computador me odia") y pensamiento mágico ("el programa solo funciona cuando uso mi gorra hacia atrás").
- Programación de camino aleatorio (el intento de programar escribiendo cada programa posible y escoger el que hace lo correcto).

Si te encuentras sufriendo alguno de estos síntomas, levántate y ve a dar un paseo. Cuando te hayas tranquilizado, piensa en el programa. ¿Qué está haciendo? ¿Cuáles son algunas posibles causas de aquel comportamiento? ¿Cuándo fue la última vez que tuviste un programa eficaz, y qué hiciste después?

A veces solo toma tiempo encontrar un error de programación. A menudo encuentro errores cuando estor lejos del computador y dejo vagar a mi mente. Algunos de los mejores lugares para encontrar errores son los trenes, las duchas y en la cama, justo antes de dormirte.

A.3.5. No, realmente necesito ayuda.

Sucede. Incluso los mejores programadores se atascan ocasionalmente. A veces trabajas en un programa tan largo que no puedes ver el error. Necesitas otro punto de vista.

Antes de traer a alguien más, asegúrate de tener todo preparado. Tu programa debería ser tan simple como sea posible, y deberías estar trabajando en la entrada más pequeña que causa el error. Deberías tener sentencias `print` en los lugares apropiados (y la salida que producen debería ser comprensible). Deberías entender el problema lo suficientemente bien como para describirlo de manera concisa.

Cuando traigas a alguien para que te ayude, asegúrate de darle la información que necesita:

- Si hay un mensaje de error, ¿cuál es y qué parte del programa indica?
- ¿Qué fue lo último que hiciste antes de que ocurriera este error? ¿Cuáles fueron las últimas líneas de código que escribiste o cuál es el nuevo caso de prueba que falla?

- ¿Qué has intentado hasta ahora y qué has aprendido?

Cuando encuentres el error, tómate un segundo para pensar sobre qué podrías haber hecho para encontrarlo de manera más rápida. La próxima vez que veas algo similar, serás capaz de entontrar el error con más rapidez.

Recuerda, la meta no solo es hacer que el programa funcione. La meta es aprender cómo hacer que el programa funcione.

Apéndice B

Analysis of Algorithms

> This appendix is an edited excerpt from *Think Complexity*, by Allen B. Downey, also published by O'Reilly Media (2012). When you are done with this book, you might want to move on to that one.

Analysis of algorithms is a branch of computer science that studies the performance of algorithms, especially their run time and space requirements. See `http://en.wikipedia.org/wiki/Analysis_of_algorithms`.

The practical goal of algorithm analysis is to predict the performance of different algorithms in order to guide design decisions.

During the 2008 United States Presidential Campaign, candidate Barack Obama was asked to perform an impromptu analysis when he visited Google. Chief executive Eric Schmidt jokingly asked him for "the most efficient way to sort a million 32-bit integers." Obama had apparently been tipped off, because he quickly replied, "I think the bubble sort would be the wrong way to go." See `http://www.youtube.com/watch?v=k4RRi_ntQc8`.

This is true: bubble sort is conceptually simple but slow for large datasets. The answer Schmidt was probably looking for is "radix sort" (`http://en.wikipedia.org/wiki/Radix_sort`)[1].

The goal of algorithm analysis is to make meaningful comparisons between algorithms, but there are some problems:

- The relative performance of the algorithms might depend on characteristics of the hardware, so one algorithm might be faster on Machine A, another on Machine B. The general solution to this problem is to specify a **machine model** and analyze the number of steps, or operations, an algorithm requires under a given model.
- Relative performance might depend on the details of the dataset. For example, some sorting algorithms run faster if the data are already partially sorted; other algorithms

[1] But if you get a question like this in an interview, I think a better answer is, "The fastest way to sort a million integers is to use whatever sort function is provided by the language I'm using. Its performance is good enough for the vast majority of applications, but if it turned out that my application was too slow, I would use a profiler to see where the time was being spent. If it looked like a faster sort algorithm would have a significant effect on performance, then I would look around for a good implementation of radix sort."

run slower in this case. A common way to avoid this problem is to analyze the **worst case** scenario. It is sometimes useful to analyze average case performance, but that's usually harder, and it might not be obvious what set of cases to average over.

