Contenido

Capítulo 1

El camino del programa

El objetivo de este libro es enseñarte a pensar como un científico de la computación. 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 científicos informáticos utilizan los lenguajes formales para designar ideas (específicamente cálculos). Al igual que los ingenieros, diseñan cosas, ensamblan componentes en los sistemas y evalúan los intercambios entre las alternativas. Al igual que los científicos, observan el comportamiento de sistemas complejos, forman hipótesis y prueban predicciones.

La habilidad más importante para un científico informático es la resolución de problemas. La resolución de problemas significa la capacidad de formular problemas, pensar creativamente sobre soluciones y expresar una solución 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, aprenderá a programar, una habilidad útil en sí misma. En otro nivel, utilizará la programación como un medio para un fin. A medida que avancemos, ese final se volverá más claro.

1.1 ¿Qué es un programa?

Un programa es una secuencia de instrucciones que especifica cómo realizar un cálculo. El cálculo puede ser algo matemático, como resolver un sistema de ecuaciones o encontrar las raíces de un polinomio, pero también puede ser un cálculo simbólico, como buscar y reemplazar texto en un documento o algo gráfico, como procesar una imagen o jugar un videojuego.

Los detalles se ven diferentes en diferentes idiomas, pero algunas instrucciones básicas aparecen en casi todos los idiomas:

input: Obtiene datos del teclado, un archivo, la red o algún otro dispositivo.

output: Muestra los datos en la pantalla, los guarda en un archivo, los envía a través de la red, etc.

math: Realiza operaciones matemáticas básicas como la suma y la multiplicación.

ejecución condicional: Verifica ciertas condiciones y ejecuta el código apropiado.

repetición: Realiza alguna acción repetidamente, generalmente con alguna variación.

Lo creas o no, eso es todo lo que hay que hacer. Cada programa que hayas usado, sin importar cuán complicado sea, está compuesto por instrucciones que se parecen mucho a éstas. Entonces, puede pensar en la programación como el proceso de dividir una tarea grande y compleja en subtareas cada vez más pequeñas hasta que las subtareas sean lo suficientemente simples como para realizarse con una de estas instrucciones básicas.

1.2 Ejecutando Python

Uno de los desafíos de comenzar con Python es que puede que tenga que instalar Python y el software relacionado en su computadora. Si está familiarizado con su sistema operativo, y especialmente si se siente cómodo con la interfaz de línea de comandos, no tendrá problemas para instalar Python. Pero para los principiantes, puede ser doloroso aprender sobre la administración del sistema y la programación al mismo tiempo.

Para evitar ese problema, le recomiendo que comience a ejecutar Python en un navegador. Más tarde, cuando esté cómodo con Python, le haré sugerencias para instalar Python en su computadora.

Hay varias páginas web que puede usar para ejecutar Python. Si ya tiene una favorita, adelante y úsela. De lo contrario, recomiendo PythonAnywhere. Proporciono instrucciones detalladas para comenzar en *http://tinyurl.com/thinkpython2e* .

Hay dos versiones de Python, llamadas Python 2 y Python 3. Son muy similares, por lo que si aprendes una, es fácil cambiar a la otra. De hecho, solo hay algunas 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 el código de Python. Dependiendo de su entorno, puede iniciar el intérprete haciendo clic en un icono o escribiendo `python` en una línea de comando. Cuando comienza, debería ver resultados como este:

```
Python 3.4.0 (predeterminado, 19 de junio de 2015, 14:20:21)
[GCC 4.8.2] en Linux
Escriba "ayuda", "derechos de autor", "créditos" o "licencia" para obtener más información.
>>>
```

Las primeras tres líneas contienen información sobre el intérprete y el sistema operativo en el que se está ejecutando, por lo que podría ser diferente para usted. Pero deberías verificar que el número de versión 3.4.0, que está en este ejemplo, comienza con 3, lo que indica que estás ejecutando Python 3. Si comienza con 2, estás ejecutando (lo adivinaste) Python 2.

La última línea es un **aviso** que indica que el intérprete está listo para que ingrese el código. Si escribe una línea de código y presiona Enter, el intérprete muestra el resultado:

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

Ahora estás listo para comenzar. A partir de ahora, supongo que sabes cómo iniciar el intérprete de Python y ejecutar el código.

1.3 El primer programa

Tradicionalmente, el primer programa que escribe en un nuevo idioma 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 declaración impresa , aunque en realidad no imprime nada en papel. Muestra un resultado en la pantalla. En este caso, el resultado son las palabras

```
¡Hola Mundo!
```

Las comillas en el programa marcan el comienzo y el final del texto que se mostrará; ellos 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 instrucción de impresión es ligeramente diferente; no es una función, por lo 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 "Hello, World", el siguiente paso es la aritmética. Python proporciona **operadores**, que son símbolos especiales que representan cálculos como la suma y la multiplicación.

Los operadores +, - y * realizan la suma, resta y multiplicación, como en los siguientes ejemplos:

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

El operador /realiza la división:

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

Quizás se pregunte por qué el resultado es 42.0 en lugar de 42. Lo explicaré en la siguiente sección.

Finalmente, el operador ** realiza la exponenciación; es decir, aumenta un número a una potencia:

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

En algunos otros lenguajes, ^ se usa para la exponenciación, pero en Python es un operador bit a bit llamado XOR. Si no está familiarizado con los operadores bit a bit, el resultado lo sorprenderá:

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

No cubriré operadores bit a bit en este libro, pero puede leer sobre ellos en *http://wiki.python.org/moin/BitwiseOperators* .

1.5 Valores y tipos

Un **valor** es una de las cosas básicas con las que trabaja un programa, como una letra o un número. Algunos valores que hemos visto hasta ahora son 2, 42.0 y 'Hello, World!'

Estos valores pertenecen a diferentes **tipos**: 2 es un **número entero**, 42.0 es un número de **coma flotante**, y 'Hello, World!' es una **cadena**, llamada así porque las letras que contiene están unidas.

Si no está seguro de qué tipo tiene un valor, el intérprete puede decirle:

```
>>> escriba (2)
<class 'int'>
>>> escriba (42.0)
<clase 'float'>
>>> escriba ('Hello, World!')
<class 'str'>
```

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

No es sorprendente que los enteros pertenezcan al tipo `int`, al que pertenecen las cadenas `str` y los números de punto flotante pertenecen `float`.

¿Qué pasa con los valores como '2' y '42.0'? Se parecen a los números, pero están entre comillas como cadenas:

```
>>> tipo ('2')
<clase 'str'>
>>> tipo ('42 .0 ')
<clase' str '>
```

Son cadenas.

Cuando escribe un número entero grande, es posible que tenga la tentación de usar comas entre grupos de dígitos, como en `1,000,000`. Este no es un *número 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 Idiomas formales y naturales

Los idiomas naturales son los idiomas que las personas hablan, como inglés, español y francés. No fueron diseñados por personas (aunque las personas intentan imponerles algún orden); evolucionaron naturalmente.

Los lenguajes formales son idiomas diseñados por personas para aplicaciones específicas. Por ejemplo, la notación que usan los matemáticos es un lenguaje formal que es particularmente bueno para 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 cómputos.

Los lenguajes formales tienden a tener reglas de sintaxis estrictas que rigen la estructura de los enunciados. Por ejemplo, en matemáticas, la declaración 3+3 = 6 tiene sintaxis correcta, pero 3+= 3$6 no. En química, H 2 O es una fórmula sintácticamente correcta, pero 2Zz no lo es.

Las reglas de sintaxis vienen en dos sabores, pertenecientes a tokens y estructura. Los tokens son los elementos básicos del lenguaje, como palabras, números y elementos químicos. Uno de los problemas con 3+3=3$6 es que $ no es una ficha legal en matemáticas (al menos hasta donde sé). Del mismo modo, 2Zz no es legal porque no hay ningún elemento con la abreviatura Zz.

El segundo tipo de regla de sintaxis se refiere a cómo se combinan los tokens. La ecuación 3+=3 es ilegal porque a pesar de que + y = son símbolos legales, no puedes tener uno justo después del otro. De manera similar, en una fórmula química, el subíndice viene después del nombre del elemento, no antes.

Esta es @ frase de Engli $ h bien estructurada con t * kens no válidos en ella. Esta oración tiene todos los tokens válidos, pero la estructura no es válida con.

Cuando lee una oración en inglés o una declaración en un lenguaje formal, tiene que descubrir la estructura (aunque en un lenguaje natural lo hace inconscientemente). Este proceso se llama análisis sintáctico.

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

ambigüedad: Los lenguajes naturales están llenos de ambigüedad, que las personas manejan mediante el uso de pistas contextuales y otra información. Los lenguajes formales están diseñados para ser casi o completamente inequívocos, lo que significa que cualquier enunciado tiene exactamente un significado, independientemente del contexto.

redundancia: Con el fin de compensar la ambigüedad y reducir los malentendidos, los lenguajes naturales emplean mucha redundancia. Como resultado, a menudo son detallados. Los lenguajes formales son menos redundantes y más concisos.

literalidad: Los lenguajes naturales están llenos de modismos y metáforas. Si digo: "Se cayó el centavo", probablemente no haya ni un centavo ni nada que caiga (este modismo significa que alguien entendió algo después de un período de confusión). Los lenguajes formales significan exactamente lo que dicen.

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

Poesía: Las palabras se usan tanto para sus sonidos como para su significado, y todo el poema en conjunto crea un efecto o respuesta emocional. La ambigüedad no es solo 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 a menudo ambigua.

Programas: El significado de un programa de computadora no es ambiguo y literal, y puede entenderse completamente mediante el análisis de los tokens y la estructura.

Los lenguajes formales son más densos que los lenguajes naturales, por lo que lleva más tiempo leerlos. Además, la estructura es importante, por lo que no siempre es mejor leer de arriba a abajo, de izquierda a derecha. En cambio, aprenda a analizar el programa en su cabeza, identificando los tokens e interpretando la

estructura. Finalmente, los detalles importan. Pequeños errores en la ortografía y la puntuación, que puede salirse con la suya en los idiomas naturales, pueden marcar una gran diferencia en un lenguaje formal.

1.7 Depuración

Los programadores cometen fallos. Por razones caprichosas, las fallas de programación se llaman **errores** y el proceso de seguimiento se denomina **depuración**.

La programación, y especialmente la depuración, a veces pone de manifiesto emociones fuertes. Si está luchando con un error difícil, puede sentirse enojado, abatido o avergonzado.

Existe evidencia de que las personas responden naturalmente a las computadoras como si fueran personas. Cuando funcionan bien, los consideramos como compañeros de equipo, y cuando son obstinados o groseros, les respondemos de la misma manera que respondemos a las personas groseras y obstinadas (*Reeves y Nass, The Media Equation: How People Treat Computers, Television, y New Media Like Real People and Places*).

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

Su trabajo es ser un buen administrador: encuentre maneras de aprovechar las fortalezas y mitigar las debilidades. Y encuentre maneras de usar sus emociones para enfrentar el problema, sin permitir que sus reacciones interfieran con su capacidad para trabajar de manera efectiva.

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 expresarlo.

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 ser fácil de ejecutar por una computadora; también llamado "lenguaje de máquina" o "lenguaje ensamblador".

portabilidad:

> Una propiedad de un programa que puede ejecutarse en más de un tipo de computadora.

Interprete:

> Un programa que lee otro programa y lo ejecuta.

rápido:

Caracteres mostrados por el intérprete para indicar que está listo para recibir información del usuario.

programa:

Un conjunto de instrucciones que especifica un cálculo.

declaración de impresión:

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

operador:

Un símbolo especial que representa un cálculo simple como adición, multiplicación o concatenación de cadenas.

valor:

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

tipo:

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

entero:

Un tipo que representa números enteros.

punto flotante:

Un tipo que representa números con partes fraccionales.

cadena:

Un tipo que representa secuencias de caracteres.

lenguaje natural:

Cualquiera de los idiomas que habla la gente que evolucionó naturalmente.

lenguaje formal:

Cualquiera de los idiomas que la gente ha diseñado para fines específicos, como representar ideas matemáticas o programas de computadora; todos los lenguajes de programación son lenguajes formales.

simbólico:

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

sintaxis:

Las reglas que rigen la estructura de un programa.

analizar gramaticalmente:

Para examinar un programa y analizar la estructura sintáctica.

error:

Una falla en un programa.

depuración:

El proceso de encontrar y corregir errores.

1.9 Ejercicios

Ejercicio 1-1.

Es una buena idea leer este libro frente a una computadora para que pueda probar los ejemplos sobre la marcha.

Cada vez que experimente con una nueva característica, debe intentar cometer errores. Por ejemplo, en el programa "¡Hola, mundo!", ¿Qué sucede si omite una de las comillas? ¿Qué pasa si dejas fuera a los dos? ¿Qué pasa si deletreas `print` mal?

Este tipo de experimento te ayuda a recordar lo que lees; también ayuda cuando estás programando, porque sabes lo que significan los mensajes de error. Es mejor cometer errores ahora y a propósito que tarde y accidentalmente.

1. En una declaración `print`, ¿qué sucede si omite uno de los paréntesis o ambos?

2. Si está tratando de imprimir una cadena, ¿qué sucede si omite una de las comillas, o ambas?

3. Puedes usar un signo menos para hacer un número negativo como -2. ¿Qué sucede si pones un signo más antes de un número? ¿Qué tal 2++2?

4. En notación matemática, los ceros a la izquierda están bien, como en 02. ¿Qué pasa si pruebas esto en Python?

5. ¿Qué sucede si tienes dos valores sin operador entre ellos?

Ejercicio 1-2.

Inicie el intérprete de Python y úselo como una calculadora.

1. ¿Cuántos segundos hay en 42 minutos y 42 segundos?

2. ¿Cuántas millas hay en 10 kilómetros? Sugerencia: hay 1,61 kilómetros en una milla.

3. Si corres una carrera de 10 kilómetros en 42 minutos y 42 segundos, ¿cuál es tu ritmo promedio (tiempo por milla en minutos y segundos)? ¿Cuál es tu velocidad promedio en millas por hora?

Capítulo 2

Variables, expresiones y declaraciones

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

2.1 Declaraciones de asignación

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

```
>>> message = 'Y ahora para algo completamente diferente'
>>> n = 17
>>> pi = 3.141592653589793
```

Este ejemplo hace tres asignaciones. El primero asigna una cadena a una nueva variable llamada `message`; el segundo da el entero `17` a `n`; el tercero asigna el valor (aproximado) de π a `pi`.

Una forma común de representar variables en papel es escribir el nombre con una flecha que apunta a su valor. Este tipo de figura se denomina **diagrama de estado** porque muestra en qué estado se encuentra cada una de las variables (piénselo como el estado de ánimo de la variable). La Figura 2-1 muestra el resultado del ejemplo anterior.

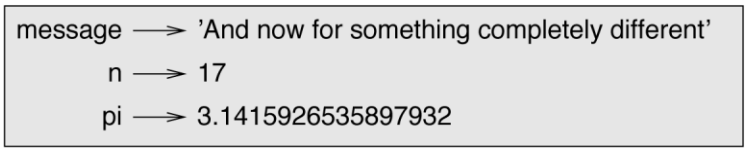

Figura 2-1. Diagrama de estado.

2.2 Nombres de variables

Los programadores generalmente eligen nombres para sus variables que son significativos, y documentan para qué se usa la variable.

Los nombres de variables pueden ser tan largos como desee. Pueden contener letras y 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 carácter de subrayado, _ puede aparecer en un nombre. A menudo se usa en nombres con varias palabras, como `your_name` o `airspeed_of_unladen_swallow`.

Si asigna a una variable un nombre ilegal, obtendrá un error de sintaxis:

```
>>> 76trombones = 'gran desfile'
```

```
Error de sintaxis: sintaxis invalida
>>> more@ = 1000000
Error de sintaxis: sintaxis invalida
>>> class = 'Advanced Zymurgy teórico'
Error de sintaxis: sintaxis invalida
```

`76trombones` es ilegal porque comienza con un número. `more@` es ilegal, ya que contiene un carácter ilegal, `@`. Pero, ¿qué pasa con `class`?

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

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 en un color diferente; si intenta usar uno como nombre de variable, lo sabrá.

2.3 Expresiones y declaraciones

Una **expresión** es una combinación de valores, variables y operadores. Un valor en sí mismo se considera una expresión, y también lo es una variable, por lo que las siguientes son todas expresiones legales:

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

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

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

```
>>> n = 17
>>> imprimir (n)
```

La primera línea es una declaración de asignación que da un valor a `n`. La segunda línea es una declaración de impresión que muestra el valor de `n`.

Cuando escribe una declaración, el intérprete la **ejecuta**, lo que significa que hace lo que dice la declaración. En general, las declaraciones no tienen valores.

2.4 Modo Script

Hasta ahora, hemos ejecutado Python en **modo interactivo**, lo que significa que usted interactúa directamente con el intérprete. El modo interactivo es una buena forma de comenzar, pero si trabajas con más de unas pocas líneas de código, puede ser torpe.

La alternativa es guardar el código en un archivo llamado **script** y luego ejecutar el intérprete en **modo script** para ejecutar el script. Por convención, las secuencias de comandos de Python tienen nombres que terminan con `.py`.

Si sabe cómo crear y ejecutar un script en su computadora, ya puede hacerlo. De lo contrario, recomiendo usar PythonAnywhere nuevamente. He publicado instrucciones para ejecutar en modo script en *http://tinyurl.com/thinkpython2e*.

Como Python proporciona ambos modos, puede probar bits de código en modo interactivo antes de ponerlos en un script. Pero existen diferencias entre el modo interactivo y el modo de secuencia de comandos que pueden ser confusos.

Por ejemplo, si está usando Python como calculadora, puede escribir:

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

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

Pero si escribe el mismo código en un script y lo ejecuta, no obtendrá ningún resultado. En modo script, una expresión, por sí sola, no tiene efecto visible. Python realmente evalúa la expresión, pero no muestra el valor a menos que usted le diga:

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

Este comportamiento puede ser confuso al principio.

Un script generalmente contiene una secuencia de declaraciones. Si hay más de una declaración, los resultados aparecen de uno en uno a medida que se ejecutan las declaraciones.

Por ejemplo, la secuencia de comandos

```
imprimir (1)
x = 2
imprimir (x)
```

produce la salida

```
1
2
```

La instrucción de asignación no produce salida.

Para verificar su comprensión, escriba las siguientes declaraciones en el intérprete de Python y vea lo que hacen:

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

Ahora ponga las mismas declaraciones en un script y ejecútelo. ¿Cuál es el resultado? Modifique la secuencia de comandos transformando cada expresión en una declaración de impresión y luego ejecútela nuevamente2..

2.5 Orden de operaciones

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

- El parámetro **P** tienen la precedencia más alta y se pueden usar para forzar que una expresión se evalúe en el orden que desee. Dado que las expresiones entre paréntesis se evalúan primero, `2 * (3-1)` es 4 y `(1+1)**(5-2)` es 8. También puede usar paréntesis para hacer que una expresión sea más fácil de leer, como en `(minute * 100) / 60`, incluso si no cambia el resultado.

- **E**xponenciación tiene la siguiente precedencia más alta, entonces `1 + 2**3` es 9, no 27, y `2 * 3**2` es 18, no 36.

- **M**ultiplicación y **D**ivisión tienen mayor precedencia que **A**dición y **S**ustracción. Entonces `2*3-1` es 5, no 4, y `6+4/2` es 8, no 5.

- Los operadores con la misma precedencia se evalúan de izquierda a derecha (excepto la exponenciación). Entonces en la expresión `degrees / 2 * pi`, la división ocurre primero y el resultado se multiplica por `pi`. Para dividir por, puede usar paréntesis o escribir `degrees / 2 / pi`.

No trabajo muy duro para recordar la precedencia de los operadores. Si no puedo decir al mirar la expresión, uso paréntesis para hacerlo obvio.

2.6 Operaciones de cadena

En general, no puede realizar operaciones matemáticas en cadenas, incluso si las cadenas se parecen a números, por lo que las siguientes son ilegales:

```
'2' - '1' 'huevos' / 'fácil' 'tercero' * 'un amuleto'
```

Pero hay dos excepciones, + y *.

El operador + realiza la concatenación de cadenas, lo que significa que une las cadenas uniéndolas de extremo a extremo. Por ejemplo:

```
>>> first = 'throat'
>>> second = 'curruca'
>>> primer + segundo
trabador de garganta
```

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

El uso de + y * tiene analogía con la suma y la multiplicación. Así como `4*3` es equivalente `4+4+4`, esperamos `'Spam'*3` ser lo mismo que `'Spam'+'Spam'+'Spam'`, y lo es. Por otro lado, hay una forma significativa en que la concatenación y la repetición de cadenas son diferentes de la suma y multiplicación de enteros. ¿Puedes pensar en una propiedad que además tiene esa concatenación de cadenas no?

2.7 Comentarios

A medida que los programas crecen y se vuelven más complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es difícil mirar un fragmento de código y descubrir qué está haciendo o por qué.

Por esta razón, es una buena idea agregar notas a sus programas para explicar en lenguaje natural lo que está haciendo 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 solo en una línea. También puedes poner comentarios al final de una línea:

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

Todo desde el # hasta el final la línea se ignora, no tiene ningún efecto en la ejecución del programa.

Los comentarios son más útiles cuando documentan características no obvias del código. Es razonable suponer que el lector puede descubrir *qué hace* el código; es más útil para 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 # de velocidad en metros / segundo.
```

Los buenos nombres de variables pueden reducir la necesidad de comentarios, pero los nombres largos pueden hacer que las expresiones complejas sean difíciles de leer, por lo que hay una solución de compromiso.

2.8 Depuración

Tres tipos de errores pueden ocurrir en un programa: errores de sintaxis, errores de tiempo de ejecución y errores semánticos. Es útil distinguirlos para rastrearlos más rápidamente.

Error de sintaxis:

"Sintaxis" se refiere a la estructura de un programa y las reglas sobre esa estructura. Por ejemplo, los paréntesis deben venir en parejas iguales, por lo que (1 + 2) es legal, pero 8) es un error de sintaxis.

Si hay un error de sintaxis en cualquier parte de su programa, Python muestra un mensaje de error y se cierra, y no podrá ejecutar el programa. Durante las primeras semanas de su carrera de programación, puede pasar mucho tiempo rastreando los errores de sintaxis. A medida que gane experiencia, hará menos errores y los encontrará 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 de que el programa haya comenzado a ejecutarse. Estos errores también se llaman excepciones porque generalmente indican que algo excepcional (y malo) ha sucedido.

Los errores de tiempo de ejecución son raros en los programas simples que verá en los primeros capítulos, por lo que podría pasar un tiempo antes de que encuentre uno.

Error semántico:

El tercer tipo de error es "semántico", lo que significa relacionado con el significado. Si hay un error semántico en su programa, se ejecutará sin generar mensajes de error, pero no hará lo correcto. Hará algo más. Específicamente, hará lo que le dijo que hiciera.

La identificación de errores semánticos puede ser difícil porque requiere que trabajes hacia atrás mirando el resultado del programa y tratando de descubrir qué está haciendo.

2.9 Glosario

variable:

Un nombre que se refiere a un valor.

asignación:

Una declaración que asigna un valor a una variable.

diagrama de estado:

Una representación gráfica de un conjunto de variables y los valores a los que se refieren.

palabra clave:

Una palabra reservada que se usa para analizar un programa; no se puede utilizar como palabras clave `if`, `def` y `while` como nombres de variables.

operando:

Uno de los valores en los que opera un operador.

expresión:

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

evaluar:

Para simplificar una expresión realizando las operaciones para producir un solo valor.

declaración:

Una sección de código que representa un comando o acción. Hasta ahora, las declaraciones que hemos visto son asignaciones e instrucciones de impresión.

ejecutar:

Ejecuta una declaración y hacer lo que dice.

modo interactivo:

Una forma de usar el intérprete de Python escribiendo el código en el prompt.

modo de secuencia de comandos:

Una forma de utilizar el intérprete de Python para leer el código de un script y ejecutarlo.

script:

Un programa almacenado en un archivo.

Orden de operaciones:

Reglas que rigen el orden en que se evalúan las expresiones que involucran múltiples operadores y operandos.

concatenar:

Para unir dos operandos de extremo a extremo.

comentario:

Información en un programa que está destinado a otros programadores (o cualquier persona que lea el código fuente) y no tiene ningún efecto en la ejecución del programa.

error de sintaxis:

Un error en un programa que hace que sea imposible de analizar (y por lo tanto imposible de interpretar).

excepción:

Un error que se detecta mientras el programa se está ejecutando.

semántica:

El significado de un programa.

error semántico:

Un error en un programa que lo hace hacer algo diferente de lo que el programador pretendía.

2.10 Ejercicios

Ejercicio 2-1.

Repitiendo mi consejo del capítulo anterior, cada vez que aprenda una nueva característica, debe probarla en modo interactivo y cometer errores a propósito para ver qué falla.

- Hemos visto que n = 42 es legal. ¿Qué tal 42 = n?
- ¿Qué tal x = y = 1?
- En algunos lenguajes de cada declaración termina con un punto y coma, ;. ¿Qué sucede si pones un punto y coma al final de una declaración de Python?
- ¿Qué pasa si coloca un punto al final de una declaración?
- En notación matemática se puede multiplicar x e y como esto: . ¿Qué pasa si pruebas 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 $4/3\pi r^3$. ¿Cuál es el volumen de una esfera con radio 5?
2. Supongamos que el precio de portada de un libro es de $ 24.95, pero las librerías obtienen un descuento del 40%. El envío cuesta $ 3 por la primera copia y 75 centavos por cada copia adicional. ¿Cuál es el costo total al por mayor de 60 copias?
3. Si salgo de mi casa a las 6:52 a.m. y corro 1 milla a un ritmo fácil (8:15 por milla), luego 3 millas a ritmo (7:12 por milla) y 1 milla a un ritmo fácil otra vez, ¿a qué hora Llego a casa para el desayuno?

Capítulo 3

Funciones

En el contexto de la programación, una **función** es una secuencia de declaraciones nombrada que realiza un cálculo. Cuando define una función, especifica el nombre y la secuencia de las declaraciones. Más tarde, puede "llamar" a la función por su nombre.

3.1 Llamadas de función

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

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

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

Es común decir que una función "toma" un argumento y "devuelve" un resultado. El resultado también se llama **valor de retorno**.

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 se queja de lo contrario:

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

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

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

`float` convierte números enteros y cadenas a números de coma flotante:

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

Finalmente, `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 mayoría de las funciones matemáticas familiares. Un módulo es un archivo que contiene una colección de funciones relacionadas.

Antes de que podamos usar las funciones en un módulo, tenemos que importarlo con una declaración de importación:

```
>>> import math
```

Esta declaración crea un objeto de **módulo llamado** matemática. Si visualiza el objeto del módulo, obtiene información al respecto:

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

El objeto del módulo contiene las funciones y variables definidas en el módulo. Para acceder a una de las funciones, debe especificar el nombre del módulo y el nombre de la función, separados por un punto (también conocido como periodo). Este formato se llama **notación de puntos**.

```
>>> ratio = signal_power / noise_power
>>> decibelios = 10 * math.log10 (ratio)


>>> radians = 0.7
>>> height = math.sin (radians)
```

El primer ejemplo se usa `math.log10` para calcular una relación señal / ruido en decibelios (asumiendo que `signal_power` y `noise_power` están definidos). El módulo matemático también proporciona `log`, que calcula los logaritmos base e.

El segundo ejemplo encuentra el seno de `radians`. El nombre de la variable es un indicio de que `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) tomar argumentos en radianes. Para convertir de grados a radianes, divida por 180 y multiplique 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 matemático. Su valor es una aproximación de punto flotante de π, con una precisión de aproximadamente 15 dígitos.

Si conoce la trigonometría, puede verificar el resultado anterior comparándolo con la raíz cuadrada de 2 dividida por 2:

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

3.3 Composición

Hasta ahora, hemos analizado los elementos de un programa (variables, expresiones y declaraciones) de forma aislada, sin hablar de cómo combinarlos.

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

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

E incluso llamadas a funciones:

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

Casi en cualquier lugar donde pueda poner un valor, puede poner una expresión arbitraria, con una excepción: el lado izquierdo de una declaración 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 adelante).

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

3.4 Agregar nuevas funciones

Hasta ahora, solo hemos usado las funciones que vienen con Python, pero también es posible agregar nuevas funciones. Una definición de función específica el nombre de una nueva función y la secuencia de instrucciones que se ejecutan cuando se llama a la función.

Aquí hay un ejemplo:

```
def print_lyrics ():
    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 `print_lyrics`. Las reglas para los nombres de las funciones son las mismas que para los nombres de las variables: las letras, los números y los guiones bajos son legales, pero el primer carácter no puede ser un número. No puede usar una palabra clave como el nombre de una función, y debe 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**; el resto se llama el **cuerpo**. El encabezado debe terminar con dos puntos y el cuerpo debe sangrarse. Por convención, la sangría es siempre cuatro espacios. El cuerpo puede contener cualquier cantidad de declaraciones.

Las cadenas en las instrucciones de impresión están entre comillas dobles. Las comillas simples y las comillas dobles hacen lo mismo; la mayoría de las personas usa comillas simples, excepto en casos como este donde aparece una comilla simple (que también es un apóstrofo) en la cadena.

Todas las comillas (simple y doble) deben ser "comillas rectas", generalmente ubicadas al lado de Enter en el teclado. Las “comillas rizadas”, como las de esta oración, no son legales en Python.

Si escribe una definición de función en modo interactivo, el intérprete imprime puntos (...) para hacerle saber que la definición no está completa:

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

Para finalizar la función, debe ingresar una línea vacía.

La definición de una función crea un **objeto de función**, que es de tipo `function`:

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

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

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

Una vez que haya definido una función, puede usarla dentro de otra función. Por ejemplo, para repetir el estribillo anterior, podríamos escribir una función llamada repeat_lyrics:

```
def repeat_lyrics ():
    print_lyrics ()
    print_lyrics ()
```

Y luego llama `repeat_lyrics`:

```
>>> repeat_lyrics()
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 esa no es la forma en que va la canción.

3.5 Definiciones y Usos

Al juntar los fragmentos de código de la sección anterior, todo el programa se ve así:

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


def repeat_lyrics():
    print_lyrics()
    print_lyrics()


repeat_lyrics()
```

Este programa contiene dos definiciones de funciones: `print_lyrics` y `repeat_lyrics`. Las definiciones de funciones se ejecutan como otras declaraciones, pero el efecto es crear objetos de función. Las instrucciones dentro de la función no se ejecutan hasta que se llama a la función y la definición de la función no genera salida.

Como era de esperar, debe crear una función antes de poder ejecutarla. En otras palabras, la definición de la función debe ejecutarse antes de que se llame a la función.

Como ejercicio, mueva la última línea de este programa a la parte superior, de modo que la llamada a la función aparezca antes de las definiciones. Ejecute el programa y vea qué mensaje de error recibe.

Ahora mueva la llamada de función a la parte inferior y mueva la definición de `print_lyrics` después de la definición de `repeat_lyrics`. ¿Qué pasa cuando ejecutas este programa?

3.6 Flujo de ejecución

Para asegurarse de que se define una función antes de su primer uso, debe conocer las sentencias de orden que se ejecutan, que se denomina **flujo de ejecución**.

La ejecución siempre comienza en la primera declaración del programa. Las declaraciones se ejecutan una a la vez, en orden de arriba a abajo.

Las definiciones de función no alteran el flujo de ejecución del programa, pero recuerde que las instrucciones dentro de la función no se ejecutan hasta que se llama a la función.

Una llamada a función es como un desvío en el flujo de ejecución. En lugar de ir al siguiente enunciado, el flujo salta al cuerpo de la función, ejecuta los enunciados allí y luego vuelve a retomar donde lo dejó.

Eso suena bastante simple, hasta que recuerde que una función puede llamar a otra. Mientras está en medio de una función, el programa podría tener que ejecutar las declaraciones en otra función. Luego, mientras ejecuta esa nueva función, ¡el programa podría tener que ejecutar otra función más!

Afortunadamente, Python es bueno en el seguimiento de dónde está, por lo que cada vez que se completa una función, el programa continúa donde lo dejó en la función que lo llamó. Cuando llega al final del programa, termina.

En resumen, cuando lee un programa, no siempre quiere leer de arriba 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 lo llamas `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 se asignan a variables llamadas **parámetros**. Aquí hay una definición para una función que toma un argumento:

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

Esta función asigna el argumento a un parámetro nombrado `bruce`. Cuando se llama a la función, imprime el valor del parámetro (cualquiera que sea) dos veces.

Esta función funciona con cualquier valor que se pueda imprimir:

```
>>> print_twice ('Spam')
Spam
Spam
>>> print_twice (42)
42
42
>>> print_twice (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, por lo que podemos usar cualquier tipo de expresión como argumento para print_twice:

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

El argumento se evalúa antes de llamar a la función, por lo que en los ejemplos las expresiones `'Spam' * 4` y `math.cos(math.pi)` solo se evalúan una vez.

También puedes usar una variable como argumento:

```
>>> michael = 'Eric, la mitad de una abeja.'
>>> print_twice (michael)
Eric, la mitad de una abeja.
Eric, la mitad de una abeja.
```

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 llamó el valor; adentro de `print_twice`, llamamos a todos `bruce`.

3.8 Las variables y los parámetros son locales

Cuando crea una variable dentro de una función, es **local**, lo que significa que solo existe dentro de la función. Por ejemplo:

```
def cat_twice (part1, part2):
    cat = part1 + part2
    print_twice (cat)
```

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

```
>>> line1 = 'Bing tiddle'
>>> line2 = ' tiddle bang.'
>>> cat_twice (line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

Cuando `cat_twice` termina, la variable `cat` se destruye. Si tratamos de imprimirlo, obtenemos una excepción:

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

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

3.9 Diagramas de stack

Para realizar un seguimiento de las variables que se pueden usar donde, a veces es útil dibujar un **diagrama de stack**. Al igual que los diagramas de estado, los diagramas de stack muestran el valor de cada variable, pero también muestran la función a la que pertenece cada variable.

Cada función está representada por un marco. Un marco es un cuadro con el nombre de una función al lado y los parámetros y variables de la función dentro de él. El diagrama de stack para el ejemplo anterior se muestra en la Figura 3-1.

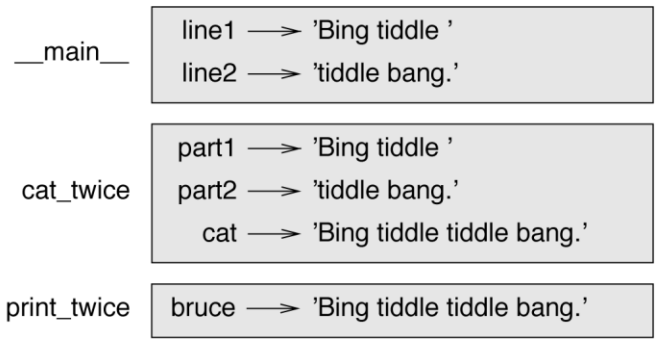

Figura 3-1. Diagrama de stack.

Los marcos están dispuestos en una pila que indica qué función llamó a qué, y así sucesivamente. En este ejemplo, `print_twice` fue llamado por `cat_twice`, y `cat_twice` fue llamado por `__main__`, que es un nombre especial para el marco superior. Cuando creas una variable fuera de cualquier función, pertenece a `__main__`.

Cada parámetro se refiere al mismo valor que su argumento correspondiente. Por lo tanto, `part1` tiene el mismo valor que `line1`, `part2` tiene el mismo valor que `line2`, y `bruce` tiene el mismo valor que `cat`.

Si se produce 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 que, todo el camino de vuelta a `__main__`.

Por ejemplo, si intenta acceder `cat` desde dentro `print_twice`, obtiene un NameError:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

Esta lista de funciones se llama **traceback**. Le dice en qué archivo de programa se produjo el error, qué línea y qué funciones se estaban ejecutando en ese momento. También muestra la línea de código que causó el error.

El orden de las funciones en la traceback es el mismo que el orden de las tramas en el diagrama de la pila. La función que se está ejecutando actualmente se encuentra en la parte inferior.

3.10 Funciones fructíferas y funciones nulas

Algunas de las funciones que hemos utilizado, como las funciones matemáticas, resultados de retorno; a falta de un nombre mejor, los llamo **funciones fructíferas**. Otras funciones, como `print_twice`, realizan una acción pero no devuelven un valor. Se llaman **funciones vacías**.

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

```
x = math.cos (radianes)
dorados = (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 fructífera por sí misma, el valor de retorno se pierde para siempre!

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

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

Las funciones vacías pueden mostrar algo en la pantalla o tener algún otro efecto, pero no tienen un valor de retorno. Si asigna el resultado a una variable, obtiene un valor especial llamado `None`:

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

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

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

Las funciones que hemos escrito hasta ahora son todas nulas. Comenzaremos a escribir funciones fructíferas en algunos capítulos.

3.11 ¿Por qué funciones?

Puede que no esté claro por qué vale la pena dividir un programa en funciones. Hay varias razones:

- Crear una nueva función le da la oportunidad de nombrar un grupo de declaraciones, lo que hace que su programa sea más fácil de leer y depurar.
- Las funciones pueden hacer que un programa sea más pequeño al eliminar el código repetitivo. Más tarde, si haces un cambio, solo tienes que hacerlo en un solo lugar.
- La división de un programa largo en funciones le permite depurar las partes de una en una y luego ensamblarlas en un todo funcional.
- Las funciones bien diseñadas son a menudo útiles para muchos programas. Una vez que escribe y depura uno, puede reutilizarlo.

3.12 Depuración

Una de las habilidades más importantes que adquirirá es la depuración. Aunque puede ser frustrante, la depuración es una de las partes de la programación más intelectualmente rica, desafiante e interesante.

En cierto modo, la depuración es como el trabajo de detective. Te enfrentas a pistas y tienes que inferir los procesos y eventos que llevaron a los resultados que ves.

La depuración también es como una ciencia experimental. Una vez que tenga una idea de lo que está pasando mal, modifique su programa y vuelva a intentarlo. Si su hipótesis es correcta, puede predecir el resultado de la modificación y dar un paso más hacia un programa en funcionamiento. Si su hipótesis era incorrecta, tiene que encontrar una nueva. Como señaló Sherlock Holmes, "cuando eliminas lo imposible, lo que queda, aunque sea improbable, debe ser la verdad" (A. Conan Doyle, *The Sign of Four*).

Para algunas personas, la programación y la depuración son la misma cosa. Es decir, la programación es el proceso de depuración gradual de un programa hasta que haga lo que usted desea. La idea es que debe comenzar con un programa que funcione y realizar pequeñas modificaciones, depurándolo a medida que avanza.

Por ejemplo, Linux es un sistema operativo que contiene millones de líneas de código, pero comenzó como un simple programa que Linus Torvalds utilizó 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 posteriormente a Linux." (*La Guía de usuarios de Linux* versión beta 1).

3.13 Glosario

función:

> Una secuencia de declaraciones con nombre que realiza alguna operación útil. Las funciones pueden o no tomar argumentos y pueden o no producir un resultado.

definición de la función:

> Una declaración que crea una nueva función, especificando su nombre, parámetros y las declaraciones que contiene.

objeto de función:

> Un valor creado por una definición de función. El nombre de la función es una variable que hace referencia a un objeto de función.

encabezado:

> La primera línea de una definición de función.

cuerpo:

> La secuencia de instrucciones dentro de una definición de función.

parámetro:

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

Llamada de función:

> Una declaración que ejecuta una función. Consiste en el nombre de la función seguido de una lista de argumentos entre paréntesis.

argumento:

> Un valor proporcionado a una función cuando se llama a la función. Este valor se asigna 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 una expresión, el valor de retorno es el valor de la expresión.

función fructífera:

> Una función que devuelve un valor.

función nula:

> Una función que siempre regresa `None`.

None:

> Un valor especial devuelto por las funciones de vacío.

módulo:

> Un archivo que contiene una colección de funciones relacionadas y otras definiciones.

declaración de importación:

> Una declaración que lee un archivo de módulo y crea un objeto de módulo.

objeto de módulo:

> Un valor creado por una declaración `import` que proporciona acceso a los valores definidos en un módulo.

notación de puntos:

> La sintaxis para llamar a una función en otro módulo al especificar 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 declaración como parte de una declaración más grande.

flujo de ejecución:

Orden de ejecución de las instrucciones.

diagrama de stack:

Una representación gráfica de una pila de funciones, sus variables y los valores a los que se refieren.

marco:

Un cuadro en un diagrama de stack que representa una llamada de función. Contiene las variables locales y los parámetros de la función.

traceback:

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

3.14 Ejercicios

Ejercicio 3-1.

Escriba una función llamada `right_justify` que toma una cadena nombrada `s` como parámetro e imprime la cadena con suficientes espacios iniciales para que la última letra de la cadena esté en la columna 70 de la pantalla:

```
>>> right_justify('monty')
                                                                Monty
```

Sugerencia: utilice la concatenación de cadenas y la repetición. 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 puede asignar a una variable o pasar como un argumento. Por ejemplo, `do_twice` es una función que toma un objeto de función como argumento y lo llama dos veces:

```
def do_twice (f):
    f ()
    f ()
```

Aquí hay un ejemplo que usa `do_twice` para llamar a una función nombrada `print_spam` dos veces:

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


do_twice (print_spam)
```

1. Escriba este ejemplo en un script y pruébelo.

2. Modifique `do_twice` para que tome dos argumentos, un objeto de función y un valor, y llama a la función dos veces, pasando el valor como un argumento.

3. Copie la definición de `print_twice` mostrada anteriormente en este capítulo a su secuencia de comandos.

4. Use la versión modificada de `do_twice` para llamar a `print_twice` dos veces, pasando `'spam'` como un argumento.

5. Defina una nueva función llamada `do_four` que toma un objeto de función y un valor y llama a la función cuatro veces, pasando el valor como parámetro. Debería haber solo dos enunciados en el cuerpo de esta función, no cuatro.

Solución: *http://thinkpython2.com/code/do_four.py*.

Ejercicio 3-3.

Nota: Este ejercicio debe hacerse usando solo las declaraciones y otras características que hemos aprendido hasta ahora.

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

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

   Sugerencia: para imprimir más de un valor en una línea, puede imprimir una secuencia de valores separados por comas:

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

   De forma predeterminada, `print` avanza a la siguiente línea, pero puede anular ese comportamiento y poner un espacio al final, como este:

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

El resultado de estas declaraciones es '+ -'.

Una declaración `print` sin argumento termina la línea actual y va a 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 se basa en un ejercicio en Oualline, *Programación Práctica C*, Tercera Edición , O'Reilly Media, 1997.

Capítulo 4

Caso de estudio: Diseño de Interfaz

Este capítulo presenta un caso de estudio que demuestra un proceso para diseñar funciones que funcionan juntas.

Introduce el módulo `turtle`, que te permite crear imágenes usando gráficos de tortuga. El módulo `turtle` está incluido en la mayoría de las instalaciones de Python, pero si está ejecutando Python usando PythonAnywhere, no podrá ejecutar los ejemplos de tortuga (al menos no podría cuando escribí esto).

Si ya ha instalado Python en su computadora, debería poder ejecutar los ejemplos. De lo contrario, ahora es un buen momento para instalar. He publicado instrucciones en *http://tinyurl.com/thinkpython2e*.

Los ejemplos de código 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 ejecuta este código, debe crear una nueva ventana con una pequeña flecha que represente a la tortuga. Cerrar la ventana.

Cree un archivo con nombre `mypolygon.py` y escriba el siguiente código:

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

El módulo `turtle` (con una *t* minúscula) proporciona una función llamada `Turtle` (con una *T* mayúscula) que crea un objeto Turtle, que asignamos a una variable llamada bob. La impresión de bob muestra algo como:

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

Esto significa que se `bob` refiere a un objeto de 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 hacer excepto cerrar la ventana.

Una vez que creas una Tortuga, puedes llamar a un método para moverla por la ventana. Un método es similar a una función, pero usa una sintaxis ligeramente diferente. Por ejemplo, para mover la tortuga hacia adelante:

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

El método, `fd` está asociado con el objeto de tortuga que estamos llamando con `bob`. Llamar a un método es como hacer una solicitud: estás pidiendo que `bob` siga adelante.

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

Otros métodos que puedes utilizar para una Tortuga son `bk` retroceder, `lt` girar a la izquierda y `rt` girar a la derecha. El argumento para `lt` y `rt` es un ángulo en grados.

Además, cada Tortuga sostiene un bolígrafo, que está hacia abajo o hacia arriba; si la pluma está abajo, la Tortuga deja un rastro cuando se mueve. Los métodos `pu` y `pd` significan “pluma arriba” y “pluma abajo”.

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

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

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

Ahora modifica el programa para dibujar un cuadrado. ¡No continúes hasta que lo tengas funcionando!