- Relative performance also depends on the size of the problem. A sorting algorithm that is fast for small lists might be slow for long lists. The usual solution to this problem is to express run time (or number of operations) as a function of problem size, and group functions into categories depending on how quickly they grow as problem size increases.

The good thing about this kind of comparison is that it lends itself to simple classification of algorithms. For example, if I know that the run time of Algorithm A tends to be proportional to the size of the input, n, and Algorithm B tends to be proportional to n^2, then I expect A to be faster than B, at least for large values of n.

This kind of analysis comes with some caveats, but we'll get to that later.

B.1. Order of growth

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input: Algorithm A takes $100n + 1$ steps to solve a problem with size n; Algorithm B takes $n^2 + n + 1$ steps.

The following table shows the run time of these algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	100 010 001

At $n = 10$, Algorithm A looks pretty bad; it takes almost 10 times longer than Algorithm B. But for $n = 100$ they are about the same, and for larger values A is much better.

The fundamental reason is that for large values of n, any function that contains an n^2 term will grow faster than a function whose leading term is n. The **leading term** is the term with the highest exponent.

For Algorithm A, the leading term has a large coefficient, 100, which is why B does better than A for small n. But regardless of the coefficients, there will always be some value of n where $an^2 > bn$, for any values of a and b.

The same argument applies to the non-leading terms. Even if the run time of Algorithm A were $n + 1000000$, it would still be better than Algorithm B for sufficiently large n.

In general, we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems, there may be a **crossover point** where another algorithm is better. The location of the crossover point depends on the details of the algorithms, the inputs, and the hardware, so it is usually ignored for purposes of algorithmic analysis. But that doesn't mean you can forget about it.

If two algorithms have the same leading order term, it is hard to say which is better; again, the answer depends on the details. So for algorithmic analysis, functions with the same leading term are considered equivalent, even if they have different coefficients.

An **order of growth** is a set of functions whose growth behavior is considered equivalent. For example, $2n$, $100n$ and $n + 1$ belong to the same order of growth, which is written $O(n)$ in **Big-Oh notation** and often called **linear** because every function in the set grows linearly with n.

All functions with the leading term n^2 belong to $O(n^2)$; they are called **quadratic**.

The following table shows some of the orders of growth that appear most commonly in algorithmic analysis, in increasing order of badness.

Order of growth	Name
$O(1)$	constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

For the logarithmic terms, the base of the logarithm doesn't matter; changing bases is the equivalent of multiplying by a constant, which doesn't change the order of growth. Similarly, all exponential functions belong to the same order of growth regardless of the base of the exponent. Exponential functions grow very quickly, so exponential algorithms are only useful for small problems.

Ejercicio B.1. *Read the Wikipedia page on Big-Oh notation at* `http://en.wikipedia.org/wiki/Big_O_notation` *and answer the following questions:*

1. *What is the order of growth of $n^3 + n^2$? What about $1000000n^3 + n^2$? What about $n^3 + 1000000n^2$?*
2. *What is the order of growth of $(n^2 + n) \cdot (n + 1)$? Before you start multiplying, remember that you only need the leading term.*
3. *If f is in $O(g)$, for some unspecified function g, what can we say about $af + b$, where a and b are constants?*
4. *If f_1 and f_2 are in $O(g)$, what can we say about $f_1 + f_2$?*
5. *If f_1 is in $O(g)$ and f_2 is in $O(h)$, what can we say about $f_1 + f_2$?*
6. *If f_1 is in $O(g)$ and f_2 is $O(h)$, what can we say about $f_1 \cdot f_2$?*

Programmers who care about performance often find this kind of analysis hard to swallow. They have a point: sometimes the coefficients and the non-leading terms make a real difference. Sometimes the details of the hardware, the programming language, and the characteristics of the input make a big difference. And for small problems, order of growth is irrelevant.