4.2 Repetición simple

Es probable que haya escrito algo como esto:

```
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 forma más concisa con una declaración `for`. Agregue este ejemplo a mypolygon.py y ejecútelo de nuevo:

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

Debería ver algo como esto:

```
Hello!
Hello!
Hello!
Hello!
```

Este es el uso más simple de la declaración `for`; veremos más más tarde. Pero eso debería ser suficiente para permitirle reescribir su programa de dibujo cuadrado. No continúes hasta que lo hagas.

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

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

La sintaxis de una declaración `for` es similar a una definición de función. Tiene un encabezado que termina con dos puntos y un cuerpo sangrado. El cuerpo puede contener cualquier cantidad de declaraciones.

Una declaración `for` también se llama bucle porque el flujo de ejecución se ejecuta a través del cuerpo y luego vuelve a la parte superior. En este caso, ejecuta el cuerpo cuatro veces.

Esta versión es en realidad un poco diferente del código de dibujo cuadrado anterior porque hace otro giro después de dibujar el último lado del cuadrado. El turno adicional lleva más tiempo, pero simplifica el código si hacemos lo mismo cada vez a través del ciclo. Esta versión también tiene el efecto de dejar a la tortuga atrás en la posición inicial, mirando hacia la dirección de inicio.

4.3 Ejercicios

La siguiente es una serie de ejercicios que usan TurtleWorld. Están destinados a ser divertidos, pero tienen un punto, también. Mientras trabajas en ellos, piensa cuál es el punto.

Las siguientes secciones tienen soluciones para los ejercicios, por lo tanto, no mire hasta que haya terminado (o al menos intentado).

1. Escribe una función llamada `square` que toma un parámetro llamado `t`, que es una tortuga. Debería usar la tortuga para dibujar un cuadrado.

   Escriba una llamada a función que pase `bob` como un argumento a `square`, y luego ejecute el programa nuevamente.

2. Agregue otro parámetro, nombrado `length`, a `square`. Modifique el cuerpo para que la longitud de los lados sea `length`, y luego modifique la llamada a la función para proporcionar un segundo argumento. Ejecuta el programa de nuevo. Pruebe su programa con un rango de valores para `length`.

3. Haga una copia de `square` y cambie el nombre a `polygon`. Agregue otro parámetro llamado `n` y modifique el cuerpo para que dibuje un polígono regular `n-lados`.

   Sugerencia: los ángulos exteriores de un polígono regular de n-lados son *360/n* grados.

4. Escribe una función llamada circle que toma una tortuga, ty un radio, r como parámetros y que dibuja un círculo aproximado llamando polygon con una longitud y número de lados apropiados. Pruebe su función con un rango de valores de r.

   Sugerencia: descubra la circunferencia del círculo y asegúrese de eso *length * n = circunferencia*.

5. Cree una versión más general de `circle` llamada `arc` que tome un parámetro adicional llamado `angle`, que determina qué fracción de un círculo dibujar. `angle` está en unidades de grados, entonces cuando `angle=360`, `arc` debería dibujar un círculo completo.

4.4 Encapsulación

El primer ejercicio le pide que ponga su código de dibujo cuadrado en una definición de función y luego llame a la función, pasando a la tortuga como un parámetro. Aquí hay una solución:

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


square(bob)
```

Las instrucciones más internas, `fd` y `lt`, se sangran dos veces para mostrar que están dentro del bucle `for`, que está dentro de la definición de la función. La siguiente línea `square(bob)`, está alineada con el margen izquierdo, que indica el final del bucle `for` y la definición de la 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 para `square`:

```
alice = Turtle()
square(alice)
```

Envolver un trozo de código en una función se llama **encapsulación**. Uno de los beneficios de la encapsulación es que agrega un nombre al código, que sirve como un tipo de documentación. Otra ventaja es que si reutiliza 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 `length` a `square`. Aquí hay una solución:

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


square(bob, 100)
```

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

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

```
def polygon(t, n, length):
    angle = 360 / n
    for i in range(n):
        t.fd(length)
        t.lt(angle)


polygon(bob, 7, 70)
```

Este ejemplo dibuja un polígono de 7 lados con una longitud de lado 70.

Si está utilizando Python 2, el valor de `angle` podría estar desactivado debido a la división de enteros. Una solución simple es calcular `angle = 360.0 / n`. Como el numerador es un número de coma flotante, el resultado es un punto flotante.

Cuando una función tiene más de unos pocos argumentos numéricos, es fácil olvidar lo que 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:

```
polygon(bob, n=7, length=70)
```

Estos se denominan argumentos de palabra clave porque incluyen los nombres de los parámetros como "palabras clave" (que no deben confundirse con las palabras clave de Python como `while` y `def`).

Esta sintaxis hace que el programa sea más legible. También es un recordatorio de cómo funcionan los argumentos y los parámetros: cuando llama a una función, los argumentos se asignan a los parámetros.

4.6 Diseño de interfaz

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

```
import math


def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

Ahora el número de segmentos es un número entero cercano a `circumference/3`, por lo que la longitud de cada segmento es aproximadamente 3, lo suficientemente pequeña como para que los círculos se vean bien, pero lo suficientemente grandes como para ser eficientes y aceptables para cualquier tamaño de círculo.

4.7 Refactoring

Cuando escriba `circle`, pude reutilizar polygon porque un polígono de muchos lados es una buena aproximación de un círculo. Pero `arc` no es tan cooperativo; no podemos usar `polygon` o `circle` para dibujar un arco.

Una alternativa es comenzar con una copia de `polygon` y transformarla en `arc`. El resultado puede verse así:

```
def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

La segunda mitad de esta función se ve `polygon`, pero no podemos reutilizar `polygon` sin cambiar la interfaz. Podríamos generalizar `polygon` para tomar un ángulo como un tercer argumento, ¡pero `polygon` ya no sería un nombre apropiado! En cambio, llamemos a la función más general `polyline`:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Ahora podemos reescribir `polygon` y `arc` usando `polyline`:

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)


def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Finalmente, podemos reescribir `circle` para usar `arc`:

```
def circle(t, r):
```

```
    arc(t, r, 360)
```

Este proceso de reorganizar un programa para mejorar las interfaces y facilitar la reutilización de código se denomina refactorización. En este caso, notamos que había un código similar en `arc` y `polygon`, así que lo "descompusimos" en `polyline`.

Si hubiéramos planeado con anticipación, podríamos haber escrito `polyline` primero y evitado la refactorización, pero a menudo no se sabe lo suficiente al comienzo de un proyecto para diseñar todas las interfaces. Una vez que comienzas a codificar, entiendes mejor el problema. A veces refactorizar 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 "encapsulación y generalización". Los pasos de este proceso son:

1. Comience por escribir un pequeño programa sin definiciones de funciones.

2. Una vez que tenga el programa funcionando, identifique una pieza coherente, encapsule la pieza en una función y asígnele un nombre.

3. Generalice la función agregando los parámetros apropiados.

4. Repita los pasos 1-3 hasta que tenga un conjunto de funciones de trabajo. Copie y pegue código de trabajo para evitar volver a escribir (y volver a depurar).

5. Busque oportunidades para mejorar el programa mediante refactorización. Por ejemplo, si tiene un código similar en varios lugares, considere factorizarlo en una función general apropiada.

Este proceso tiene algunos inconvenientes, veremos alternativas más adelante, pero puede ser útil si no sabe de antemano cómo dividir el programa en funciones. Este enfoque te permite diseñar sobre la marcha.

4.9 Docstring

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

```
def polyline(t, n, length, angle):
    """Draws n line segments with the given length and
    angle (in degrees) between them.  t is a turtle.
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Por convención, todas las cadenas de docs son cadenas de comillas triples, también conocidas como cadenas de líneas múltiples porque las comillas triples permiten que la cadena abarque más de una línea.

Es escueto, pero contiene la información esencial que alguien necesitaría para usar esta función. Explica de forma concisa qué hace la función (sin entrar en detalles sobre cómo lo hace). Explica qué efecto tiene cada parámetro en el comportamiento de la función y qué tipo debe 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 debe ser simple de explicar; si tiene dificultades para explicar una de sus funciones, tal vez la interfaz podría mejorarse.

4.10 Depuración

Una interfaz es como un contrato entre una función y una persona que llama. La persona que llama acepta proporcionar ciertos parámetros y la función acepta realizar cierto trabajo.

Por ejemplo, `polyline` requiere cuatro argumentos: `t` tiene que ser una Tortuga; `n` tiene que ser un número entero; `length` debería ser un número positivo; y `angle` tiene que ser un número, que se entiende en grados.

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

Las condiciones previas son responsabilidad de quien llama. Si la persona que llama viola una condición previa (documentada adecuadamente) y la función no funciona correctamente, el error está en la persona que llama, no en la función.

Si se cumplen las condiciones previas y las postcondiciones no, el error está en la función. Si sus condiciones previas y posteriores son claras, pueden ayudar con la depuración.

4.11 Glosario

método:

> Una función que se asocia con un objeto y se llama mediante notación de puntos.

bucle:

> Una parte de un programa que puede ejecutarse repetidamente.

encapsulación:

> El proceso de transformar una secuencia de enunciados 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 como una "palabra clave".

interfaz:

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

refactorización:

El proceso de modificar un programa de trabajo 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.

condición previa:

Un requisito que debe cumplir la persona que llama antes de que comience una función.

postcondición:

Un requisito que debe cumplir la función antes de que finalice.

4.12 Ejercicios

Ejercicio 4-1.

Descargue el código en este capítulo de *http://thinkpython2.com/code/polygon.py*.

1. Dibuje un diagrama de pila que muestre el estado del programa durante la ejecución `circle(bob, radius)`. Puede hacer la aritmética a mano o agregar unas declaraciones `print` al código.

2. La versión de `arc` en *"Refactoring"* no es muy preciso debido a que la aproximación lineal del círculo es siempre fuera del círculo verdadero. Como resultado, la Tortuga termina a unos pocos píxeles del destino correcto. Mi solución muestra una forma de reducir el efecto de este error. Lea el código y vea si tiene sentido para usted. Si dibuja un diagrama, puede ver cómo funciona.

Ejercicio 4-2.

Escriba un conjunto de funciones apropiadamente general que pueda dibujar flores como en la Figura 4-1.

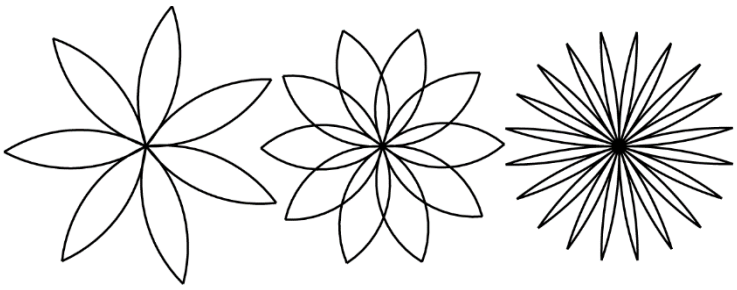

Figura 4-1. Flores de tortuga.

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

Ejercicio 4-3.

Escriba un conjunto de funciones apropiadamente general que pueda dibujar formas como en la Figura 4-2.

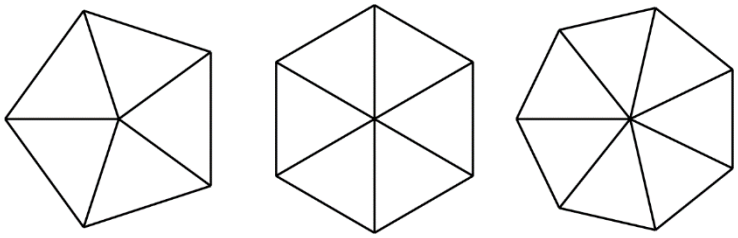

Figura 4-2. Pasteles de tortuga.

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

Ejercicio 4-4.

Las letras del alfabeto se pueden construir a partir de un número moderado de elementos básicos, como líneas verticales y horizontales y algunas curvas. Diseñe un alfabeto que se pueda dibujar con un número mínimo de elementos básicos y luego escriba funciones que dibujen las letras.

Usted debe escribir una función para cada letra, con nombres `draw_a`, `draw_b`, etc., y ponga sus funciones en un archivo llamado `letters.py`. Puede descargar una "máquina de escribir de tortuga" de *http://thinkpython2.com/code/typewriter.py* para ayudarlo a probar su código.

Puede obtener una solución de *http://thinkpython2.com/code/letters.py*; también requiere *http://thinkpython2.com/code/polygon.py*.

Ejercicio 4-5.

Lea sobre espirales en *http://en.wikipedia.org/wiki/Spiral*; luego escribe un programa que dibuje una espiral de Archimedian (o uno de los otros tipos).

Solución: *http://thinkpython2.com/code/spiral.py*.

Capítulo 5

Condicionales y Recursión

El tema principal de este capítulo es la declaración `if`, que ejecuta un código diferente según el estado del programa. Pero primero quiero presentar dos nuevos operadores: división de piso y módulo.

5.1 División de piso y módulo

El operador de **división de piso**, `//` divide dos números y redondea a un número entero. Por ejemplo, supongamos que el tiempo de ejecución de una película es de 105 minutos. Es posible que desee saber cuánto tiempo es en horas. La división convencional devuelve un número de punto flotante:

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

Pero normalmente no escribimos horas con puntos decimales. La división del piso devuelve el número entero de horas, dejando caer la parte de la fracción:

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

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

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

Una alternativa es utilizar el **operador de módulo**, % que 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, puede verificar si un número es divisible por otro, si `x % y` es cero, entonces `x` es divisible por `y`.

Además, puede extraer el dígito o los dígitos del más a la derecha de un número. Por ejemplo, `x % 10` arroja el dígito más a la derecha de `x` (en la base 10). Del mismo modo `x % 100` produce los dos últimos dígitos.

Si está usando Python 2, la división funciona de manera diferente. El operador de división, /realiza la división de piso si ambos operandos son enteros, y la división de coma flotante si cualquiera de los operandos es a `float`.

5.2 Expresiones Booleanas

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

```
>>> 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 que o igual a y
x <= y # x es menor igual o igual a y
```

Aunque estas operaciones probablemente le resulten familiares, los símbolos de Python son diferentes de los símbolos matemáticos. Un error común es usar un solo signo igual (=) en lugar de un signo doble igual (==). Recuerde que = es un operador de asignación y == es un operador relacional. No existe tal cosa =< o =>.

5.3 Operadores lógicos

Hay 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 verdadero solo si x es mayor que 0 y menor que 10.

`n%2 == 0 or n%3 == 0` es verdadero si una o ambas de las condiciones son verdaderas, es decir, si el número es divisible entre 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 a `y`.

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

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

```
True
```

Esta flexibilidad puede ser útil, pero hay algunas sutilezas que pueden ser confusas. Es posible que desee evitarlo (a menos que sepa lo que está haciendo).

5.4 Ejecución condicional

Para escribir programas útiles, casi siempre necesitamos la capacidad de verificar las condiciones y cambiar el comportamiento del programa en consecuencia. Las **declaraciones condicionales** nos dan esta habilidad. La forma más simple es la declaración `if`:

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

La expresión booleana posterior `if` se llama **condición**. Si es verdadero, se ejecuta la instrucción sangrada. Si no, no pasa nada.

Las declaraciones `if` tienen la misma estructura que las definiciones de funciones: un encabezado seguido de un cuerpo sangrado. Las declaraciones como esta se llaman **declaraciones compuestas**.

No hay límite en la cantidad de declaraciones que pueden aparecer en el cuerpo, pero tiene que haber al menos una. Ocasionalmente, es útil tener un cuerpo sin declaraciones (por lo general, como guardián de lugar para el código que aún no ha escrito). En ese caso, puede usar la declaración `pass`, que no hace nada.

```
if x < 0:
    pass # TODO: ¡necesita manejar valores negativos!
```

5.5 Ejecución alternativa

Una segunda forma de la declaración `if` es la "ejecución alternativa", en la que 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 cuando x está dividido por 2 es 0, entonces sabemos que x es par, y el programa muestra un mensaje apropiado. Si la condición es falsa, se ejecuta el segundo conjunto de instrucciones. Como 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 Condiciones encadenadas

A veces hay más de dos posibilidades y necesitamos más de dos ramas. Una forma de expresar un cálculo como ese es un **condicional concatenado**:

```
if x < y:
    print ('x es menor que y')
elif x> y:
```

```
    print ('x es mayor que y')
else:
    print ('x y y son iguales')
```

`elif` es una abreviatura de "else if". Nuevamente, exactamente una rama se ejecutará. No hay límite en el número de declaraciones `elif`. Si hay una cláusula `else`, tiene que ser al final, pero no tiene que haber una.

```
if choice == 'a':
    draw_a ()
elif choice == 'b':
    draw_b ()
elif choice == 'c':
    draw_c ()
```

Cada condición se verifica en orden. Si el primero es falso, el siguiente está marcado, y así sucesivamente. Si uno de ellos es verdadero, la rama correspondiente se ejecuta y la instrucción finaliza. Incluso si más de una condición es verdadera, solo se ejecuta la primera rama verdadera.

5.7 Acontecimientos anidados

Un condicional también se puede anidar dentro de otro. Podríamos haber escrito el ejemplo en la sección anterior de esta manera:

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

El condicional externo contiene dos ramas. La primera rama contiene una declaración simple. La segunda rama contiene otra instrucción `if`, que tiene dos ramas propias. Esas dos ramas son declaraciones simples, aunque también podrían haber sido declaraciones condicionales.

Aunque la sangría de las declaraciones hace que la estructura sea aparente, los **condicionales anidados** se vuelven difíciles de leer muy rápidamente. Es una buena idea evitarlos cuando puedas.

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

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

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

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

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

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

5.8 Recursión

Es legal que una función llame a otra; también es legal que una función se llame a sí misma. Puede que no sea obvio por qué es algo bueno, pero resulta ser una de las cosas más mágicas que un programa puede hacer. Por ejemplo, mira la siguiente función:

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Si `n` es 0 o negativo, emite la palabra, "¡Blastoff!". De lo contrario, emite `n` y luego llama a una función llamada `countdown` -así mismo- y pasa `n-1` como un argumento.

¿Qué sucede si llamamos a esta función así?

```
>>> countdown(3)
```

La ejecución de `countdown` *comienza con* `n=3`*, y como n es mayor que 0, emite el valor 3, y luego se llama a sí mismo ...*

> *La ejecución de* `countdown` *comienza con* `n=2`*, y como n es mayor que 0, emite el valor 2, y luego se llama a sí mismo ...*
>
> > *La ejecución de* `countdown` *comienza con* `n=1`*, y como n es mayor que 0, emite el valor 1, y luego se llama a sí mismo ...*
> >
> > > *La ejecución de* `countdown` *comienza con* `n=0`*, y como n no es mayor que 0, emite la palabra, "¡Blastoff!" Y luego regresa.*
> >
> > *El* `countdown` *que obtuvo* `n=1` *de retorno.*
>
> *El* `countdown` *que obtuvo* `n=2` *de retorno.*

El `countdown` *que obtuvo* `n=3` *de retorno.*

Y luego estás de regreso __main__. Entonces, el resultado total se ve así:

```
3
2
1
Blastoff!
```

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

Como otro ejemplo, podemos escribir una función que imprime una cadena por `n` tiempos:

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

Si `n <= 0` la declaración de devolución sale de la función. El flujo de ejecución vuelve inmediatamente a la persona que llama, y las líneas restantes de la función no se ejecutan.

El resto de la función es similar a `countdown`: muestra `s` y luego se llama para mostrar `s` n-1 veces adicionales. Entonces la cantidad de líneas de salida es `1 + (n - 1)`, lo que se suma a `n`.

Para ejemplos simples como este, probablemente sea más fácil usar un bucle `for`. Pero veremos ejemplos más adelante que son difíciles de escribir con un bucle `for` y fáciles de escribir con recursión, por lo que es bueno comenzar temprano.

5.9 Diagramas de stack para funciones recursivas

En "Diagramas de stack" o diagramas de pilas, es usado para representar el estado de un programa durante una llamada de función. El mismo tipo de diagrama puede ayudar a interpretar una función recursiva.

Cada vez que se llama a una función, Python crea un marco para contener las variables locales y los parámetros de la función. Para una función recursiva, puede haber más de un cuadro en la pila al mismo tiempo.

La Figura 5-1 muestra un diagrama de pila para `countdown` llamado con `n = 3`.

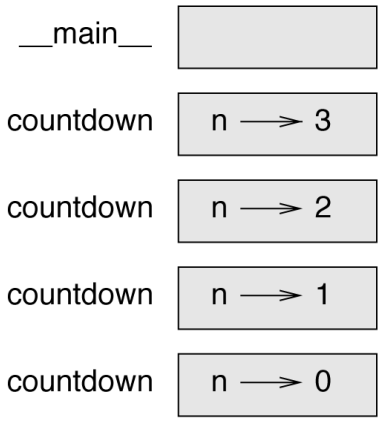

Figura 5-1. Diagrama de pila.

Como de costumbre, la parte superior de la pila es el marco para `__main__`. Está vacío porque no creamos ninguna variable en `__main__` ni le pasamos ningún argumento.

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

Como ejercicio, dibuja un diagrama de pila para `print_n` invocado con `s = 'Hello'` y `n=2`. Luego escriba una función llamada `do_n` que toma un objeto de función y un número `n`, como argumentos, y que llama a la función dada `n` veces.

5.10 Recursión infinita

Si una recursión nunca llega a un caso base, continúa realizando llamadas recursivas para siempre, y el programa nunca termina. Esto se conoce como **recursión infinita**, y generalmente no es una buena idea. Aquí hay un programa mínimo con una recursión infinita:

```
def recurse ():
    recurse ()
```

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

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

Este rastreo es un poco más grande que el que vimos en el capítulo anterior. Cuando se produce el error, ¡hay 1,000 marcos en la pila recurse!

Si escribe una recursión infinita por accidente, revise su función para confirmar que hay un caso base que no hace una llamada recursiva. Y si hay un caso base, verifique si está garantizado para alcanzarlo.

5.11 Entrada de teclado

Los programas que hemos escrito hasta el momento no aceptan ninguna entrada del usuario. Simplemente hacen lo mismo todo el tiempo.

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 se reanuda y `input` devuelve lo que el usuario tipeó como una cadena. En Python 2, se llama a la misma función `raw_input`.

```
>>> text = input ()
¿Qué estás esperando?
```

```
>>> texto
¿Qué estás esperando?
```

Antes de recibir información del usuario, es una buena idea imprimir un mensaje indicándole al usuario qué escribir. `input` puede tomar un aviso como argumento:

```
>>> name = input ('¿Cuál ... es tu nombre? \ n')
¿Cuál ... es tu nombre?
¡Arthur, rey de los británicos!
>>> name
¡Arthur, Rey de los Británicos!
```

La secuencia `\n` al final del prompt representa una **nueva línea**, que es un carácter especial que causa un salto de línea. Es por eso que la entrada del usuario aparece debajo del aviso.

Si espera que el usuario escriba un número entero, puede intentar convertir el valor devuelto en `int`:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

Pero si el usuario escribe algo que no sea una cadena de dígitos, se obtiene un error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

Veremos cómo manejar este tipo de error más adelante.

5.12 Depuración

Cuando se produce un error de sintaxis o tiempo de ejecución, el mensaje de error contiene mucha información, pero puede ser abrumador. Las partes más útiles son usualmente:

- Qué tipo de error fue
- Donde ocurrió

Los errores de sintaxis suelen ser fáciles de encontrar, pero hay algunos errores. Los errores de espacios en blanco pueden ser complicados porque los espacios y las pestañas 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 sangrada por un espacio. Pero el mensaje de error apunta a `y`, que es engañoso. En general, los mensajes de error indican dónde se descubrió el problema, pero el error real podría ser anterior en el código, a veces en una línea anterior.

Lo mismo es cierto de los errores de tiempo de ejecución. Supongamos que está tratando de calcular una relación señal / ruido en decibelios. La fórmula es $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$. En Python, puedes escribir algo como esto:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

Cuando ejecuta este programa, obtiene una excepción:

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

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

Debe tomarse el tiempo para leer los mensajes de error con cuidado, pero no asuma que todo lo que dicen es correcto.

5.13 Glosario

división del piso:

Un operador, denotado //, que divide dos números y redondea hacia abajo (hacia cero) a un número entero.

operador de módulo:

Un operador, denotado con un signo de porcentaje (%), que trabaja en enteros y devuelve el resto cuando un número se divide por otro.

expresión booleana:

Una expresión cuyo valor es `True` o bien o `False`.

operador relacional:

Uno de los operadores que comparan sus operandos: ==, !=, >, <, >=, y <=.

operador lógico:

Uno de los operadores que combinan expresiones booleanas: `and`, `or`, y `not`.

sentencia condicional:

Una declaración que controla el flujo de ejecución dependiendo de alguna condición.

condición:

La expresión booleana en una declaración condicional que determina qué rama se ejecuta.

declaración compuesta:

Una declaración que consiste en un encabezado y un cuerpo. El encabezado termina con dos puntos (:). El cuerpo está tabulado en relación con el encabezado.

rama:

Una de las secuencias alternativas de declaraciones en una declaración condicional.

encadenado condicional:

Una declaración condicional con una serie de ramas alternativas.

condicional anidado:

Una declaración condicional que aparece en una de las ramas de otra instrucción condicional.

Declaración de retorno:

Una declaración que hace que una función termine inmediatamente y regrese a la persona que llama.

recursión:

El proceso de llamar a la función que se está ejecutando actualmente.

caso base:

Una rama condicional en una función recursiva que no hace una llamada recursiva.

recursión infinita:

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

5.14 Ejercicios

El módulo `time` proporciona una función, también llamada `time`, que devuelve el tiempo medio actual de Greenwich en "la época", que es un tiempo arbitrario utilizado como punto de referencia. En los sistemas UNIX, la época es el 1 de enero de 1970.

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

Escriba una secuencia de comandos que lea la hora actual y la convierta a una hora del día en horas, minutos y segundos, más el 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 llamada `check_fermat` que tome cuatro parámetros `a`, y `b`, `c` y `n`, y comprueba si el teorema de Fermat se cumple. Si `n` es mayor que 2 y

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

   el programa debería imprimir, "Holy Smokes, Fermat was wrong!" De lo contrario, el programa debería imprimir, "No, eso no funciona".

2. Escribir una función que indica al usuario que ingrese los valores para `a`, `b`, `c` y `n`, los convierte en números enteros, y utiliza `check_fermat` para comprobar si infringen teorema de Fermat.

Ejercicio 5-3.

Si le dan tres palos, puede o no puede organizarlos en un triángulo. Por ejemplo, si uno de los palos mide 12 pulgadas de largo y los otros dos tienen una pulgada de largo, no podrá lograr que los palos cortos se encuentren en el medio. Para tres longitudes, 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, puedes. (Si la suma de dos longitudes es igual a la tercera, forman lo que se llama un triángulo "degenerado").*

1. Escriba una función llamada `is_triangle` que toma tres enteros como argumentos, y que imprime "Sí" o "No", dependiendo de si puede o no puede formar un triángulo a partir de barras con las longitudes dadas.

2. Escriba una función que le pida al usuario que ingrese tres longitudes de barras para `is_triangle`, las convierta en enteros y las use para verificar si las barras con las longitudes dadas pueden formar un triángulo.

Ejercicio 5-4.

¿Cuál es el resultado del siguiente programa? Dibuje un diagrama de pila que muestra el estado del programa cuando imprime el resultado.

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


recurse(3, 0)
```

1. ¿Qué pasaría si llamaras a esta función así `recurse(-1, 0):` ?

2. Escribe un docstring que explique todo lo que alguien debería saber para utilizar esta función (y nada más).

Los siguientes ejercicios usan el módulo `turtle`, descrito en el Capítulo 4:

Ejercicio 5-5.

Lea la siguiente función y vea si puede descubrir lo que hace (vea los ejemplos en el Capítulo). Luego ejecútelo y vea si lo hizo bien.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

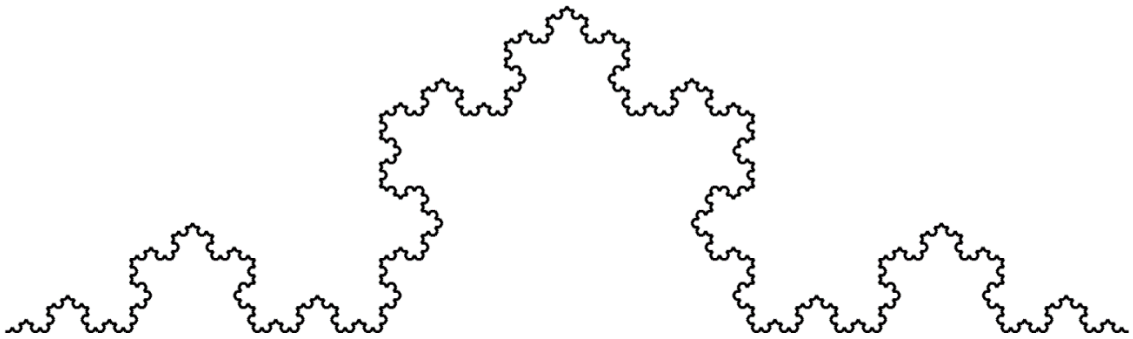

Figura 5-2. Una curva de Koch.

Ejercicio 5-6.

La curva de Koch es un fractal que se parece a la figura 5-2. Para dibujar una curva de Koch con una longitud `x`, todo lo que tiene que hacer es:

1. Dibuja una curva de Koch con una longitud x / 3 .
2. Gire a la izquierda 60 grados.
3. Dibuja una curva de Koch con una longitud x / 3 .
4. Gire a la derecha 120 grados.
5. Dibuja una curva de Koch con una longitud x / 3 .
6. Gire a la izquierda 60 grados.
7. Dibuja una curva de Koch con una longitud x / 3 .

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

1. Escribe una función llamada `koch` que toma una tortuga y una longitud como parámetros, y que usa la tortuga para dibujar una curva de Koch con la longitud dada.
2. Escribe una función llamada `snowflake` que dibuje tres curvas de Koch para hacer el contorno de un copo de nieve.

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

3. La curva de Koch se puede generalizar de varias maneras. Consulte *http://en.wikipedia.org/wiki/Koch_snowflake* para ver ejemplos e implemente su favorito.

Capítulo 6

Funciones fructíferas

Muchas de las funciones de Python que hemos utilizado, como las funciones matemáticas, producen valores devueltos. Pero las funciones que hemos escrito son todas vacías: 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 fructíferas.

6.1 Valores de retorno

Llamar a la función genera un valor de retorno, que generalmente asignamos a una variable o usamos como parte de una expresión.

```
e = math.exp (1.0)
height = radius * math.sin (radianes)
```

Las funciones que hemos escrito hasta ahora son nulas. Hablando casualmente, no tienen valor de retorno; más precisamente, su valor de retorno es `None`.

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

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

Hemos visto la declaración `return` antes, pero en una función fructífera, la declaración `return` incluye una expresión. Esta declaración significa: "Regrese inmediatamente de esta función y use 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(radius):
    return math.pi * radius**2
```

Por otro lado, las variables temporales como `a` pueden facilitar la depuración.

Algunas veces es útil tener múltiples declaraciones de retorno, una en cada rama de un condicional:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
```

```
        return x
```

Como estas declaraciones están en una condicional alternativa, solo se ejecuta una.

Tan pronto como se ejecuta una declaración de retorno, la función finaliza sin ejecutar ninguna declaración posterior. El código que aparece después de una declaración `return`, o en cualquier otro lugar al que nunca llegue el flujo de ejecución, se llama **código muerto**.

En una función fructífera, es una buena idea asegurarse de que cada ruta posible a través del programa llegue a una declaración `return`. Por ejemplo:

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

Esta función es incorrecta porque si `x` pasa a ser 0, ninguna condición es verdadera y la función finaliza sin tocar una instrucción `return`. Si el flujo de ejecución llega al final de una función, el valor de retorno es `None`, que no es el valor absoluto de 0:

```
>>> absolute_value (0)
None
```

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

Como ejercicio, escriba una función llamada `compare` que toma dos valores, `x` y `y`, y devuelve 1 `if x > y`, 0 `if x == y` y -1 `if x < y`.

6.2 Desarrollo incremental

A medida que escribe funciones más grandes, puede pasar más tiempo depurando.

Para lidiar con programas cada vez más complejos, es posible que desee probar un proceso llamado **desarrollo incremental**. El objetivo del desarrollo incremental es evitar largas sesiones de depuración agregando y probando solo una pequeña cantidad de código a la vez.

Como ejemplo, suponga que desea encontrar la distancia entre dos puntos, dada por las coordenadas (x_1, y_1) y (x_2, y_2). Según el teorema de Pitágoras, la distancia es:

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

El primer paso es considerar cómo `distance` debería ser una función 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 puede representar utilizando cuatro números. El valor de retorno es la distancia representada por un valor de coma flotante.

Inmediatamente puede escribir un resumen de la función:

```
def distance(x1, y1, x2, y2):
```

```
    return 0.0
```

Obviamente, esta versión no calcula distancias; siempre devuelve cero. Pero es sintácticamente correcto y se ejecuta, lo que significa que puede probarlo antes de hacerlo más complicado.

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

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

Elegí estos valores para que la distancia horizontal sea 3 y la distancia vertical sea 4; de esa manera, el resultado es 5, la hipotenusa de un triángulo 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 a agregar código al cuerpo. Un próximo paso razonable es encontrar las diferencias de $x_2 - x_1$ y $y_2 - y_1$. La siguiente versión almacena esos valores en variables temporales y los imprime:

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

Si la función está funcionando, debería mostrar `dx = 3` y `dy = 4`. Si es así, sabemos que la función está obteniendo los argumentos correctos y realizando la primera computación correctamente. Si no, solo hay unas pocas líneas para verificar.

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

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

De nuevo, ejecutaría el programa en esta etapa y verificaría el resultado (que debería ser 25). Finalmente, puede usar `math.sqrt` para calcular y devolver el resultado:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

Si eso funciona correctamente, has terminado. De lo contrario, es posible que desee imprimir el valor de resultantes de la declaración de devolución.

La versión final de la función no muestra nada cuando se ejecuta; solo devuelve un valor. Las declaraciones `print` que escribimos son útiles para la depuración, pero una vez que obtienes la función funcionando, debes eliminarlas. Código como ese se llama andamiaje porque es útil para construir el programa pero no es parte del producto final.

Cuando comiences, deberías agregar solo una o dos líneas de código a la vez. A medida que adquiera más experiencia, podría encontrarse escribiendo y depurando trozos más grandes. De cualquier manera, el desarrollo incremental puede ahorrarle mucho tiempo de depuración.

Los aspectos clave del proceso son:

1. Comience con un programa de trabajo y realice pequeños cambios incrementales. En cualquier punto, si hay un error, debe tener una buena idea de dónde está.

2. Use variables para mantener valores intermedios para que pueda visualizarlos y verificarlos.

3. Una vez que el programa está funcionando, es posible que desee eliminar algunos de los andamios o consolidar declaraciones múltiples en expresiones compuestas, pero solo si no hace que el programa sea difícil de leer.

Como ejercicio, use el desarrollo incremental para escribir una función llamada `hypotenuse` que devuelve la longitud de la hipotenusa de un triángulo rectángulo dadas las longitudes de los otros dos lados como argumentos. Registre cada etapa del proceso de desarrollo a medida que avanza.

6.3 Composición

Como deberías esperar ahora, puedes llamar a una función desde otra. Como ejemplo, escribiremos una función que toma dos puntos, el centro del círculo y un punto en el perímetro, y calcula el área del círculo.

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

```
radius = distance(xc, yc, xp, yp)
```

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

result = area(radius)

Encapsulando estos pasos en una función, obtenemos:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

Las variables temporales `radius` y `result` son útiles para el desarrollo y la depuración, pero una vez que el programa está funcionando, podemos hacerlo más concisa mediante la composición de la función de llamadas:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Funciones Booleanas

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

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

Es común dar nombres de funciones booleanas que suenan como preguntas sí / no; `is_divisible` devuelve cualquiera `True` o `False` para indicar si `x` es divisible por `y`.

Aquí hay un ejemplo:

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

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

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

Las funciones booleanas a menudo se usan en declaraciones condicionales:

```
if is_divisible(x, y):
    print('x is divisible by y')
```

Puede ser tentador escribir algo como:

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

Pero la comparación adicional es innecesaria.

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

6.5 Más recursión

Solo cubrimos un pequeño subconjunto de Python, pero es posible que le interese saber que este subconjunto es un lenguaje de programación completo, lo que significa que todo lo que se pueda calcular se puede expresar en este idioma. Cualquier programa que se haya escrito se puede reescribir utilizando solo las características de idioma que haya aprendido hasta ahora (en realidad, necesitaría algunos comandos para controlar dispositivos como el mouse, discos, etc., pero eso es todo).

Probar que reclamar es un ejercicio no trivial realizado por primera vez por Alan Turing, uno de los primeros científicos de la computación (algunos dirían que él era matemático, pero muchos de los primeros científicos de la computación comenzaron como matemáticos). En consecuencia, se conoce como Tesis de Turing. Para una discusión más completa (y precisa) de la tesis de Turing, recomiendo el libro de Michael Sipser Introducción a la teoría de la computación (Course Technology, 2012).

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

vorpal:

> Un adjetivo utilizado para describir algo que es vorpal.

Si vio esa definición en el diccionario, podría estar molesto. ¡Por otro lado, si buscas la definición de la función factorial, denotada con el símbolo ! , puedes obtener algo como esto:

$$0! = 1$$

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

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 .

Entonces 3! es 3 por 2! , que es 2 por 1! , que es 1 por 0! . Poniéndolo todo junto, 3! es igual a 3 veces 2 veces 1 por 1, que es 6.

Si puede escribir una definición recursiva de algo, puede escribir un programa de Python para evaluarlo. El primer paso es decidir cuáles deberían ser los parámetros. En este caso, debe quedar claro que `factorial` toma un número entero:

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

Si el argumento pasa a ser 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:
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

El flujo de ejecución para este programa es similar al flujo de `countdown` en "Recursión". Si llamamos a `factorial` con el valor 3:

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

> *Como 2 no es 0, tomamos la segunda rama y calculamos el factorial de* `n-1`*...*
>
> > *Como 1 no es 0, tomamos la segunda rama y calculamos el factorial de* `n-1`*...*
> >
> > > *Como 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 de función que inició todo el proceso.

La Figura 6-1 muestra cómo se ve el diagrama de pila para esta secuencia de llamadas a funciones.

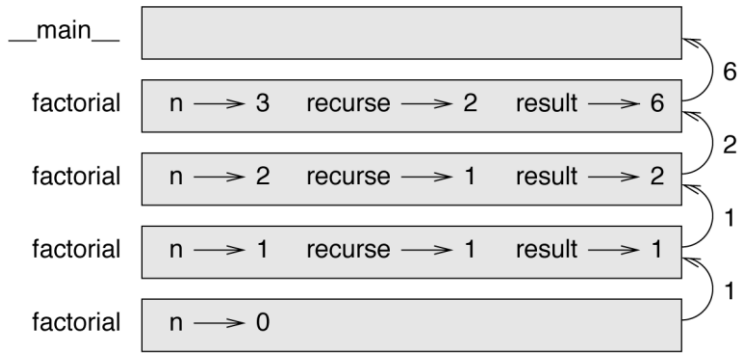

Figura 6-1. Diagrama de pila.

Los valores de retorno se muestran volviendo a pasar por la pila. En cada cuadro, el valor de retorno es el valor de `result`, que es el producto de `n` y `recurse`.

En el último cuadro, las variables locales `recurse` y `result` no existen, porque la rama que las crea no se ejecuta.

6.6 Salto de fe

Seguir el flujo de ejecución es una forma de leer programas, pero rápidamente puede ser abrumador. Una alternativa es lo que llamo el "salto de fe". Cuando llega a una llamada de función, en lugar de seguir el flujo de ejecución, supone que la función funciona correctamente y devuelve el resultado correcto.

De hecho, ya está practicando este acto de fe cuando usa funciones integradas. Cuando llama `math.cos` o `math.exp`, no examina los cuerpos de esas funciones. Simplemente asume que funcionan porque las personas que escribieron las funciones integradas eran buenos programadores.

Lo mismo es cierto cuando llamas a una de tus propias funciones. Por ejemplo, en "Funciones booleanas", escribimos una función llamada `is_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 las pruebas, podemos usar la función sin mirar el cuerpo de nuevo.

Lo mismo es cierto para los programas recursivos. Cuando llegue a la llamada recursiva, en lugar de seguir el flujo de ejecución, debe suponer que la llamada recursiva funciona (devuelve el resultado correcto) y luego preguntarse: "Suponiendo que pueda encontrar el factorial de n-1 , ¿puedo? calcula el factorial de n ? "Está claro que puedes, multiplicando por n.

Por supuesto, es un poco extraño suponer que la función funciona correctamente cuando no ha terminado de escribirla, ¡pero es por eso que se llama un acto de fe!

6.7 Un ejemplo más

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

$$fibonacci(0) = 0$$

$$fibonacci(1) = 1$$

$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$$

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 tratas de seguir el flujo de ejecución aquí, incluso para valores bastante pequeños de n, tu cabeza explota. Pero según el salto de la fe, si supone que las dos llamadas recursivas funcionan correctamente, entonces está claro que obtendrá el resultado correcto al sumarlas.

6.8 Comprobando tipos

¿Qué pasa si lo llamamos `factorial` y le damos 1.5 como argumento?

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

Parece una recursión infinita. ¿Como puede ser? La función tiene un caso base: cuándo `n == 0`. Pero si `n` no es un número entero, podemos perder el caso base y recurrir para siempre.

En la primera llamada recursiva, el valor de `n` es 0.5. En el siguiente, es -0.5. A partir de ahí, se hace más pequeño (más negativo), pero nunca será 0.

Tenemos dos opciones. Podemos tratar de generalizar la función `factorial` para trabajar con números de coma flotante, o podemos verificar el tipo de su argumento. La primera opción se llama función gamma y está un poco fuera del alcance de este libro. Así que iremos por el segundo.

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

```
def factorial (n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

El primer caso base maneja no enteros; el segundo maneja enteros negativos. En ambos casos, el programa imprime un mensaje de error y regresa `None` para indicar que algo salió mal:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

Si pasamos los dos controles, sabemos que n es positivo o cero, por lo que podemos probar que la recursión termina.

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

En "Búsqueda inversa" veremos una alternativa más flexible a la impresión de un mensaje de error: generar una excepción.

6.9 Depuración

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

- Hay algo mal con los argumentos que está obteniendo la función; una condición previa es violada.
- Hay algo mal con la función; una poscondición es violada.
- Hay algo mal con el valor de retorno o la forma en que se usa.

Para descartar la primera posibilidad, puede agregar una declaración `print` al comienzo de la función y visualizar los valores de los parámetros (y tal vez sus tipos). O puede escribir código que verifique las condiciones previas de forma explícita.

Si los parámetros se ven bien, agregue una declaración `print` antes de cada instrucción `return` y muestre el valor de retorno. Si es posible, verifique el resultado a mano. Considere llamar a la función con valores que faciliten la verificación del resultado (como en "Desarrollo incremental").

Si la función parece estar funcionando, observe la llamada a la función para asegurarse de que el valor de retorno se esté utilizando correctamente (¡o usado en absoluto!).

Agregar instrucciones de impresión al principio y al final de una función puede ayudar a que el flujo de ejecución sea más visible. Por ejemplo, aquí hay una versión de `factorial` con declaraciones de impresión:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

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

```
                factorial 4
```

```
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
```

Si está confundido acerca del flujo de ejecución, este tipo de resultado puede ser útil. Lleva cierto tiempo desarrollar andamios eficaces, pero un poco de andamio puede ahorrar mucha depuración.

6.10 Glosario

variable temporal:

> Una variable utilizada para almacenar un valor intermedio en un cálculo complejo.

código muerto:

> Parte de un programa que nunca se puede ejecutar, a menudo porque aparece después de un enunciado `return`.

desarrollo incremental:

> Un plan de desarrollo de programa destinado a evitar la eliminación de errores agregando y probando solo una pequeña cantidad de código a la vez.

andamio:

Código que se usa durante el desarrollo del programa, pero que no forma parte de la versión final.

guardián:

> Un patrón de programación que usa una instrucción condicional para verificar y manejar circunstancias que pueden causar 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
    square = b(total)**2
    return square


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

Ejercicio 6-2.

La función de Ackermann, $A(m, n)$ se define:

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

Ver *http://en.wikipedia.org/wiki/Ackermann_function*. Escriba una función nombrada que evalúe la función de Ackermann. Usa tu función para evaluar, que debería ser 125. ¿Qué ocurre con los valores más grandes de y ? `ackack(3, 4)mn`

Solución: *http://thinkpython2.com/code/ackermann.py*.

Ejercicio 6-3.

Un palíndromo es una palabra que se escribe igual hacia atrás y hacia adelante, como "oso" y "oro". Recursivamente, una palabra es un palíndromo si la primera y la última letra son iguales y el medio es un palíndromo.

Las siguientes son funciones que toman un argumento de cadena y devuelven las letras primera, última y media:

```
def first(word):
    return word[0]


def last(word):
    return word[-1]


def middle(word):
```

```
    return word[1:- 1]
```

Veremos cómo funcionan en el Capítulo 8.

1. Escriba estas funciones en un archivo llamado `palindrome.py` y pruébelas. ¿Qué pasa si llamas `middle` con una cadena con dos letras? ¿Una carta? ¿Qué pasa con la cadena vacía, que está escrita `''` y no contiene letras?

2. Escribe una función llamada `is_palindrome` que toma un argumento de cadena y devuelve `True` si es un palíndromo y de lo contrario retorna `False`. Recuerde que puede 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, uno, es una potencia de b si es divisible por b y a / b es una potencia de b . Escribir una función llamada `is_power` que toma parámetros `a` y `b` y vuelve `True` si a es una potencia de b. Nota: tendrá que pensar en el caso base.

Ejercicio 6-5.

El máximo común divisor (MCD) de un y b es el número más grande que divide ambos sin resto.

Una forma de encontrar el MCD de dos números se basa en la observación de que si r es el resto cuando a se divide por b , entonces $mcd(a, b) = mcd(b, r)$. Como caso base, podemos usar $mcd(a, 0) = a$.

Escribir una función llamada `mcd` que toma parámetros a y b, y devuelve su máximo común divisor.

Crédito: Este ejercicio se basa en un ejemplo de la Estructura e Interpretación de Programas de Computadora de Abelson y Sussman (MIT Press, 1996).

Capítulo 7

Iteración

Este capítulo trata sobre la iteración, que es la capacidad de ejecutar un bloque de enunciados repetidamente. Vimos un tipo de iteración, usando recursión, en "Recursión". Vimos otro tipo, utilizando un ciclo `for`, en "Repetición simple". En este capítulo veremos otro tipo más, usando una declaración `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 anterior).

```
>>> 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 la reasignación en un diagrama de estado.

En este punto quiero abordar una fuente común de confusión. Debido a que Python usa el signo igual (=) para la asignación, es tentador interpretar una afirmación a = b como una proposición matemática de igualdad; es decir, el enunciado 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 declaración a = 7 es legal y 7 = a no lo es.

Además, en matemáticas, una proposición de igualdad es verdadera o falsa para todos los tiempos. Si a = b ahora, entonces a siempre será igual a b. En Python, una declaración de asignación puede hacer que dos variables sean iguales, pero no tienen que permanecer de esa manera:

```
>>> a = 5
>>> b = a # a y b ahora son iguales
>>> 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 a menudo es útil, pero debe usarla con precaución. Si los valores de las variables cambian con frecuencia, puede hacer que el código sea difícil de leer y depurar.

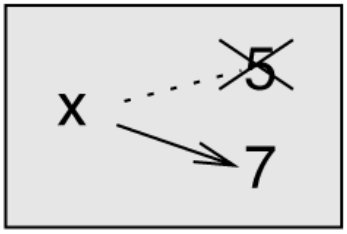

Figura 7-1. Diagrama de estado.

7.2 Actualizando variables

Un tipo común de reasignación es una actualización, donde el nuevo valor de la variable depende de la anterior.

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

Esto significa "obtener el valor actual de x, agregar uno y luego actualizar x con el nuevo valor".

Si intenta actualizar una variable que no existe, obtendrá un error, porque Python evalúa el lado derecho antes de asignarle un valor a x:

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

Antes de que pueda actualizar una variable, debe inicializarla, generalmente con una simple tarea:

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

Actualizar una variable agregando 1 se llama incremento; restar 1 se llama decremento.

7.3 La declaración while

Las computadoras se utilizan a menudo para automatizar tareas repetitivas. Repetir tareas idénticas o similares sin cometer errores es algo que las computadoras hacen bien y las personas lo hacen mal. En un programa de computadora, la repetición también se llama iteración.