But if you keep those caveats in mind, algorithmic analysis is a useful tool. At least for large problems, the "better" algorithm is usually better, and sometimes it is *much* better. The difference between two algorithms with the same order of growth is usually a constant factor, but the difference between a good algorithm and a bad algorithm is unbounded!

B.2. Analysis of basic Python operations

In Python, most arithmetic operations are constant time; multiplication usually takes longer than addition and subtraction, and division takes even longer, but these run times don't depend on the magnitude of the operands. Very large integers are an exception; in that case the run time increases with the number of digits.

Indexing operations—reading or writing elements in a sequence or dictionary—are also constant time, regardless of the size of the data structure.

A `for` loop that traverses a sequence or dictionary is usually linear, as long as all of the operations in the body of the loop are constant time. For example, adding up the elements of a list is linear:

```
total = 0
for x in t:
    total += x
```

The built-in function `sum` is also linear because it does the same thing, but it tends to be faster because it is a more efficient implementation; in the language of algorithmic analysis, it has a smaller leading coefficient.

As a rule of thumb, if the body of a loop is in $O(n^a)$ then the whole loop is in $O(n^{a+1})$. The exception is if you can show that the loop exits after a constant number of iterations. If a loop runs k times regardless of n, then the loop is in $O(n^a)$, even for large k.

Multiplying by k doesn't change the order of growth, but neither does dividing. So if the body of a loop is in $O(n^a)$ and it runs n/k times, the loop is in $O(n^{a+1})$, even for large k.

Most string and tuple operations are linear, except indexing and `len`, which are constant time. The built-in functions `min` and `max` are linear. The run-time of a slice operation is proportional to the length of the output, but independent of the size of the input.

String concatenation is linear; the run time depends on the sum of the lengths of the operands.

All string methods are linear, but if the lengths of the strings are bounded by a constant—for example, operations on single characters—they are considered constant time. The string method `join` is linear; the run time depends on the total length of the strings.

Most list methods are linear, but there are some exceptions:

- Adding an element to the end of a list is constant time on average; when it runs out of room it occasionally gets copied to a bigger location, but the total time for n operations is $O(n)$, so the average time for each operation is $O(1)$.
- Removing an element from the end of a list is constant time.
- Sorting is $O(n \log n)$.

Most dictionary operations and methods are constant time, but there are some exceptions:

- The run time of `update` is proportional to the size of the dictionary passed as a parameter, not the dictionary being updated.
- `keys`, `values` and `items` are constant time because they return iterators. But if you loop through the iterators, the loop will be linear.

The performance of dictionaries is one of the minor miracles of computer science. We will see how they work in Section B.4.

Ejercicio B.2. *Read the Wikipedia page on sorting algorithms at* `http://en.wikipedia.org/wiki/Sorting_algorithm` *and answer the following questions:*

1. *What is a "comparison sort?" What is the best worst-case order of growth for a comparison sort? What is the best worst-case order of growth for any sort algorithm?*
2. *What is the order of growth of bubble sort, and why does Barack Obama think it is "the wrong way to go?"*
3. *What is the order of growth of radix sort? What preconditions do we need to use it?*
4. *What is a stable sort and why might it matter in practice?*
5. *What is the worst sorting algorithm (that has a name)?*
6. *What sort algorithm does the C library use? What sort algorithm does Python use? Are these algorithms stable? You might have to Google around to find these answers.*
7. *Many of the non-comparison sorts are linear, so why does does Python use an $O(n \log n)$ comparison sort?*

B.3. Analysis of search algorithms

A **search** is an algorithm that takes a collection and a target item and determines whether the target is in the collection, often returning the index of the target.

The simplest search algorithm is a "linear search", which traverses the items of the collection in order, stopping if it finds the target. In the worst case it has to traverse the entire collection, so the run time is linear.

The `in` operator for sequences uses a linear search; so do string methods like `find` and `count`.