Ya hemos visto dos funciones, `countdown` y `print_n`, iterando usando recursión. Debido a que la iteración es tan común, Python proporciona funciones de lenguaje para hacerlo más fácil. Una es la declaración `for` que vimos en "Repetición simple". Volveremos sobre eso más tarde.

Otra es la declaración `while`. Aquí hay una versión de `countdown` que usa una declaración `while`:

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
```

```
    print('Blastoff!')
```

Casi puede leer la declaración while como si fuera inglés. Significa, "Mientras n sea mayor que 0, muestre el valor de n y luego disminuya n. Cuando llegue a 0, muestre la palabra Blastoff!"

Más formalmente, aquí está el flujo de ejecución de una declaración `while`:

1. Determine si la condición es verdadera o falsa.

2. Si es falso, salga de la declaración `while` y continúe la ejecución en la siguiente declaración.

3. Si la condición es verdadera, ejecute el cuerpo y luego regrese al paso 1.

Este tipo de flujo se denomina bucle porque el tercer paso gira alrededor de la parte superior.

El cuerpo del bucle debe cambiar el valor de una o más variables de modo que la condición se convierta finalmente en falsa y el bucle termine. De lo contrario, el ciclo se repetirá para siempre, lo que se denomina **bucle infinito**. Una fuente interminable de diversión para los científicos informáticos es la observación de que las instrucciones del champú, "Enjabona, enjuaga, repite", son un ciclo infinito.

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

Para algunos otros bucles, no es tan fácil de decir. Por ejemplo:

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n is even
            n = n / 2
        else:                 # n is odd
            n = n*3 + 1
```

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

Cada vez que n pasa el ciclo, el programa emite el valor de 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 sequence es 3, los valores resultantes de n son 3, 10, 5, 16, 8, 4, 2, 1.

Dado que a n veces aumenta y, a veces, disminuye, no hay pruebas obvias de que n llegue a 1, o que el programa finalice. Para algunos valores particulares de n, podemos probar la terminación. Por ejemplo, si el valor inicial es una potencia de dos, n será igual cada vez que pase por el ciclo hasta que alcance 1. El ejemplo anterior finaliza con dicha secuencia, comenzando con 16.

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

Como ejercicio, vuelva a escribir la función `print_n` de "Recursividad" usando iteración en lugar de recursión.

7.4 Break

A veces no sabes que es hora de terminar un ciclo hasta que llegues a la mitad del cuerpo. En ese caso, puede usar la instrucción `break` para saltar fuera del ciclo.

Por ejemplo, supongamos que quiere tomar información del usuario hasta que escriba `done`. Podrías escribir:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)

print('Done!')
```

La condición del bucle es True, que siempre es verdadera, por lo que el bucle se ejecuta hasta que llegue a la declaración de interrupción.

Cada vez que pasa, se le solicita al usuario un paréntesis angular. Si el usuario escribe done, la instrucción `break` sale del ciclo. De lo contrario, el programa repite todo lo que el usuario escriba y vuelve al principio del ciclo. Aquí hay una muestra de ejecución:

```
> not done
not done
> done
Done!
```

Esta forma de escribir bucles `while` es común porque puede verificar la condición en cualquier punto del ciclo (no solo en la parte superior) y puede expresar la condición de stop afirmativamente ("detener cuando esto sucede") en lugar de negativamente ("seguir hasta que eso ocurra" ").

7.5 Raíces cuadradas

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

Por ejemplo, una forma de calcular las raíces cuadradas es el método de Newton. Supongamos que quiere saber la raíz cuadrada de a. Si comienza con casi cualquier estimación, x, puede 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 la 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 se necesitan para llegar a la respuesta correcta, pero sabemos cuándo llegamos 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 ciclo que comienza con una estimación inicial x, y lo 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 con `float`. Los valores de punto 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 exactamente con a `float`.

En lugar de verificar si `x` y `y` son exactamente iguales, es más seguro usar la función incorporada `abs` para calcular el valor absoluto, o la magnitud, de la diferencia entre ellos:

```
if abs(y-x) < epsilon:
    break
```

Donde `epsilon` tiene un valor, como 0.0000001, que determina qué tan cerca está lo suficientemente cerca.

7.6 Algoritmos

El método de Newton es un ejemplo de un **algoritmo**: es un proceso mecánico para resolver una categoría de problemas (en este caso, calcular raíces cuadradas).

Para entender qué es un algoritmo, podría ser útil comenzar con algo que no sea un algoritmo. Cuando aprendió a multiplicar números de un solo dígito, probablemente haya memorizado la tabla de multiplicar. En efecto, memorizaste 100 soluciones específicas. Ese tipo de conocimiento no es algorítmico.

Pero si fueras "flojo", es posible que hayas aprendido algunos trucos. Por ejemplo, para encontrar el producto de n y 9, puede escribir n-1 como primer dígito y 10-n como segundo dígito. Este truco es una solución general para multiplicar cualquier número de un solo dígito por 9. ¡Eso es un algoritmo!

De manera similar, las técnicas que aprendiste para agregar con acarreo, resta con préstamo y división larga son todos algoritmos. Una de las características de los algoritmos es que no requieren inteligencia para llevar a cabo. Son procesos mecánicos en los que cada paso se sigue de acuerdo con un simple conjunto de reglas.

Ejecutar algoritmos es aburrido, pero diseñarlos es interesante, intelectualmente desafiante y una parte central de la informática.

Algunas de las cosas que las personas hacen naturalmente, sin dificultad o pensamiento consciente, son las más difíciles de expresar algorítmicamente. Comprender el 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 comienzas a escribir programas más grandes, es posible que pases más tiempo depurando. Más código significa más posibilidades de cometer un error y más lugares para ocultar errores.

Una forma de reducir el tiempo de depuración es "depurar mediante bisección". Por ejemplo, si hay 100 líneas en su programa y las verifica una a la vez, tomaría 100 pasos.

En cambio, intente dividir el problema a la mitad. Mire en el medio del programa, o cerca de él, un valor intermedio que puede verificar. Agregue una declaración `print` (u otra cosa que tenga un efecto verificable) y ejecute el programa.

Si la comprobación del punto medio es incorrecta, debe haber un problema en la primera mitad del programa. Si es correcto, el problema está en la segunda mitad.

Cada vez que realiza un control como este, reduce a la mitad el número de líneas que debe buscar. Después de seis pasos (que son menos de 100), estaría en una o dos líneas de código, al menos en teoría.

En la práctica, no siempre está claro qué es el "medio del programa" y no siempre es posible verificarlo. No tiene sentido contar líneas y encontrar el punto medio exacto. En su lugar, piense en los lugares del programa donde podría haber errores y lugares donde es fácil verificarlo. Luego, elija un lugar donde crea que las probabilidades son las mismas que las del error antes o después del control.

7.8 Glosario

reasignación:

Asignando un nuevo valor a una variable que ya existe.

actualizar:

Una tarea en la que el nuevo valor de la variable depende de la anterior.

inicialización:

Una tarea que da un valor inicial a una variable que se actualizará.

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 declaraciones utilizando una llamada a función recursiva o un ciclo.

Bucle infinito:

Un bucle en el que la condición de terminación nunca se cumple.

algoritmo:

Un proceso general para resolver una categoría de problemas.

7.9 Ejercicios

Ejercicio 7-1.

Copia el bucle de "Raíces cuadradas" y encapsúlalo en una función llamada `mysqrt` que toma a como parámetro, elige un valor razonable de x, y devuelve una estimación de la raíz cuadrada de a.

Para probarlo, escriba una función llamada `test_square_root` que imprima una tabla como esta:

```
a mysqrt (a) math.sqrt (a) diff
- --------- ------------ ----
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 un calculado con `mysqrt`; 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'>
```

Escriba una función llamada eval_loop que induce de forma iterativa al usuario, toma la entrada resultante y la evalúa usando eval e imprime el resultado.

Debería continuar hasta que el usuario ingrese 'done', y luego devolver el valor de la última expresión que evaluó.

Ejercicio 7-3.

El matemático Srinivasa Ramanujan encontró una serie infinita que puede usarse 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 `estimate_pi` que usa esta fórmula para calcular y devolver una estimación de π. Debería usar un ciclo while para calcular los términos de la suma hasta que el último término sea más pequeño que 1e-15 (para lo cual se usa la notación de Python). Puede verificar el resultado comparándolo con math.pi.

Solución: *http://thinkpython2.com/code/pi.py*.

Capítulo 8

Strings

Los strings o cadenas no son como enteros, flotantes y booleanos. Una cadena es una **secuencia**, lo que significa que es una colección ordenada de otros valores. En este capítulo, verá cómo acceder a los caracteres que componen una cadena y obtendrá información sobre algunos de los métodos que proporcionan las cadenas.

8.1 Una cadena es una secuencia

Una cadena es una secuencia de caracteres. Puede acceder a los caracteres uno a la vez con el operador de corchetes:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

La segunda declaración selecciona el número el carácter 1 de fruit y lo asigna a letter.

La expresión entre paréntesis se llama **índice**. El índice indica qué carácter de la secuencia desea (de ahí el nombre).

Pero es posible que no obtenga lo que espera:

```
>>> letter
'a'
```

Para la mayoría de las personas, la primera letra de 'banana' es b, no a. Pero para los científicos informáticos, el índice es un desplazamiento desde el comienzo de la cadena, y el desplazamiento de la primera letra es cero.

```
>>> letter = fruit[0]
>>> letter
'b'
```

Así b es la décima letra 0 de 'banana', a es la 1ª letra, y n es la 2ª letra.

Como índice, puede usar una expresión que contenga variables y operadores:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

Pero el valor del índice tiene que ser un número entero. De lo contrario, obtienes:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

8.2 Len

len es una función incorporada que devuelve la cantidad de caracteres en una cadena:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Para obtener la última letra de una cadena, es posible que tengas la tentación de probar algo como esto:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

La razón de **IndexError** es que no hay ninguna letra en la palabra 'banana' con el índice 6. Como comenzamos a contar en cero, las seis letras están numeradas de 0 a 5. Para obtener el último carácter, debe restar 1 de length:

```
>>> last = fruit [length-1]
>>> last
'a'
```

O puede usar índices negativos, que cuentan hacia atrás desde el final de la cadena. La expresión `fruit[-1]` produce la última letra, `fruit[-2]` produce la penúltima, y así sucesivamente.

8.3 Recorrido con un bucle for

Una gran cantidad de cálculos implican el procesamiento de una cadena de un carácter a la vez. A menudo comienzan desde el principio, seleccionan a cada personaje por turno, le hacen algo y continúan hasta el final. Este patrón de procesamiento se denomina recorrido. Una forma de escribir un recorrido es con un bucle `while`:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

Este bucle atraviesa la cuerda y muestra cada letra en una línea sola. La condición de bucle es `index < len(fruit)`, por lo que cuando index es igual a la longitud de la cadena, la condición es falsa, y el cuerpo del bucle no se ejecuta. El último carácter al que se accede es el que tiene el índice `len(fruit)-1`, que es el último carácter de la cadena.

Como ejercicio, escriba una función que tome una cadena como argumento y muestre las letras hacia atrás, una por línea.

Otra forma de escribir un recorrido es con un bucle `for`:

```
for letter in fruit:
    print(letter)
```

Cada vez que pasa el ciclo, el siguiente carácter de la cadena se asigna a la variable `letter`. El ciclo continúa hasta que no queden caracteres.

El siguiente ejemplo muestra cómo usar la concatenación (adición de cadena) y un ciclo `for` para generar una serie abecedaria (es decir, en orden alfabético). En el libro de Robert McCloskey *Make Way for Ducklings*, los nombres de los patitos son Jack, Kack, Lack, Mack, Nack, Ouack, Pack y Quack. Este ciclo emite estos nombres en orden:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'


for letter in prefixes:
    print(letter + suffix)
```

El resultado es:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Por supuesto, eso no está del todo bien porque "Ouack" y "Quack" están mal escritos. Como ejercicio, modifique el programa para corregir este error.

8.4 Rebanadas de cadena

Un segmento de una cadena se llama **slice**. Seleccionar un segmento 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 del carácter "n-ésimo" al carácter "m-ésimo", incluido el primero, pero excluyendo el último. Este comportamiento es contrario a la intuición, pero podría ayudar a imaginar los índices que apuntan entre los personajes, como en la figura 8-1.

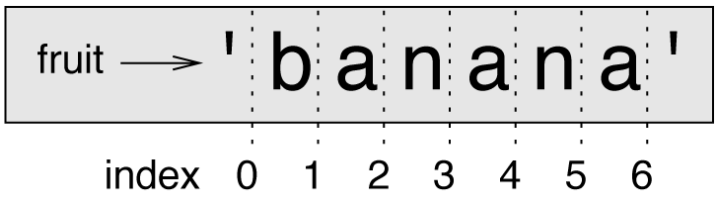

Figura 8-1. Corte de los índices.

Si omite el primer índice (antes de los dos puntos), el corte comienza al principio de la cadena. Si omite el segundo índice, el corte va al final de la cadena:

```
>>> fruit = 'banana'
>>> fruit [: 3]
'ban'
>>> fruit [3:]
'ana
```

Si el primer índice es mayor o igual que el segundo, el resultado es una cadena vacía, representada por dos comillas:

```
>>> fruit = 'banana'
>>> fruit [3: 3]
''
```

Una cadena vacía no contiene caracteres y tiene una longitud de 0, pero aparte de eso, es igual que cualquier otra cadena.

Continuando con este ejemplo, ¿qué crees que `fruit[:]` significa? Pruébalo y mira.

8.5 Las cadenas son inmutables

Es tentador usar el operador [] en el lado izquierdo de una tarea, con la intención de cambiar un carácter en una cadena. Por ejemplo:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

El "objeto" en este caso es la cadena y el "elemento" es el carácter que trataste de asignar. Por ahora, un objeto es lo mismo que un valor, pero redefiniremos esa definición más tarde ("Objetos y valores").

El motivo del error es que las cadenas son **inmutables**, lo que significa que no puede cambiar una cadena existente. Lo mejor que puedes hacer es crear una nueva cadena que sea una variación del original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

Este ejemplo concatena una nueva primera letra en una porción de `greeting`. No tiene ningún efecto en la cadena original.

8.6 Búsqueda

¿Qué hace la siguiente función?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

En cierto sentido, `find` es el inverso 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 regresa `-1`.

Este es el primer ejemplo que hemos visto de una declaración `return` dentro de un ciclo. Si `word[index] == letter`, la función sale del bucle y vuelve inmediatamente.

Si el carácter no aparece en la cadena, el programa sale del ciclo normalmente y regresa -1.

Este patrón de computación (atravesar una secuencia y regresar cuando encontramos lo que estamos buscando) se denomina **búsqueda**.

Como ejercicio, modifique `find` para que tenga un tercer parámetro: el índice en el `word` que debería comenzar a buscar.

8.7 Looping y conteo

El siguiente programa cuenta el número de veces que la letra `a` aparece en una cadena:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Este programa demuestra otro patrón de computación llamado **contador**. La variable `count` se inicializa a 0 y luego se incrementa cada vez que `a` se encuentra una. Cuando el ciclo sale, `count` contiene el resultado: el número total de `a`'s.

Como ejercicio, encapsula este código en una función llamada `count`, y generalízalo para que acepte la cadena y la letra como argumentos.

A continuación, vuelva a escribir la función para que, en lugar de atravesar la cadena, utilice la versión de tres parámetros 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 la función `upper(word)`, utiliza la sintaxis del método `word.upper()`:

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

Esta forma de notación de puntos especifica el nombre del método, `upper` y el nombre de la cadena de aplicar el método a, `word`. Los paréntesis vacíos indican que este método no toma argumentos.

Una llamada a un método se llama **invocación**; en este caso, podríamos decir que estamos invocando `upper` en `word`.

Como resultado, hay un método de cadena llamado `find` que es notablemente similar a la función que escribimos:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

En este ejemplo, se invoca `find` en `word` y pasar la letra que estamos buscando como un parámetro.

En realidad, el método `find` es más general que nuestra función; puede encontrar sub-cadenas, no solo caracteres:

```
>>> word.find ('na')
2
```

De forma predeterminada, `find` comienza al principio de la cadena, pero puede tomar un segundo argumento, el índice donde debería comenzar:

```
>>> word.find ('na', 3)
4
```

Este es un ejemplo de un argumento **opcional**. `find` también puede tomar un tercer argumento, el índice donde debería parar:

```
>>> name = 'bob'
>>> name.find ('b', 1, 2)
-1
```

Esta búsqueda falla porque bno aparece en el rango de índices de 1que 2, sin incluir 2. La búsqueda hasta, pero sin incluir, el segundo índice es findconsistente con el operador de división.

8.9 El operador

La palabra `in` es un operador booleano que toma dos cadenas y regresa `True` si la primera aparece como una sub-cadena en la segunda:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

Por ejemplo, la siguiente función imprime todas las letras de `word1` que también aparecen en `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

Con nombres de variables bien elegidos, Python a veces se lee como inglés. Puede leer este ciclo, "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 manzanas y naranjas:

```
>>> in_both ('manzanas', 'naranjas')
a
e
s
```

8.10 Comparación de cuerdas

Los operadores relacionales trabajan en cadenas. Para ver si dos cadenas son iguales:

```
if word == 'banana':
    print('All right, bananas.')
```

Otras operaciones relacionales son útiles para poner las palabras en orden alfabético:

```
if word < 'banana':
```

```
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python no maneja las letras mayúsculas y minúsculas de la misma manera que las personas. Todas las letras mayúsculas aparecen antes que las letras minúsculas, así que:

```
Your word, Pineapple, comes before banana.
```

Una forma común de resolver este problema es convertir cadenas a un formato estándar, como todas minúsculas, antes de realizar la comparación. Tenga esto en cuenta en caso de que tenga que defenderse contra un hombre armado con una piña.

8.11 Depuración

Cuando utiliza índices para recorrer los valores en una secuencia, es complicado obtener el principio y el final del recorrido correcto. Aquí hay una función que se supone que compara dos palabras y regresa `True` si una de las palabras es la inversa de la otra, pero contiene dos errores:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

La primera declaración `if` verifica si las palabras tienen la misma longitud. Si no, podemos regresar `False` de inmediato. De lo contrario, para el resto de la función, podemos suponer que las palabras tienen la misma longitud. Este es un ejemplo del patrón de guardián en "Tipos de comprobación".

`i` y `j` son índices: `i` atraviesa `word1` hacia adelante mientras `j` atraviesa `word2` hacia atrás. Si encontramos dos letras que no coinciden, podemos devolverlas `False` de inmediato. Si pasamos todo el ciclo y todas las letras coinciden, retornamos True.

Si probamos esta función con las palabras "pots" y "stop", esperamos el valor de retorno True, pero obtenemos un IndexError:

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[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)        # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

Ahora cuando ejecuto el programa nuevamente, obtengo más información:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

La primera vez a través del ciclo, el valor de `j` es 4, que está fuera del rango de la cadena `'pots'`. El índice del último carácter es 3, por lo que el valor inicial `j` debe ser `len(word2)-1`.

Si corrijo ese error y vuelvo a ejecutar el programa, obtengo:

```
>>> is_reverse ('pots', 'stop')
0 3
1 2
2 1
True
```

Esta vez obtenemos la respuesta correcta, pero parece que el ciclo solo se ejecutó tres veces, lo que es sospechoso. Para tener una mejor idea de lo que está sucediendo, es útil dibujar un diagrama de estado. Durante la primera iteración, el marco para `is_reverse` se muestra en la Figura 8-2.

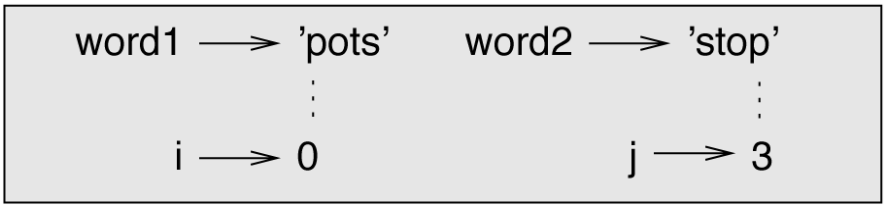

Figura 8-2. Diagrama de estado.

Tomé cierta libertad organizando las variables en el marco y agregando líneas punteadas para mostrar que los valores de `i` y `j` indican los caracteres en `word1` y `word2`.

Comenzando con este diagrama, ejecute 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 que una variable puede referirse. Por ahora, puedes usar "objeto" y "valor" de manera intercambiable.

secuencia:

Una colección ordenada de valores donde cada valor se identifica mediante un índice entero.

item:

Uno de los valores en una secuencia.

índice:

Un valor entero utilizado para seleccionar un elemento en una secuencia, como un carácter en una cadena. En Python los índices comienzan desde 0.

slice:

Una parte de una cadena especificada por un rango de índices.

cuerda vacía:

Una cadena sin caracteres y longitud 0, representada por dos comillas.

inmutable:

La propiedad de una secuencia cuyos elementos no se pueden cambiar.

atravesar:

Para recorrer los elementos en una secuencia, realizando una operación similar en cada uno.

buscar:

Un patrón de recorrido que se detiene cuando encuentra lo que está buscando.

contador:

Una variable utilizada para contar algo, generalmente inicializada a cero y luego incrementada.

invocación:

Una declaración que llama a un método.

argumento opcional:

Un argumento de función o método que no es necesario.

8.13 Ejercicios

Ejercicio 8-1.

Lea la documentación de los métodos de cadena en *http://docs.python.org/3/library/stdtypes.html#string-methods*. Es posible que desee experimentar con algunos de ellos para asegurarse de que entiende cómo funcionan. y son particularmente útiles `.stripreplace`

La documentación usa una sintaxis que puede ser confusa. Por ejemplo, en `find(sub[, start[, end]])`, los corchetes indican argumentos opcionales. Entonces `sub` es obligatorio, pero `start` es opcional, y si lo incluye start, entonces `end` es opcional.

Ejercicio 8-2.

Hay un método de cadena llamado `count` que es similar a la función en "Looping y conteo". Lea la documentación de este método y escriba una invocación que cuente la cantidad de `a`'s en 'banana'.

Ejercicio 8-3.

Un segmento de cadena puede tomar un tercer índice que especifica el "tamaño de paso"; es decir, la cantidad de espacios entre caracteres sucesivos. Un tamaño de paso de 2 significa cualquier otro carácter; 3 significa cada tercio, etc.

```
>>> fruit = 'banana'
>>> fruit [0: 5: 2]
'bnn'
```

Un tamaño de paso de -1 pasa por la palabra hacia atrás, por lo que el corte [::-1]genera una secuencia invertida.

Use este modismo para escribir una versión de una línea en `is_palindrome` del Ejercicio 6-3.

Ejercicio 8-4.

Las siguientes funciones están destinadas a verificar si una cadena contiene letras minúsculas, pero al menos algunas están equivocadas. Para cada función, describa qué hace realmente la función (suponiendo que el parámetro sea una cadena).

```
def any_lowercase1(s):
```

```
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Ejercicio 8-5.

Una cifra César es una forma débil de cifrado que implica "rotar" cada letra en un número fijo de lugares. Rotar una letra significa desplazarla por el alfabeto, volviendo al principio si es necesario, por lo que 'A' desplazada por 3 es 'D' y 'Z' desplazada por 1 es 'A'.

Para rotar una palabra, rota cada letra por la misma cantidad. Por ejemplo, " cheer" girada por 7 es " jolly" y " melon" rotado por -10 es "cubed". En la película 2001: *A Space Odyssey*, la computadora del barco se llama HAL, que es IBM girada por -1.

Escribe una función llamada `rotate_word` que toma una cadena y un número entero como parámetros, y devuelve una nueva cadena que contiene las letras de la cadena original girada por la cantidad dada.

Es posible que desee utilizar la función incorporada `ord`, que convierte un carácter en un código numérico y `chr` que convierte los códigos numéricos en caracteres. Las letras del alfabeto están codificadas en orden alfabético, por ejemplo:

```
>>> ord('c') - ord('a')
2
```

Porque `'c'` es la letra de dos-enésima del alfabeto. Pero cuidado: los códigos numéricos para letras mayúsculas son diferentes.

En ocasiones, los chistes potencialmente ofensivos en Internet están codificados en ROT13, que es un cifrado César con rotación 13. Si no se ofende fácilmente, busque y decodifique algunos de ellos.

Solución: *http://thinkpython2.com/code/rotate.py*.

Capítulo 9

Caso de estudio: Juego de palabras

Este capítulo presenta el segundo caso de estudio, que consiste en resolver acertijos de palabras buscando palabras que tienen 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: la reducción a un problema resuelto previamente.

9.1 Lectura de 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 recogidas y aportadas al dominio público por Grady Ward como parte del proyecto léxico de Moby (ver *http://wikipedia.org/wiki/Moby_Project*). Es una lista de 113,809 crucigramas oficiales; es decir, palabras que se consideran válidas en crucigramas y otros juegos de palabras. En la colección de Moby, el nombre del archivo es 113809of.ficwords.txt; puede descargar una copia, con el nombre más simple, de *http://thinkpython2.com/code/words.txt*.

Este archivo está en texto plano, por lo que puede abrirlo con un editor de texto, pero también puede leerlo desde Python. La función incorporada `open` toma el nombre del archivo como parámetro y devuelve un objeto de archivo que puede usar para leer el archivo.

```
>>> fin = open('words.txt')
```

`fin` es un nombre común para un objeto de archivo utilizado para la entrada. El objeto de archivo proporciona varios métodos para leer, incluidos readline, que lee caracteres del archivo hasta que llega a una nueva línea y devuelve el resultado como una cadena:

```
>>> fin.readline ()
'aa \ r \ n'
```

La primera palabra en esta lista en particular es "aa", que es un tipo de lava. La secuencia \r\n representa dos espacios en blanco, un retorno de carro y una nueva línea, que separan esta palabra de la siguiente.

El objeto de archivo realiza un seguimiento de dónde se encuentra en el archivo, por lo que si `readline` vuelve a llamar, obtendrá la siguiente palabra:

```
>>> fin.readline ()
'aah \ r \ n'
```

La siguiente palabra es "aah", que es una palabra perfectamente legítima, así que deja de mirarme así. O bien, si es el espacio en blanco el que lo está molestando, podemos deshacernos de él con el método de cadena `strip`:

```
>>> line = fin.readline()
```

```
>>> word = line.strip()
>>> word
'aahed'
```

También puede usar 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 line in fin:
    word = line.strip()
    print(word)
```

9.2 Ejercicios

Hay soluciones para estos ejercicios en la siguiente sección. Al menos debe intentar cada uno antes de leer las soluciones.

Ejercicio 9-1.

Escriba 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* que no contiene la letra "e". Como "e" es la letra más común en inglés, no es fácil de hacer.

De hecho, es difícil construir un pensamiento solitario sin usar el símbolo más común. Al principio es lento, pero con precaución y horas de entrenamiento puede ganar facilidades gradualmente.

De acuerdo, me detendré ahora.

Escriba una función llamada `has_no_e` que retorna `True` si la palabra dada no tiene la letra "e" en ella.

Modifique su programa de la sección anterior para imprimir solo las palabras que no tienen "e" y calcule el porcentaje de las palabras en la lista que no tienen "e".

Ejercicio 9-3.

Escribe una función llamada `avoids` que tome una palabra y una cadena de letras prohibidas, y eso retorna `True` si la palabra no usa ninguna de las letras prohibidas.

Modifique su programa para solicitar al usuario que ingrese una cadena de letras prohibidas y luego imprima el número de palabras que no contienen ninguna de ellas. ¿Puedes encontrar una combinación de cinco letras prohibidas que excluya el menor número de palabras?

Ejercicio 9-4.

Escriba una función llamada `uses_only` que tome una palabra y una cadena de letras, y que retorna `True` si la palabra contiene solo letras en la lista. ¿Puedes hacer una oración usando solo las letras `acefhlo`? ¿Aparte de "azada alfalfa"?

Ejercicio 9-5.

Escriba una función llamada `uses_all` que tome una palabra y una cadena de letras requeridas, y eso retorna `True` si la palabra usa todas las letras requeridas al menos una vez. ¿Cuántas palabras hay que usan todas las vocales `aeiou`? ¿Qué tal `aeiouy`?

Ejercicio 9-6.

Escriba una función llamada `is_abecedarian` que retorna `True` si las letras en una palabra aparecen en orden alfabético (las letras dobles son correctas). ¿Cuántas palabras alfabéticas hay?

9.3 Buscar

Todos los ejercicios en la sección previa tienen algo en común; se pueden resolver con el patrón de búsqueda que vimos en "Búsqueda". El ejemplo más simple es:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

El bucle `for` recorre los caracteres de `word`. Si encontramos la letra "e", podemos regresar inmediatamente `False`; de lo contrario, tenemos que pasar a la siguiente letra. Si salimos del circuito normalmente, eso significa que no encontramos una "e", así que volvemos True.

Podría escribir esta función de forma más concisa utilizando el operador in, pero comencé con esta versión porque demuestra la lógica del patrón de búsqueda.

`Avoids` es una versión más general de `has_no_e` pero tiene la misma estructura:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Podemos regresar `False` tan pronto como encontremos una letra prohibida; si llegamos al final del ciclo, regresamos `True`.

`uses_only` es similar, excepto que el sentido de la condición se invierte:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

En lugar de una lista de letras prohibidas, tenemos una lista de letras disponibles. Si encontramos una letra en `word` que no está `available`, podemos regresar `False`.

`uses_all` es similar, excepto que invertimos el rol de la palabra en la cadena de letras:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

En lugar de atravesar las letras en `word`, el bucle atraviesa las letras requeridas. Si alguna de las letras requeridas no aparece en la palabra, podemos regresar `False`.

Si estuvieras realmente pensando como un científico de la computación, habrías reconocido que `uses_all` era una instancia de un problema previamente resuelto, y hubieras escrito:

```
def uses_all(word, required):
    return uses_only(required, word)
```

Este es un ejemplo de un plan de desarrollo de programa llamado **reducción a un problema resuelto previamente**, lo que significa que reconoce el problema en el que está trabajando como una instancia de un problema resuelto y aplica una solución existente.

9.4 Looping con Índices

Escribí las funciones en la sección anterior con bucles `for` porque solo necesitaba los caracteres en las cuerdas; No tuve que hacer nada con los índices.

Para `is_abecedarian` que tengamos que comparar letras adyacentes, que es un poco complicado con un bucle `for`:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

Una alternativa es usar recursión:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
```

```
        return False
    return is_abecedarian(word[1:])
```

Otra opción es usar un bucle `while`:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

El ciclo comienza en `i=0` y termina cuando `i=len(word)-1`. Cada vez que pasa el ciclo, compara el carácter i-ésimo (que se puede considerar como el personaje actual) con el carácter i + 1 (que se puede ver como el siguiente).

Si el siguiente carácter es menor que (alfabéticamente antes) el actual, entonces hemos descubierto una ruptura en la tendencia abecedaria, y retornamos `False`.

Si llegamos al final del ciclo sin encontrar una falla, entonces la palabra pasa la prueba. Para convencerse de que el ciclo finaliza correctamente, considere un ejemplo como `'flossy'`. La longitud de la palabra es 6, por lo que la última vez que se ejecuta el ciclo 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 `is_palindrome` (ver Ejercicio 6-3) que usa dos índices: uno comienza al principio y sube; el otro comienza al final y baja.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1


    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1


    return True
```

O podríamos reducir a un problema resuelto previamente y escribir:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Usando `is_reverse` de la Figura 8-2.

9.5 Depuración

Probar programas es difícil. Las funciones de este capítulo son relativamente fáciles de probar porque puede verificar los resultados a mano. Aun así, es difícil o imposible elegir un conjunto de palabras que prueben todos los posibles errores.

Tomando `has_no_e` como ejemplo, hay dos casos obvios para verificar: las palabras que tienen una 'e' deben regresar `False`, y las palabras que no deben regresar `True`. No deberías tener problemas para encontrar uno de cada uno.

Dentro de cada caso, hay algunos subcampos menos obvios. Entre las palabras que tienen una "e", debe probar las palabras con una "e" al principio, al final y en algún lugar en el medio. Debe probar palabras largas, cortas y 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 a menudo acechan los errores.

Además de los casos de prueba que genera, también puede probar su programa con una lista de palabras como `words.txt`. Al escanear el resultado, es posible que pueda detectar errores, pero tenga cuidado: podría detectar un tipo de error (palabras que no deberían incluirse, pero sí lo son) y no otro (palabras que deberían incluirse, pero no lo son).

En general, las pruebas pueden ayudarlo a encontrar errores, pero no es fácil generar un buen conjunto de casos de prueba, e incluso si lo hace, no puede estar seguro de que su programa sea correcto. De acuerdo con un legendario informático:

> *Las pruebas del programa se pueden usar 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 resuelto previamente:

Una forma de resolver un problema expresándolo como una instancia de un problema resuelto previamente.

caso especial:

Un caso de prueba que es atípico o no obvio (y es menos probable que se maneje correctamente).

Ejercicios

Ejercicio 9-7.

Esta pregunta se basa en un Puzzler que se emitió en el programa de radio Car Talk (*http://www.cartalk.com/content/puzzlers*):

Dame una palabra con tres letras dobles consecutivas. Le daré un par de palabras que casi califican, pero no las tengo. Por ejemplo, la palabra committee, c-o-m-m-i-t-t-e-e. Sería genial, excepto 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's, funcionaría. Pero hay una palabra que tiene tres pares consecutivos de letras y, a lo mejor de mi conocimiento, esta puede ser la única palabra. Por supuesto, probablemente haya 500 más, pero solo puedo pensar en uno. ¿Cuál es la palabra?

Escribe un programa para encontrarlo.

Solución: *http://thinkpython2.com/code/cartalk1.py*.

Ejercicio 9-8.

Aquí hay otro rompecabezas de Car Talk (*http://www.cartalk.com/content/puzzlers*):

"Estaba conduciendo en la carretera el otro día y noté mi odómetro por casualidad. Como la mayoría de los odómetros, muestra seis dígitos, solo en millas. Entonces, si mi auto tuviera 300,000 millas, por ejemplo, vería 3-0-0-0-0-0.

"Ahora, lo que vi ese día fue muy interesante. Noté que los últimos 4 dígitos eran palindrómicos; es decir, ellos leen lo mismo hacia adelante que 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 tarde, los últimos 5 números fueron palindrómicos. Por ejemplo, podría haber leído 3-6-5-4-5-6. Una milla después de eso, el centro 4 de 6 números eran palindrómicos. ¿Y estás listo para esto? Una milla más tarde, ¡los 6 fueron palindrómicos!

Escriba un programa de Python que pruebe todos los números de seis dígitos e imprima cualquier número que satisfaga estos requisitos.

Solución: *http://thinkpython2.com/code/cartalk2.py*.

Ejercicio 9-9.

Aquí hay otro rompecabezas de Car Talk que puedes resolver con una búsqueda (*http://www.cartalk.com/content/puzzlers*):

"Recientemente tuve una visita con mi madre y nos dimos cuenta de que los dos dígitos que componen mi edad cuando se revirtieron dieron como resultado su edad. Por ejemplo, si tiene 73 años, yo tengo 37. Nos preguntamos con qué frecuencia esto ha sucedido a lo largo de los años, pero nos desviamos de otros temas y nunca obtuvimos una respuesta.

"Cuando llegué a casa me di cuenta de que los dígitos de nuestra edad han sido reversibles seis veces hasta el momento. También me di cuenta de que si tenemos suerte volvería a suceder en unos años, y si tenemos suerte, pasaría una vez más después de eso. En otras palabras, hubiera sucedido 8 veces sobre todo. Entonces la pregunta es, ¿cuántos años tengo ahora?

Escriba un programa de Python que busque soluciones para este rompecabezas. Sugerencia: es posible que encuentre 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: listas. También aprenderá más sobre los objetos y lo que puede suceder si tiene más de un nombre para el mismo objeto.

10.1 Una lista es una secuencia

Como 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 **ítems** o, a veces, **elementos**.

Hay varias formas de crear una nueva lista; lo más simple es incluir los elementos entre corchetes ([ ]):

```
[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 flotante, un entero y (¡menos!) 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; puede crear uno con corchetes vacíos, [].

Como era de esperar, puede asignar valores de lista a las variables:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

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 entre corchetes especifica el índice. Recuerde que los índices comienzan en 0:

```
>>> cheeses[0]
'Cheddar'
```

A diferencia de las cadenas, las listas son mutables. Cuando el operador de corchetes aparece en el lado izquierdo de una tarea, identifica el elemento de la lista que se asignará:

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

El elemento 1-enesimo de `numbers`, que solía ser 123, ahora es 5.

La Figura 10-1 muestra el diagrama de estado para `cheeses`, `numbers` y `empty`.

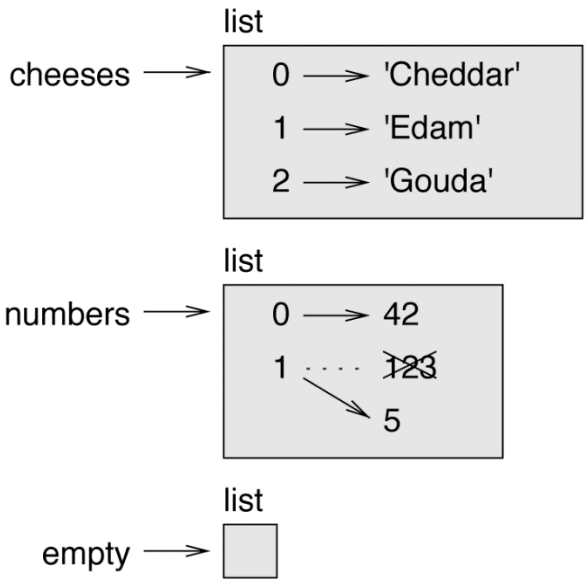

Figura 10-1. Diagrama de estado.

Las listas están representadas por cuadros con la palabra "lista" afuera y los elementos de la lista dentro. `cheeses` se refiere a una lista con tres elementos indexados 0, 1 y 2. `numbers` contiene dos elementos; el diagrama muestra que el valor del segundo elemento ha sido reasignado de 123 a 5. `empty` se refiere a una lista sin elementos.

Los índices de lista funcionan del mismo modo que los índices de cadena:

- Cualquier expresión entera se puede usar como un índice.
- Si intenta leer o escribir un elemento que no existe, obtiene un `IndexError`.
- Si un índice tiene un valor negativo, cuenta hacia atrás desde el final de la lista.

El operador `in` también trabaja en listas:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
```

```
True
>>> 'Brie' in cheeses
False
```

10.3 Atravesando una lista

La forma más común de recorrer los elementos de una lista es con un bucle `for`. La sintaxis es la misma que para las cadenas:

```
for cheese in cheeses:
    print(cheese)
```

Esto funciona bien si solo necesita leer los elementos de la lista. Pero si desea escribir o actualizar los elementos, necesita los índices. Una forma común de hacerlo es combinar las funciones incorporadas `range` y `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Este bucle atraviesa la lista y actualiza cada elemento. `len` devuelve la cantidad de elementos en la lista. `range` devuelve una lista de índices de 0 a `n-1`, donde `n` es la longitud de la lista. Cada vez que `i` pasa el ciclo, obtiene el índice del siguiente elemento. La declaración de asignación en el cuerpo utiliza `i` para leer el valor anterior del elemento y para asignar el nuevo valor.

Un bucle `for` sobre una lista vacía nunca ejecuta el cuerpo:

```
for x in []:
    print('This never happens.')
```

Aunque una lista puede contener otra lista, la lista anidada todavía cuenta como un elemento único. 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 determinado de veces:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

El primer ejemplo se repite [0] cuatro veces. El segundo ejemplo repite la lista [1, 2, 3] tres veces.

10.5 Porciones de listas

El operador de sector también trabaja en 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 omite el primer índice, el corte comienza al principio. Si omite el segundo, la porción llega al final. Entonces, si omites ambos, la porción es una copia de toda la lista:

```
>>> t [:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Como las listas son mutables, a menudo es útil hacer una copia antes de realizar operaciones que modifiquen listas.

Un operador de porción en el lado izquierdo de una tarea puede actualizar varios 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 una lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append ('d')
>>> t
['a', 'b', 'c', 'd']
```

`extend` toma una lista como argumento y agrega todos los elementos:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
```

```
>>> t1.extend (t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

Este ejemplo t2no se modifica.

`sort` organiza los elementos 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; ellos modifican la lista y regresan `None`. Si escribe accidentalmente `t=t.sort(),` se sentirá decepcionado con el resultado.

10.7 Mapa, filtro y reducir

Para sumar todos los números en una lista, puede usar un ciclo como este:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` se inicializa a 0. Cada vez que `x` pasa el ciclo, obtiene un elemento de la lista. El operador `+=` proporciona una forma breve de actualizar una variable. Esta declaración 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 utilizada de esta manera a veces se denomina **acumulador**.

Sumar los elementos de una lista es una operación tan común que Python lo proporciona como una función incorporada sum:

```
>>> t = [1, 2, 3]
>>> suma (t)
6
```

Una operación como esta que combina una secuencia de elementos en un solo valor a veces se llama **reductor**.

Algunas veces quiere atravesar una lista mientras construye otra. Por ejemplo, la siguiente función toma una lista de cadenas y devuelve una nueva lista que contiene cadenas en mayúscula:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` se inicializa con una lista vacía; cada vez que pasa el ciclo, agregamos el siguiente elemento. Entonces en `res` hay otro tipo de acumulador.

Una operación como, `capitalize_all` a veces, se denomina mapa porque "mapea" una función (en este caso, el método `capitalize`) en cada uno de los elementos en una secuencia.

Otra operación común es seleccionar algunos de los elementos de una lista y devolver una sub-lista. Por ejemplo, la siguiente función toma una lista de cadenas y devuelve una lista que contiene solo las cadenas en mayúsculas:

```
def only_upper(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 contiene solo letras mayúsculas.

Una operación como `only_upper` se llama filtro porque selecciona algunos de los elementos y filtra los otros.

Las operaciones de lista más comunes pueden expresarse como una combinación de mapa, filtro y reducción.

10.8 Eliminar elementos

Hay varias formas de eliminar elementos de una lista. Si conoce el índice del elemento que desea, puede 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 proporciona un índice, elimina y devuelve el último elemento.

Si no necesita el valor eliminado, puede usar el operador `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t [1]
>>> t
['a', 'c']
```

Si conoce el elemento que desea eliminar (pero no el índice), puede 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, puede usar `del` con un índice de sector:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t [1: 5]
>>> t
['a', ' F']
```

Como de costumbre, el corte selecciona todos los elementos hasta, pero sin incluir, el segundo índice.

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 de una cadena a una lista de caracteres, puede usar `list`:

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

Debido a que `list` es el nombre de una función incorporada, debe evitar usarlo como un nombre de variable. También evito `l` porque se parece demasiado `1`. Entonces es por eso que uso `t`.

La función `list` divide una cadena en letras individuales. Si quiere dividir una cadena en palabras, puede 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 límites de palabras. El siguiente ejemplo usa un guion como delimitador:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> 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 debe invocarlo en el delimitador y pasar la lista como parámetro:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.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, puede usar la cadena vacía, `''`, como un delimitador.

10.10 Objetos y valores

Si ejecutamos estas declaraciones de asignación:

```
a = 'banana'
b = 'banana'
```

Sabemos que `a` y `b` nos referimos a una cadena, pero no sabemos si se refieren a la misma cadena. Hay dos estados posibles, que se muestran en la Figura 10-2.

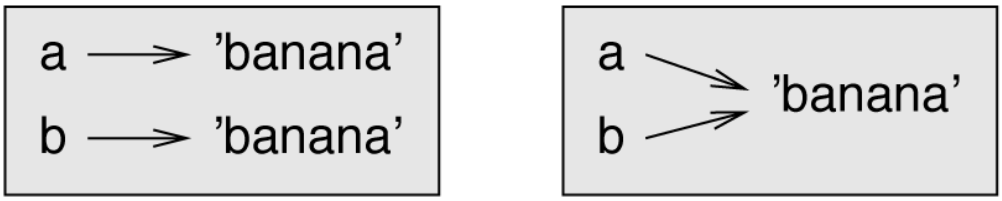

Figura 10-2. Diagrama de estado.

En un 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, puede usar el operador `is`:

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
```

```
True
```

En este ejemplo, Python sólo creó un objeto de cadena, y ambos `a` y `b` se refieren a él. 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.

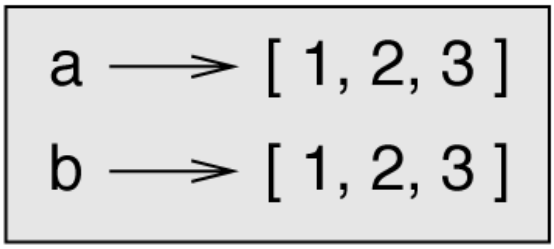

Figura 10-3. Diagrama de estado.

En este caso, diríamos que las dos listas son **equivalentes**, porque tienen los mismos elementos, pero no son **idénticos**, porque no son el mismo objeto. Si dos objetos son idénticos, también son equivalentes, pero si son equivalentes, no son necesariamente idénticos.

Hasta ahora, hemos usado "objeto" y "valor" de forma intercambiable, pero es más preciso decir que un objeto tiene un valor. Si evalúa [1, 2, 3], obtiene un objeto de lista cuyo valor es una secuencia de números enteros. Si otra lista tiene los mismos elementos, decimos que tiene el mismo valor, pero no es el mismo objeto.

10.11 Aliasing

Si `a` se refiere a un objeto y lo asigna `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 parece a la Figura 10-4.

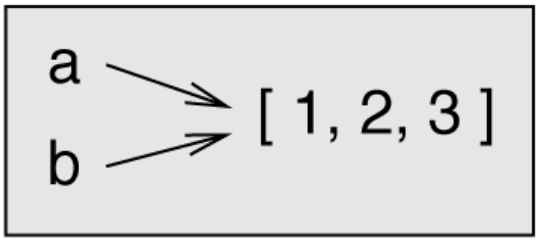

Figura 10-4. Diagrama de estado.

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 un **alias**.

Si el objeto con alias es mutable, los cambios realizados con un alias afectan al otro:

```
>>> b [0] = 42
>>> a
[42, 2, 3]
```

Aunque este comportamiento puede ser útil, es propenso a errores. En general, es más seguro evitar el **aliasing** cuando se trabaja con objetos mutables.

Para objetos inmutables como cadenas, el alias no es un gran problema. En este ejemplo:

```
a = 'banana'
b = 'banana'
```

Casi nunca hace una diferencia si `a` y se `b` refieren a la misma cadena o no.

10.12 Lista de argumentos

Cuando pasa una lista a una función, la función obtiene una referencia a la lista. Si la función modifica la lista, la persona que llama ve el cambio. Por ejemplo, `delete_head` elimina el primer elemento de una lista:

```
def delete_head (t):
    del t [0]
```

Así es como se usa:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

El parámetro `t` y la variable `letters` son alias para el mismo objeto. El diagrama de pila se parece a la Figura 10-5.

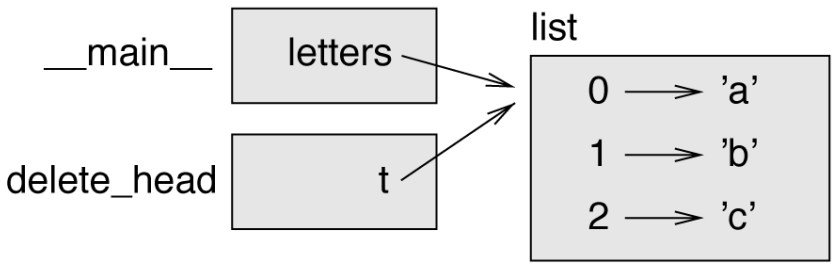

Figura 10-5. Diagrama de pila.

Como la lista está compartida por dos marcos, la dibujé entre ellos.

Es importante distinguir entre las operaciones que modifican las listas y las operaciones que crean nuevas listas. Por ejemplo, el método `append` modifica una lista, pero el `+` operador crea una nueva lista:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

`append` modifica la lista y devuelve `None`:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t1
```

El operador + crea una nueva lista y deja la lista original sin cambios.