If the elements of the sequence are in order, you can use a **bisection search**, which is $O(\log n)$. Bisection search is similar to the algorithm you might use to look a word up in a dictionary (a paper dictionary, not the data structure). Instead of starting at the beginning and checking each item in order, you start with the item in the middle and check whether the word you are looking for comes before or after. If it comes before, then you search the first half of the sequence. Otherwise you search the second half. Either way, you cut the number of remaining items in half.

If the sequence has 1,000,000 items, it will take about 20 steps to find the word or conclude that it's not there. So that's about 50,000 times faster than a linear search.

Bisection search can be much faster than linear search, but it requires the sequence to be in order, which might require extra work.

There is another data structure, called a **hashtable** that is even faster—it can do a search in constant time—and it doesn't require the items to be sorted. Python dictionaries are implemented using hashtables, which is why most dictionary operations, including the `in` operator, are constant time.

B.4. Hashtables

To explain how hashtables work and why their performance is so good, I start with a simple implementation of a map and gradually improve it until it's a hashtable.

I use Python to demonstrate these implementations, but in real life you wouldn't write code like this in Python; you would just use a dictionary! So for the rest of this chapter, you have to imagine that dictionaries don't exist and you want to implement a data structure that maps from keys to values. The operations you have to implement are:

`add(k, v)`: Add a new item that maps from key `k` to value `v`. With a Python dictionary, `d`, this operation is written `d[k] = v`.

`get(k)`: Look up and return the value that corresponds to key `k`. With a Python dictionary, `d`, this operation is written `d[k]` or `d.get(k)`.

For now, I assume that each key only appears once. The simplest implementation of this interface uses a list of tuples, where each tuple is a key-value pair.

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` appends a key-value tuple to the list of items, which takes constant time.

`get` uses a `for` loop to search the list: if it finds the target key it returns the corresponding value; otherwise it raises a `KeyError`. So `get` is linear.

An alternative is to keep the list sorted by key. Then `get` could use a bisection search, which is $O(\log n)$. But inserting a new item in the middle of a list is linear, so this might not be the best option. There are other data structures that can implement `add` and `get` in log time, but that's still not as good as constant time, so let's move on.

One way to improve `LinearMap` is to break the list of key-value pairs into smaller lists. Here's an implementation called `BetterMap`, which is a list of 100 LinearMaps. As we'll see in a second, the order of growth for `get` is still linear, but `BetterMap` is a step on the path toward hashtables:

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())
```

```
    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` makes a list of `n` `LinearMaps`.

`find_map` is used by `add` and `get` to figure out which map to put the new item in, or which map to search.

`find_map` uses the built-in function `hash`, which takes almost any Python object and returns an integer. A limitation of this implementation is that it only works with hashable keys. Mutable types like lists and dictionaries are unhashable.

Hashable objects that are considered equivalent return the same hash value, but the converse is not necessarily true: two objects with different values can return the same hash value.

`find_map` uses the modulus operator to wrap the hash values into the range from 0 to `len(self.maps)`, so the result is a legal index into the list. Of course, this means that many different hash values will wrap onto the same index. But if the hash function spreads things out pretty evenly (which is what hash functions are designed to do), then we expect $n/100$ items per LinearMap.

Since the run time of `LinearMap.get` is proportional to the number of items, we expect BetterMap to be about 100 times faster than LinearMap. The order of growth is still linear, but the leading coefficient is smaller. That's nice, but still not as good as a hashtable.

Here (finally) is the crucial idea that makes hashtables fast: if you can keep the maximum length of the LinearMaps bounded, `LinearMap.get` is constant time. All you have to do is keep track of the number of items and when the number of items per LinearMap exceeds a threshold, resize the hashtable by adding more LinearMaps.

Here is an implementation of a hashtable:

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
```

```
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

`__init__` creates a `BetterMap` and initializes `num`, which keeps track of the number of items.

`get` just dispatches to `BetterMap`. The real work happens in `add`, which checks the number of items and the size of the `BetterMap`: if they are equal, the average number of items per LinearMap is 1, so it calls `resize`.

`resize` make a new `BetterMap`, twice as big as the previous one, and then "rehashes" the items from the old map to the new.