Esta diferencia es importante cuando escribe funciones que se supone que modifican listas. Por ejemplo, esta función no elimina el encabezado de una lista:

```
def bad_delete_head(t):
    t = t[1:]              # WRONG!
```

El operador de división crea una lista nueva y la asignación hace que se refiere a ella, pero eso no afecta a la persona que llama.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head (t4)
>>> t4
[1, 2, 3]
```

Al comienzo de `bad_delete_head`, `t` y `t4` se refieren a la misma lista. Al final, se `t` refiere a una nueva lista, pero `t4` todavía se refiere a la lista original no modificada.

Una alternativa es escribir una función que crea y devuelve una nueva lista. Por ejemplo, `tail` devuelve todo menos el primer elemento de una lista:

```
def tail(t):
    return t[1:]
```

Esta función deja la lista original sin modificaciones. Así es como se usa:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

10.13 Depuración

El uso descuidado de listas (y otros objetos mutables) puede llevar a largas horas de depuración. Aquí hay algunos errores comunes y formas de evitarlos:

1. La mayoría de los métodos de lista modifican el argumento y regresan None. Esto es lo contrario de los métodos de cadena, que devuelven una nueva cadena y dejan el original solo.

   Si está acostumbrado a escribir código de cadena como este:

   ```
   word = word.strip ()
   ```

   Es tentador escribir un código de lista como este:

   ```
   t = t.sort () # ¡INCORRECTO!
   ```

   Debido a que las `sort` devuelve `None`, `t` es probable que la siguiente operación con la que realice falle.

   Antes de utilizar los métodos y operadores de lista, debe leer la documentación cuidadosamente y luego probarla en modo interactivo.

2. Elija un idioma y quédese con él.

   Parte del problema con las listas es que hay demasiadas formas de hacer las cosas. Por ejemplo, para eliminar un elemento de una lista, puede utilizar `pop`, `remove`, `del`, o incluso una rebanada de asignación.

   Para agregar un elemento, puede usar el método `append` o el operador `+`. Suponiendo que `t` es una lista y `x` es un elemento de lista, estos son correctos:

   ```
   t.append (x)
   t = t + [x]
   t + = [x]
   ```

   Y estos son incorrectos:

   ```
   t.append([x])          # WRONG!
   t = t.append(x)        # WRONG!
   t + [x]                # WRONG!
   t = t + x              # WRONG!
   ```

   Pruebe cada uno de estos ejemplos en modo interactivo para asegurarse de que entiende lo que hacen. Tenga en cuenta que solo el último causa un error de tiempo de ejecución; los otros tres son legales, pero hacen lo incorrecto.

3. Haga copias para evitar el aliasing.

   Si desea utilizar un método como `sort` que modifica el argumento, pero también necesita conservar la lista original, puede hacer una copia:

```
>>> t = [3, 1, 2]
>>> t2 = t [:]
>>> t2.sort ()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3 ]
```

   En este ejemplo, también puede usar la función incorporada `sorted`, que devuelve una nueva lista ordenada y deja solo el 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 llamado ítems.

lista anidada:

Una lista que es un elemento de otra lista.

acumulador:

Una variable utilizada en un ciclo para sumar o acumular un resultado.

asignación aumentada:

Una declaración que actualiza el valor de una variable usando un operador como **+=**.

reductor:

Un patrón de procesamiento que atraviesa una secuencia y acumula los elementos en un solo resultado.

mapeo:

Un patrón de procesamiento que atraviesa una secuencia y realiza una operación en cada elemento.

filtrado:

Un patrón de procesamiento que atraviesa una lista y selecciona los elementos que satisfacen algún criterio.

objeto:

Algo a lo que una variable puede referirse. Un objeto tiene un tipo y un valor.

equivalente:

Tienen el mismo valor

idéntico:

Siendo el mismo objeto (lo que implica equivalencia).

referencia:

La asociación entre una variable y su valor.

aliasing:

Una circunstancia donde dos o más variables se refieren al mismo objeto.

delimitador:

Un carácter o cadena utilizada para indicar dónde se debe dividir una cadena.

10.15 Ejercicios

Puede descargar soluciones para estos ejercicios desde *http://thinkpython2.com/code/list_exercises.py*.

Ejercicio 10-1.

Escriba una función llamada `nested_sum` que toma una lista de listas de enteros y suma los elementos de todas las listas anidadas. Por ejemplo:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum (t)
21
```

Ejercicio 10-2.

Escribe una función llamada `cumsum` que toma una lista de números y devuelve la suma acumulativa; es decir, una nueva lista donde el elemento i-ésimo es la suma de los primeros elementos `i + 1` de la lista original. Por ejemplo:

```
>>> t = [1, 2, 3]
>>> cumsum (t)
```

```
[1, 3, 6]
```

Ejercicio 10-3.

Escriba una función llamada `middle` que toma una lista y devuelve una nueva lista que contiene todos menos el primer y último elemento. Por ejemplo:

```
>>> t = [1, 2, 3, 4]
>>> medio (t)
[2, 3]
```

Ejercicio 10-4.

Escriba una función llamada `chop` que toma una lista, la modifica quitando los primeros y últimos elementos y devuelva `None`. Por ejemplo:

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Ejercicio 10-5.

Escriba una función llamada `is_sorted` que toma una lista como parámetro y regresa `True` si la lista está ordenada en orden ascendente y de lo contrario `False`. Por ejemplo:

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Ejercicio 10-6.

Dos palabras son anagramas si puedes reordenar las letras de una para deletrear otra. Escribe una función llamada `is_anagram` que toma dos cadenas y regresa `True` si son anagramas.

Ejercicio 10-7.

Escribe una función llamada `has_duplicates` que toma una lista y devuelve `True` si hay algún elemento que aparezca más de una vez. No debería modificar la lista original.

Ejercicio 10-8.

Este ejercicio se refiere a la llamada Paradoja de cumpleaños, sobre la que puedes leer en *http://en.wikipedia.org/wiki/Birthday_paradox*.

Si hay 23 estudiantes en su clase, ¿cuáles son las probabilidades de que dos de ustedes tengan el mismo cumpleaños? Puede estimar esta probabilidad generando muestras aleatorias de 23 cumpleaños y buscando coincidencias. Sugerencia: puede generar cumpleaños aleatorios con la función `randint` en el módulo `random`.

Puede descargar mi solución desde *http://thinkpython2.com/code/birthday.py*.

Ejercicio 10-9.

Escriba una función que lea el archivo `words.txt` y cree una lista con un elemento por palabra. Escribe dos versiones de esta función, una usando el método `append` y la otra usando la expresión idiomática `t = t + [x]`. ¿Cuál tarda más en ejecutarse? ¿Por qué?

Solución: *http://thinkpython2.com/code/wordlist.py*.

Ejercicio 10-10.

Para verificar si una palabra está en la lista de palabras, podría usar el operador `in`, pero sería lenta porque busca las palabras en orden.

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 hace cuando busca una palabra en el diccionario. Comienzas en el centro y compruebas si la palabra que estás buscando aparece antes que la palabra en el medio de la lista. Si es así, busca la primera mitad de la lista de la misma manera. De lo contrario, busca la segunda mitad.

De cualquier forma, cortas el resto del espacio de búsqueda a la mitad. Si la lista de palabras tiene 113,809 palabras, tomará alrededor de 17 pasos para encontrar la palabra o concluir que no está allí.

Escribe una función llamada `in_bisect` que toma una lista ordenada y un valor objetivo y devuelve el índice del valor en la lista, si está allí, o `None` si no.

¡O podría leer la documentación del módulo `bisect` y usar eso!

Solución: *http://thinkpython2.com/code/inlist.py*.

Ejercicio 10-11.

Dos palabras son un "par inverso" si cada una es la inversa de la otra. Escriba un programa que encuentre todos los pares inversos en la lista de palabras.

Solución: *http://thinkpython2.com/code/reverse_pair.py*.

Ejercicio 10-12.

Dos palabras, si toma letras alternas de cada una forma una nueva palabra. Por ejemplo, "shoe" y " cold " para formar "schooled".

Solución: *http://thinkpython2.com/code/interlock.py*. Crédito: Este ejercicio está inspirado en un ejemplo en *http://puzzlers.org*.

1. Escribe un programa que encuentre todos los pares de palabras que se entrelazan. Sugerencia: ¡no enumere todos los pares!

2. ¿Puedes encontrar palabras entrelazadas en tres direcciones? es decir, cada tercera letra forma una palabra, comenzando desde la primera, segunda o tercera?

Capítulo 11

Diccionarios

Este capítulo presenta otro tipo incorporado llamado diccionario. Los diccionarios son una de las mejores características de Python; son los componentes básicos de muchos algoritmos eficientes y elegantes.

11.1 Un diccionario es una asignación

Un diccionario es como una lista, pero más general. En una lista, los índices tienen que ser enteros; en un diccionario pueden ser (casi) cualquier tipo.

Un diccionario contiene una colección de índices, que se llaman **claves**, y una colección de valores. Cada clave está asociada con un solo valor. La asociación de una clave y un valor se denomina **par clave-valor** o, a veces, un **elemento**.

En lenguaje matemático, un diccionario representa un **mapeo** de claves a valores, por lo que también puede decir que cada tecla "se correlaciona" con un valor. Como ejemplo, construiremos un diccionario que correlacione palabras de inglés a español, por lo que las claves y los valores son todas las cadenas.

La función `dict` crea un nuevo diccionario sin elementos. Debido a que dictes el nombre de una función incorporada, debe evitar usarlo como un nombre de variable.

```
>>> eng2sp = dict ()
>>> eng2sp
{}
```

Los corchetes ondulados {} representan un diccionario vacío. Para agregar elementos al diccionario, puede usar corchetes:

```
>>> eng2sp['one'] = 'uno'
```

Esta línea crea un elemento que se correlaciona desde la clave `'one'` hasta el valor `'uno'`. Si imprimimos el diccionario nuevamente, vemos un par clave-valor con dos puntos entre la clave y el valor:

```
>>> eng2sp
{'one': 'uno'}
```

Este formato de salida también es un formato de entrada. Por ejemplo, puede crear un nuevo diccionario con tres elementos:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Pero si imprime `eng2sp`, se sorprenderá:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

El orden de los pares clave-valor puede no ser el mismo. Si escribe el mismo ejemplo en su computadora, puede obtener un resultado diferente. En general, el orden de los elementos 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, usa las claves para buscar los valores correspondientes:

```
>>> eng2sp ['two']
'dos'
```

La clave `'two'` siempre se asigna al valor `'dos'` para que el orden de los elementos no importe.

Si la clave no está en el diccionario, obtienes una excepción:

```
>>> eng2sp['four']
KeyError: 'four'
```

La función `len` funciona en diccionarios; devuelve el número de pares clave-valor:

```
>>> len (eng2sp)
3
```

El operador `in` también trabaja en diccionarios; le dice si algo aparece como una clave en el diccionario (aparecer como un valor no es lo suficientemente bueno).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Para ver si algo aparece como un valor en un diccionario, puede usar el método `values`, que devuelve una colección de valores, y luego usar el operador `in`:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

El operador `in` usa diferentes algoritmos para listas y diccionarios. Para listas, busca los elementos de la lista en orden, como en "Busqueda". A medida que la lista se alarga, el tiempo de búsqueda se alarga en proporción directa.

Para los diccionarios, Python usa un algoritmo llamado **hashtable** que tiene una propiedad notable: el operador `in` tarda aproximadamente la misma cantidad de tiempo sin importar cuántos elementos haya en el diccionario. Explico cómo es posible en "Hashtables", pero la explicación puede no tener sentido hasta que hayas leído algunos capítulos más.

11.2 Diccionario como una colección de contadores

Supongamos que le dan una cadena y quiere contar cuántas veces aparece cada letra. Hay varias formas en que puede hacerlo:

1. Puede crear 26 variables, una para cada letra del alfabeto. Luego puede recorrer la cadena y, para cada carácter, incrementar el contador correspondiente, probablemente utilizando un condicional encadenado.

2. Podrías crear una lista con 26 elementos. Luego puede convertir cada carácter en un número (usando la función incorporada `ord`), usar el número como índice en la lista e incrementar el contador apropiado.

3. Puede crear un diccionario con caracteres como claves y contadores como los valores correspondientes. La primera vez que vea un personaje, agregará un elemento al diccionario. Después de eso, aumentaría el valor de un elemento existente.

Cada una de estas opciones realiza el mismo cálculo, pero cada una de ellas implementa ese cálculo de una manera diferente.

Una **implementación** es una forma de realizar un cálculo; algunas implementaciones son mejores que otras. Por ejemplo, una ventaja de la implementación del diccionario es que no tenemos que saber con anticipación qué letras aparecen en la cadena y solo tenemos que dejar espacio para las letras que sí aparecen.

Aquí es cómo se vería el código:

```
def histogram(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 `histogram`, que es un término estadístico para una colección de contadores (o frecuencias).

La primera línea de la función crea un diccionario vacío. El bucle `for` atraviesa la cadena. Cada vez que c pasa el ciclo, si el personaje no está en el diccionario, creamos un nuevo elemento con clave `c` y el valor inicial 1 (ya que hemos visto esta letra una vez). Si `c` ya está en el diccionario aumentamos `d[c]`.

Así es como funciona:

```
>>> h = histogram('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 predeterminado. Si la clave aparece en el diccionario, `get` devuelve el valor correspondiente; de lo contrario, devuelve el valor predeterminado. Por ejemplo:

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Como ejercicio, use `get` para escribir de manera más concisa `histogram`. Deberías poder eliminar la declaración `if`.

11.3 Looping y diccionarios

Si usa un diccionario en una instrucción `for`, atraviesa las claves del diccionario. Por ejemplo, `print_hist` imprime cada clave y el valor correspondiente:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Así es como se ve el resultado:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Nuevamente, las llaves no están en un orden particular. Para recorrer las claves en orden, puede usar la función incorporada `sorted`:

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

11.4 Búsqueda inversa

Dado un diccionario `d` y una clave `k`, es fácil encontrar el valor correspondiente `v = d[k]`. Esta operación se llama **búsqueda**.

Pero, ¿y si tienes `v` y quieres encontrar `k`? Tienes dos problemas: primero, puede haber más de una clave que se corresponda con el valor `v`. Dependiendo de la aplicación, es posible que pueda elegir una, o puede que tenga que hacer una lista que contenga todas. Segundo, no hay una sintaxis simple para hacer una **búsqueda inversa**; tienes que buscar.

Aquí hay una función que toma un valor y devuelve la primera clave que se asigna a ese valor:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

Esta función es otro ejemplo del patrón de búsqueda, pero utiliza una característica que no hemos visto antes: `raise`. La declaración de aumento provoca una excepción; en este caso causa un `LookupError`, que es una excepción incorporada utilizada para indicar que una operación de búsqueda falló.

Si llegamos al final del ciclo, eso significa `v` que no aparece en el diccionario como un valor, por lo que planteamos una excepción.

Aquí hay un ejemplo de una búsqueda inversa exitosa:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> k
'r'
```

Y una infructuosa:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

El efecto cuando se genera una excepción es el mismo que cuando Python levanta uno: imprime un rastreo y un mensaje de error.

La instrucción `raise` puede tomar un mensaje de error detallado como un argumento opcional. Por ejemplo:

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

Una búsqueda inversa es mucho más lenta que una búsqueda directa; si tiene que hacerlo a menudo, o si el diccionario se agranda, el rendimiento de su programa sufrirá.

11.5 Diccionarios y listas

Las listas pueden aparecer como valores en un diccionario. Por ejemplo, si le dan un diccionario que mapea de letras a frecuencias, puede querer invertirlo; es decir, crea un diccionario que mapea desde frecuencias a letras. Como puede haber varias letras con la misma frecuencia, cada valor en el diccionario invertido debe ser una lista de letras.

Aquí hay una función que invierte un diccionario:

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

Cada vez que `key` pasa por el ciclo, obtiene una clave `d` y `val` obtiene el valor correspondiente. Si `val` no está, eso significa que no lo hemos visto antes, entonces creamos un nuevo ítem y lo inicializamos con un singleton (una lista que contiene un solo elemento). De lo contrario, hemos visto este valor antes, por lo que agregamos la clave correspondiente a la lista.

Aquí hay un ejemplo:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

La Figura 11-1 es un diagrama de estado que muestra `hist` y `inverse`. Un diccionario se representa como un cuadro con el tipo de dict arriba y los pares clave-valor dentro. Si los valores son enteros, flotantes o cadenas, los dibujaré dentro del cuadro, pero normalmente hago listas fuera del cuadro, solo para mantener el diagrama simple.

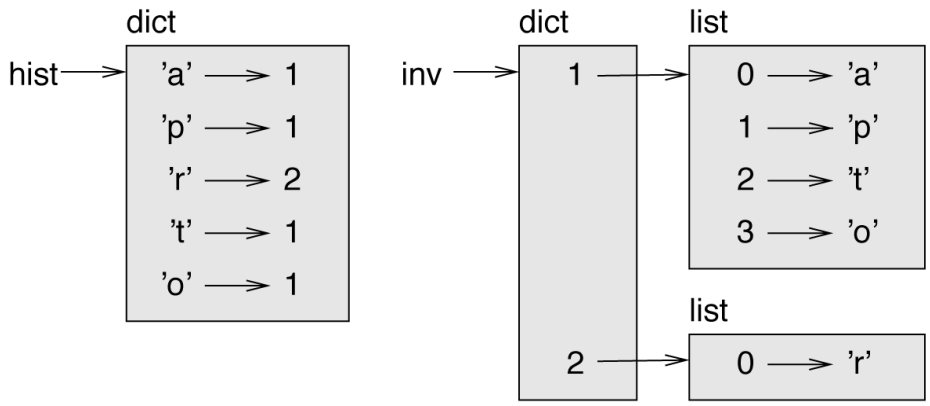

Figura 11-1. Diagrama de estado.

Las listas pueden ser valores en un diccionario, como muestra este ejemplo, pero no pueden ser claves. Esto es lo que sucede si lo intentas:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Mencioné anteriormente que un diccionario se implementa usando una tabla hash y eso significa que las claves tienen que ser **hashable**.

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 buscar pares clave-valor.

Este sistema funciona bien si las claves son inmutables. Pero si las claves son mutables, como listas, suceden cosas malas. Por ejemplo, cuando crea un par de clave-valor, Python mezcla la clave y la almacena en la ubicación correspondiente. Si modificas la clave y luego hash nuevamente, irá a una ubicación diferente. En ese caso, es posible que tenga dos entradas para la misma clave, o es posible que no pueda encontrar una clave. De cualquier manera, el diccionario no funcionaría correctamente.

Es por eso que las claves tienen que ser hashable, y por qué los tipos mutables como las listas no lo son. La forma más sencilla de evitar esta limitación es usar tuplas, que veremos en el próximo capítulo.

Como los diccionarios son mutables, no se pueden usar como claves, pero se pueden usar como valores.

11.6 Memos

Si jugaste con la función `fibonacci` de "Un ejemplo mas", habrás notado que cuanto mayor sea el argumento que proporciones, más tiempo tardará la función en ejecutarse. Además, el tiempo de ejecución aumenta rápidamente.

Para entender por qué, considere la Figura 11-2, que muestra el **gráfico de llamadas** para `fibonaccicon n=4`.

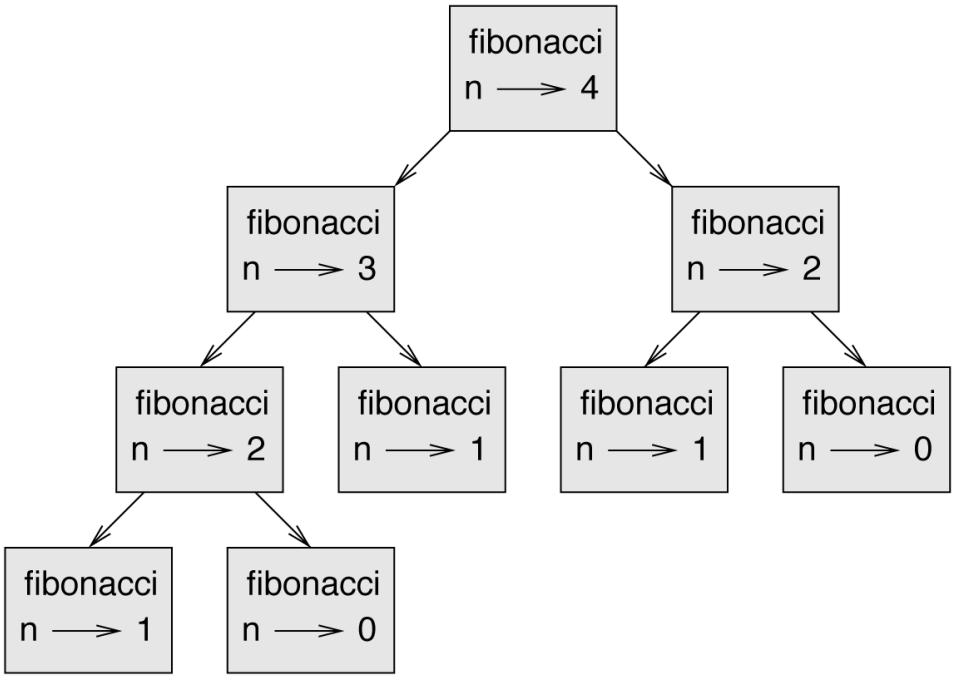

Figura 11-2. Grafico de llamada.

Un gráfico de llamadas muestra un conjunto de marcos de funciones, con líneas que conectan cada cuadro con los marcos de las funciones que llama. En la parte superior del gráfico, llamadas `fibonaccicon n=4`, Fibonacci con n=3 y n=2. A su vez, fibonaccicon n=3 llamadas fibonaccicon n=2 y n=1. Y así.

Cuenta cuántas veces `fibonacci(0)` y `fibonacci(1)` se llaman. Esta es una solución ineficiente al problema, y empeora 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 calculado anteriormente que se almacena para su uso posterior se denomina **memo**. Aquí hay una versión "memorable" de `fibonacci`:

```
known = {0:0, 1:1}


def fibonacci(n):
    if n in known:
        return known[n]


    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

known es un diccionario que realiza un seguimiento de los números de Fibonacci que ya conocemos. Comienza con dos elementos: 0 mapas a 0 y 1 mapas a 1.