Rehashing is necessary because changing the number of LinearMaps changes the denominator of the modulus operator in `find_map`. That means that some objects that used to hash into the same LinearMap will get split up (which is what we wanted, right?).

Rehashing is linear, so `resize` is linear, which might seem bad, since I promised that `add` would be constant time. But remember that we don't have to resize every time, so `add` is usually constant time and only occasionally linear. The total amount of work to run `add` n times is proportional to n, so the average time of each `add` is constant time!

To see how this works, think about starting with an empty HashTable and adding a sequence of items. We start with 2 LinearMaps, so the first 2 adds are fast (no resizing required). Let's say that they take one unit of work each. The next add requires a resize, so we have to rehash the first two items (let's call that 2 more units of work) and then add the third item (one more unit). Adding the next item costs 1 unit, so the total so far is 6 units of work for 4 items.

The next `add` costs 5 units, but the next three are only one unit each, so the total is 14 units for the first 8 adds.

The next `add` costs 9 units, but then we can add 7 more before the next resize, so the total is 30 units for the first 16 adds.

After 32 adds, the total cost is 62 units, and I hope you are starting to see a pattern. After n adds, where n is a power of two, the total cost is $2n - 2$ units, so the average work per add is a little less than 2 units. When n is a power of two, that's the best case; for other values of n the average work is a little higher, but that's not important. The important thing is that it is $O(1)$.

Figure B.1 shows how this works graphically. Each block represents a unit of work. The columns show the total work for each add in order from left to right: the first two `adds` cost 1 unit each, the third costs 3 units, etc.

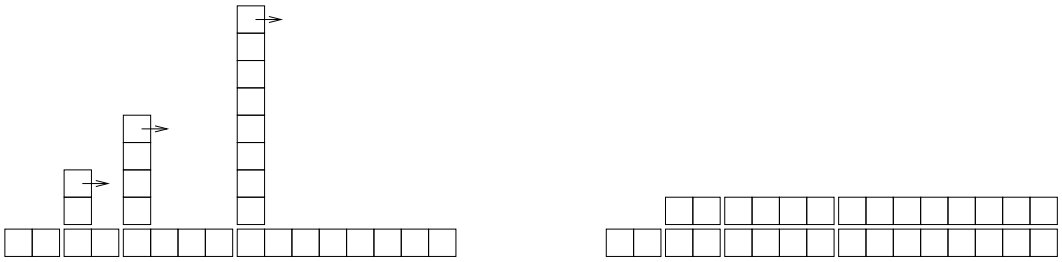

Figura B.1: The cost of a hashtable add.

The extra work of rehashing appears as a sequence of increasingly tall towers with increasing space between them. Now if you knock over the towers, spreading the cost of resizing over all adds, you can see graphically that the total cost after n adds is $2n - 2$.

An important feature of this algorithm is that when we resize the HashTable it grows geometrically; that is, we multiply the size by a constant. If you increase the size arithmetically—adding a fixed number each time—the average time per `add` is linear.

You can download my implementation of HashMap from `http://thinkpython2.com/code/Map.py`, but remember that there is no reason to use it; if you want a map, just use a Python dictionary.

B.5. Glossary

analysis of algorithms: A way to compare algorithms in terms of their run time and/or space requirements.

machine model: A simplified representation of a computer used to describe algorithms.

worst case: The input that makes a given algorithm run slowest (or require the most space).

leading term: In a polynomial, the term with the highest exponent.

crossover point: The problem size where two algorithms require the same run time or space.

order of growth: A set of functions that all grow in a way considered equivalent for purposes of analysis of algorithms. For example, all functions that grow linearly belong to the same order of growth.

Big-Oh notation: Notation for representing an order of growth; for example, $O(n)$ represents the set of functions that grow linearly.

linear: An algorithm whose run time is proportional to problem size, at least for large problem sizes.

quadratic: An algorithm whose run time is proportional to n^2, where n is a measure of problem size.

search: The problem of locating an element of a collection (like a list or dictionary) or determining that it is not present.

hashtable: A data structure that represents a collection of key-value pairs and performs search in constant time.

Índice alfabético