Cada vez que `fibonacci` se invoca, comprueba known. Si el resultado ya está allí, puede regresar inmediatamente. De lo contrario, tiene que calcular el nuevo valor, agregarlo al diccionario y devolverlo.

Si ejecuta esta versión de `fibonacci` y la compara con el original, verá que es mucho más rápida.

11.7 Variables globales

En el ejemplo anterior, `known` se crea fuera de la función, por lo que pertenece al marco especial llamado `__main__`. Las variables en `__main__` algunas veces se llaman **globales** porque se puede acceder desde cualquier función. A diferencia de las variables locales, que desaparecen cuando su función finaliza, las variables globales persisten de una llamada de función a la siguiente.

Es común usar variables globales para los **indicadores**; es decir, variables booleanas que indican ("marcar") si una condición es verdadera. Por ejemplo, algunos programas usan un indicador llamado `verbose` para controlar el nivel de detalle en el resultado:

```
verbose = True


def example1():
    if verbose:
        print('Running example1')
```

Si intenta reasignar una variable global, es posible que se sorprenda. Se supone que el siguiente ejemplo realiza un seguimiento de si se ha llamado a la función:

```
been_called = False


def example2():
    been_called = True          # WRONG
```

Pero si lo ejecuta, verá que el valor de been_calledno cambia. El problema es que `example2` crea una nueva variable local llamada `been_called`. La variable local desaparece cuando la función finaliza y no tiene efecto en la variable global.

Para reasignar una variable global dentro de una función, debe declarar la variable global antes de usarla:

```
been_called = False


def example2():
    global been_called
    been_called = True
```

La **declaración global** le dice al intérprete algo así como, "En esta función, cuando digo `been_called`, me refiero a la variable global; no crees uno local ".

Aquí hay un ejemplo que intenta actualizar una variable global:

```
count = 0


def example3():
```

```
    count = count + 1          # WRONG
```

Si lo ejecutas obtienes:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python supone que `count` es local, y bajo esa suposición usted lo está leyendo antes de escribirlo. La solución, una vez más, es declarar `count` como global:

```
def example3():
    global count
    count += 1
```

Si una variable global se refiere a un valor mutable, puede modificar el valor sin declarar la variable:

```
known = {0:0, 1:1}


def example4():
    known[2] = 1
```

Si una variable global se refiere a un valor mutable, puede modificar el valor sin declarar la variable:

```
known = {0:0, 1:1}


def example4():
    known[2] = 1
```

De modo que puede agregar, eliminar y reemplazar elementos de una lista o diccionario global, pero si desea reasignar la variable, debe declararla:

```
def example5():
    global known
    known = dict()
```

Las variables globales pueden ser útiles, pero si tiene muchas, y las modifica con frecuencia, puede dificultar la depuración de los programas.

11.8 Depuración

A medida que trabajas con conjuntos de datos más grandes, puede resultar difícil de depurar imprimiendo y verificando el resultado a mano. Aquí hay algunas sugerencias para depurar conjuntos de datos grandes:

Escala hacia abajo la entrada:

> Si es posible, reduzca el tamaño del conjunto de datos. Por ejemplo, si el programa lee un archivo de texto, comience con solo las primeras 10 líneas, o con el ejemplo más pequeño que pueda encontrar. Puede editar los archivos por sí mismo o (mejor) modificar el programa para que solo lea las primeras n líneas.

Si hay un error, puede reducir n al valor más pequeño que manifiesta el error, y luego aumentarlo gradualmente a medida que encuentre y corrija los errores.

Ver resúmenes y tipos:

En lugar de imprimir y verificar todo el conjunto de datos, considere imprimir resúmenes de los datos: por ejemplo, la cantidad de elementos en un 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 este tipo de error, a menudo es suficiente imprimir el tipo de un valor.

Escribir autocomprobaciones:

Algunas veces puede escribir código para verificar errores automáticamente. Por ejemplo, si está calculando el promedio de una lista de números, puede verificar que el resultado no sea mayor que el elemento más grande de la lista o menor que el más pequeño. Esto se llama un "control de cordura" porque detecta resultados que son "locos".

Otro tipo de verificación compara los resultados de dos cálculos diferentes para ver si son consistentes. Esto se llama una "verificación de coherencia".

Formatee el resultado:

Formatear la salida de depuración puede hacer que sea más fácil detectar un error. Vimos un ejemplo en "6.9 Depuración". El módulo `pprint` proporciona una función `pprint` que muestra los tipos incorporados en un formato más legible para el ser humano (`pprint` significa *pretty print* "impresión bonita").

De nuevo, el tiempo que dedicas a construir andamios puede reducir el tiempo que pasas depurando.

11.10 Glosario

mapeo:

Una relación en la que cada elemento de un conjunto corresponde a un elemento de otro conjunto.

diccionario:

Un mapeo de las claves a sus valores correspondientes.

par clave-valor:

La representación de la asignación de una clave a un valor.

item:

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 forma de realizar un cálculo.

hashable:

El algoritmo utilizado para implementar diccionarios de Python.

función hash:

Una función utilizada por una tabla hash para calcular la ubicación de una clave.

hashable:

Un tipo que tiene una función hash. Los tipos inmutables como enteros, flotadores y cadenas son hashable; tipos mutables como listas y diccionarios no lo son.

buscar:

Una operación de diccionario que toma una clave y encuentra el valor correspondiente.

búsqueda inversa:

Una operación de diccionario que toma un valor y encuentra una o más claves que se asignan a él.

Declaración de aumento:

Una declaración que (deliberadamente) plantea una excepción.

singleton:

Una lista (u otra secuencia) con un solo elemento.

gráfico de llamadas:

Un diagrama que muestra cada fotograma creado durante la ejecución de un programa, con una flecha de cada persona que llama a cada destinatario.

memo:

Un valor calculado almacenado para evitar cálculos futuros innecesarios.

variable global:

Una variable definida fuera de una función. Se puede acceder a las variables globales desde cualquier función.

declaración global:

Una declaración que declara un nombre de variable global.

bandera:

Una variable booleana utilizada para indicar si una condición es verdadera.

declaración:

Una declaración como `global` le dice al intérprete algo sobre una variable.

11.11 Ejercicios

Ejercicio 11-1.

Escriba una función que lea las palabras `words.txt` y las almacene como claves en un diccionario. No importa cuáles sean los valores. Luego puede usar el operador `in` como una forma rápida de verificar si una cadena está en el diccionario.

Si realizó el Ejercicio 10-10, puede comparar la velocidad de esta implementación con el operador `in` de lista y la búsqueda de bisección.

Ejercicio 11-2.

Lea la documentación del método del diccionario `setdefault` y úselo para escribir una versión más concisa de `invert_dict`.

Solución: *http://thinkpython2.com/code/invert_dict.py*.

Ejercicio 11-3.

Recuerde la función de Ackermann del ejercicio 6-2 y vea si la memorización hace posible evaluar la función con argumentos más grandes. Sugerencia: no.

Solución: *http://thinkpython2.com/code/ackermann_memo.py*.

Ejercicio 11-4.

Si realizó el Ejercicio 10-7, ya tiene una función llamada `has_duplicates` que toma una lista como parámetro y devuelve `True` si hay algún objeto que aparece más de una vez en la lista.

Use un diccionario para escribir una versión más rápida y sencilla de `has_duplicates`.

Solución: *http://thinkpython2.com/code/has_duplicates.py*.

Ejercicio 11-5.

Dos palabras son "rotar pares" si puede rotar una de ellas y obtener la otra (ver `rotate_word` en el Ejercicio 8-5).

Escribe un programa que lea una lista de palabras y encuentre todos los pares de rotación.

Solución: *http://thinkpython2.com/code/rotate_pairs.py*.

Ejercicio 11-6.

Aquí hay otro juego de palabras de Car Talk (*http://www.cartalk.com/content/puzzlers*):

Esto fue enviado por un tipo llamado Dan O'Leary. Se encontró con una palabra común de una sílaba, cinco letras recientemente que tiene la siguiente propiedad única. Cuando elimina la primera letra, las letras restantes forman un homófono de la palabra original, es decir, una palabra que suena exactamente igual. Reemplace la primera letra, es decir, vuelva a colocarla y elimine la segunda letra, y el resultado es otro homófono de la palabra original. Y la pregunta es, ¿cuál es la palabra?

Ahora voy a darles un ejemplo que no funciona. Miremos la palabra de cinco letras, 'wrack'. W-R-A-C-K, sabes que 'wrack with pain'. Si elimino la primera letra, me queda una palabra de cuatro letras, 'R-A-C-K'. Al igual que en "Santa vaca, ¿viste el estante de ese dólar? ¡Debe haber sido un nueve puntas! Es un homófono perfecto. Si devuelves la 'w' y quitas la 'r', te queda la palabra 'wack', que es una palabra real, simplemente no es un homófono de las otras dos palabras.

Pero hay, sin embargo, al menos una palabra que Dan y nosotros conocemos, que producirá dos homófonos si eliminas cualquiera de las dos primeras letras para hacer dos palabras nuevas de cuatro letras. La pregunta es, ¿cuál es la palabra?

Puede 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, puede usar el Diccionario de pronunciación de la CMU. Puede descargarlo desde *http://www.speech.cs.cmu.edu/cgi-bin/cmudict* o desde *http://thinkpython2.com/code/c06d* y también puede descargar *http://thinkpython2.com/code/pronoun.py*, que proporciona una función llamada que lee el diccionario de pronunciación y devuelve un diccionario de Python que correlaciona de cada palabra con una cadena que describe su pronunciación principal llamada `read_dictionary`.

Escribe un programa que enumere todas las palabras que resuelven el rompecabezas.

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 funcionan en conjunto. También presento una característica útil para las listas de argumentos de longitud variable: los operadores de recopilación y dispersión.

Una nota: no hay consenso sobre cómo pronunciar "tupla". Algunas personas dicen "tuh-ple", que rima con "supple". Pero en el contexto de la programación, la mayoría de la gente dice "too-ple", que rima con "quadruple".

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 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 entre paréntesis:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un solo elemento, debe incluir una coma final:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

Un valor entre paréntesis no es una tupla:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

Otra forma de crear una tupla es la función incorporada `tuple`. Sin argumentos, 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')
```

Debido a que `tuple` es el nombre de una función incorporada, debe evitar usarlo como un nombre de variable.

La mayoría de los operadores de listas también trabajan con tuplas. El operador de corchete indexa un elemento:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t [0]
'a'
```

Y el operador de sector 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
```

Debido a que las tuplas son inmutables, no puede modificar los elementos. Pero puedes reemplazar una tupla por otra:

```
>>> t = ('A',) + t[1:]
>>> t
('A', 'b', 'c', 'd', 'e')
```

Esta declaración hace una nueva tupla y luego hace treferencia a ella.

Los operadores relacionales trabajan con tuplas y otras secuencias; Python comienza comparando el primer elemento de cada secuencia. Si son iguales, pasa a los siguientes elementos, y así sucesivamente, hasta que encuentra elementos que difieren. 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 Asignar tupla

A menudo es útil intercambiar los valores de dos variables. Con asignaciones convencionales, debe usar una variable temporal. Por ejemplo, para intercambiar `a` y `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Esta solución es engorrosa; 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 se asigna a su variable respectiva. Todas las expresiones en el lado derecho se evalúan antes de cualquiera de las asignaciones.

El número de variables a la izquierda y el número de valores a la derecha 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 dividir una dirección de correo electrónico en un nombre de usuario y un dominio, puede escribir:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

El valor de retorno de `split` es una lista con dos elementos; el primer elemento está asignado a `uname`, el segundo a `domain`:

```
>>> uname
'monty'
>>> dominio
'python.org'
```

12.3 Tuplas como valores de retorno

Estrictamente hablando, una función solo puede devolver un valor, pero si el valor es una tupla, el efecto es el mismo que devolver múltiples valores. Por ejemplo, si desea dividir dos enteros y calcular el cociente y el resto, es ineficaz para calcular `x/y` y luego `x%y`. Es mejor calcularlos a la vez.

La función incorporada `divmod` toma dos argumentos y devuelve una tupla de dos valores: el cociente y el resto. Puede almacenar el resultado como una tupla:

```
>>> t = divmod (7, 3)
>>> t
(2, 1)
```

O use la asignación de tuplas para almacenar los elementos por separado:

```
>>> quot, rem = divmod (7, 3)
>>> quot
```

```
2
>>> rem
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 integradas que encuentran los elementos más grandes y más pequeños de una secuencia. `min_max` calcula ambos y devuelve una tupla de dos valores.

12.4 Tuplas de argumento de longitud variable

Las funciones pueden tomar una cantidad variable de argumentos. Un nombre de parámetro que comienza con *** reunir** argumentos en una tupla. Por ejemplo, `printall` toma cualquier cantidad de argumentos y los imprime:

```
def printall (* args):
    print (args)
```

El parámetro `args` puede tener cualquier nombre que desee, pero `args` es convencional. Así es como funciona la función:

```
>>> printall (1, 2.0, '3')
(1, 2.0, '3')
```

El complemento de reunir es **dispersar**. Si tiene una secuencia de valores y desea pasarla a una función como argumentos múltiples, puede 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 integradas usan tuplas de argumentos de longitud variable. Por ejemplo, `max` y `min` puede tomar cualquier cantidad de argumentos:

```
>>> max (1, 2, 3)
3
```

Pero `sum` no:

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

Como ejercicio, escriba una función llamada `sumall` que toma cualquier cantidad de argumentos y devuelve su suma.

12.5 Listas y Tuplas

`zip` es una función incorporada que toma dos o más secuencias y devuelve una lista de tuplas donde cada tupla contiene un elemento de cada secuencia. El nombre de la función se refiere a una cremallera, que une e intercala dos filas de dientes.

Este ejemplo cierra 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` está en un bucle `for`:

```
>>> for pair in zip(s, t):
...     print(pair)
...
('a', 0)
('b', 1)
('c', 2)
```

Un objeto `zip` es un tipo 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 puede usar un índice para seleccionar un elemento de un iterador.

Si desea utilizar operadores y métodos de lista, puede 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 del más corto:

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

Puede usar la asignación de tuplas en un bucle for para recorrer una lista de tuplas:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Cada vez a través del bucle, Python selecciona la siguiente tupla en la lista y asigna los elementos a `letter` y `number`. La salida de este ciclo es:

```
0 a
1 b
2 c
```

Si combina `zip`, `for` y la asignación de tuplas, obtiene una expresión útil para atravesar dos (o más) secuencias al mismo tiempo. Por ejemplo, `has_match` toma dos secuencias, `t1` y `t2`, y regresa `True` si hay un índice `i` tal que `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Si necesita atravesar los elementos de una secuencia y sus índices, puede usar la función incorporada `enumerate`:

```
for index, element in enumerate('abc'):
    print(index, element)
```

El resultado de enumerate es un objeto de enumerar, 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
```

De nuevo.

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)])
```

El resultado es un objeto `dict_items`, que es un iterador que itera los pares clave-valor. Puedes usarlo en un ciclo for como este:

```
>>> for key, value in d.items():
...     print(key, value)
```

```
...
c 2
a 0
b 1
```

Como debería esperar de un diccionario, los artículos no están en un orden particular.

Yendo en la otra dirección, puede 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}
```

La combinación `dict` con los `zip` rendimientos de una manera concisa para crear un diccionario:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

El método de diccionario `update` también toma una lista de tuplas y las agrega, como pares clave-valor, a un diccionario existente.

Es común usar tuplas como claves en los diccionarios (principalmente porque no puede usar listas). Por ejemplo, un directorio telefónico puede mapear desde el apellido, los primeros nombres hasta los números de teléfono. Suponiendo que hemos definido `last`, `first` y `number`, podríamos escribir:

```
directory[last, first] = number
```

La expresión entre paréntesis es una tupla. Podríamos usar la asignación de tuplas para recorrer este diccionario:

```
for last, first in directory:
    print(first, last, directory[last,first])
```

Este bucle atraviesa las claves de `directory`, que son tuplas. Asigna los elementos de cada tupla a, `last` y `first` luego imprime el nombre y el número de teléfono correspondiente.

Hay dos formas de representar tuplas en un diagrama de estado. La versión más detallada muestra los índices y elementos tal como aparecen en una lista. Por ejemplo, la tupla ('Cleese', 'John') aparecería como en la Figura 12-1.

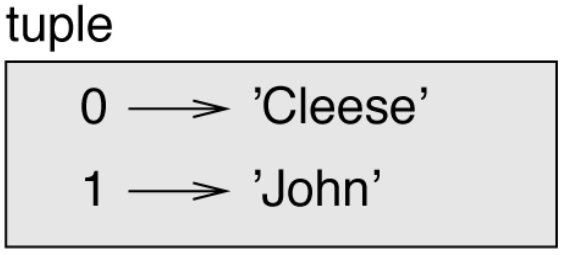

Figura 12-1. Diagrama de estado.

Pero en un diagrama más grande es posible que desee omitir los detalles. Por ejemplo, un diagrama del directorio telefónico podría aparecer como en la Figura 12-2.

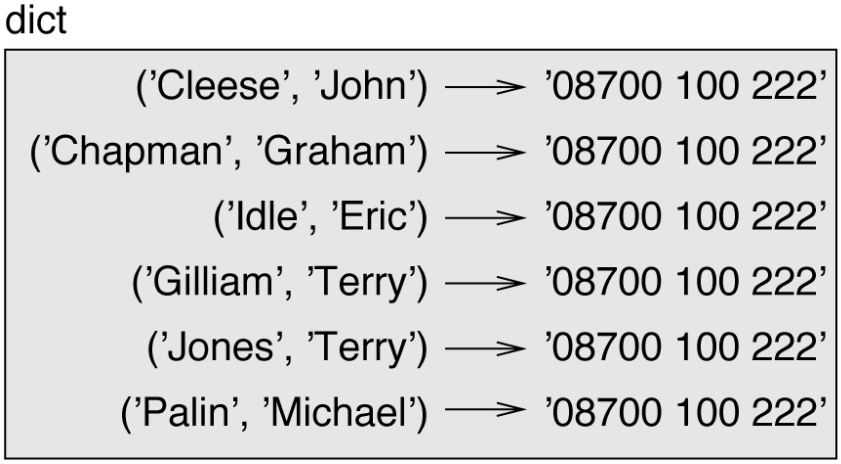

Figura 12-2. Diagrama de estado.

Aquí las tuplas se muestran usando la sintaxis de Python como una abreviatura gráfica. El número de teléfono en el diagrama es la línea de quejas para la BBC, así que no lo llames.

12.7 Secuencias de secuencias

Me he centrado en listas de tuplas, pero casi todos los ejemplos de este capítulo también funcionan 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 elegir uno sobre los demás?

Para comenzar con lo obvio, las cadenas son más limitadas que otras secuencias porque los elementos tienen que ser caracteres. Ellos también son inmutables. Si necesita la capacidad de cambiar los caracteres en una cadena (en lugar de crear una nueva cadena), es posible que desee utilizar una lista de caracteres en su lugar.

Las listas son más comunes que las tuplas, principalmente porque son mutables. Pero hay algunos casos en los que podrías preferir tuplas:

1. En algunos contextos, como una declaración `return`, es sintácticamente más simple crear una tupla que una lista.

2. Si desea usar una secuencia como clave de diccionario, debe usar un tipo inmutable como una tupla o cadena.

3. Si está pasando una secuencia como argumento a una función, el uso de tuplas reduce el potencial de comportamiento inesperado debido al aliasing.

Debido a que las tuplas son inmutables, no proporcionan métodos como `sort` y `reverse`, que modifican las listas existentes. Pero Python proporciona la función incorporada `sorted`, que 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 inverso.

12.8 Depuración

Las listas, diccionarios y tuplas son ejemplos de estructuras de datos; en este capítulo estamos empezando a ver estructuras de datos compuestos, como listas de tuplas o diccionarios que contienen tuplas como claves y listas como valores. Las estructuras de datos compuestas son útiles, pero son propensas a lo que llamo **errores de forma**; es decir, errores causados cuando una estructura de datos tiene un tipo, tamaño o estructura incorrectos. Por ejemplo, si está esperando una lista con un número entero y le doy un número entero simple antiguo (no en una lista), no funcionará.

Para ayudar a depurar este tipo 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. Puede descargarlo desde *http://thinkpython2.com/code/structshape.py*.

Aquí está el resultado de una lista simple:

```
>>> from structshape import structshape
>>> t = [1, 2, 3]
>>> structshape(t)
'list of 3 int'
```

Un programa más elegante podría escribir "lista de 3 int s ", pero era más fácil no tratar con 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 tres elementos que mapean enteros a cadenas:

```
>>> d = dict(lt)
```

```
>>> structshape(d)
'dict of 3 int->str'
```

Si tiene problemas para realizar un seguimiento de sus estructuras de datos, `structshap` epuede 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 se evalúa y luego sus elementos se asignan a las variables de la izquierda.

reunir:

La operación de ensamblar una tupla de argumento de longitud variable.

dispersión:

La operación de tratar una secuencia como una lista de argumentos.

objeto zip:

El resultado de llamar a una función incorporada zip; un objeto que itera a través de una secuencia de tuplas.

iterador:

Un objeto que puede iterar a través de una secuencia, pero que no proporciona operadores y métodos de lista.

estructura de datos:

Una colección de valores relacionados, a menudo organizados en listas, diccionarios, tuplas, etc.

error de forma:

Un error causado porque un valor tiene la forma incorrecta; es decir, el tipo o tamaño equivocado.

12.10 Ejercicios

Ejercicio 12-1.

Escribe una función llamada `most_frequent` que toma una cadena e imprime las letras en orden decreciente de frecuencia. Encuentre ejemplos de texto de diferentes idiomas y vea cómo varía la frecuencia de las letras entre los idiomas. Compare sus 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. Escriba un programa que lea una lista de palabras de un archivo (consulte "Lectura de listas de palabras") e imprima todos los conjuntos de palabras que son anagramas.

   Aquí hay un ejemplo de cómo se vería el resultado:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Sugerencia: es posible que desee construir un diccionario que haga corresponder una colección de letras con una lista de palabras que se pueden escribir con esas letras. La pregunta es, ¿cómo se puede representar la colección de letras de una manera que pueda usarse como clave?

2. Modifique el programa anterior para que imprima primero la lista más larga de anagramas, luego la segunda más larga, y así sucesivamente.

3. En Scrabble, un "bingo" es cuando juegas las siete fichas en tu rack, junto con una letra en el tablero, para formar una palabra de ocho letras. ¿Qué colección de ocho letras forma los bingos más posibles? Sugerencia: hay siete.

   Solución: *http://thinkpython2.com/code/anagram_sets.py*.

Ejercicio 12-3.

Dos palabras forman un "par de metátesis" si puede transformar una en la otra mediante el intercambio de dos letras; por ejemplo, "converse" y "conserve". Escriba un programa que encuentre todos los pares de metátesis en el diccionario. Sugerencia: no pruebe todos los pares de palabras, y no pruebe todos los posibles intercambios.

Solución: http://thinkpython2.com/code/metathesis.py . Crédito: Este ejercicio está inspirado en un ejemplo en http://puzzlers.org .

Ejercicio 12-4.

Aquí hay otro rompecabezas de Car Talk (http://www.cartalk.com/content/puzzlers):

> *¿Cuál es la palabra más larga en inglés, que sigue siendo una palabra válida en inglés, a medida que elimina sus letras de a una por vez?*
>
> *Ahora, las letras se pueden quitar de cualquier extremo, o del centro, pero no se puede reorganizar ninguna de las letras. Cada vez que sueltas una carta, terminas con otra palabra en inglés. Si lo haces, eventualmente terminarás con una letra y esa también será una palabra en inglés, una que se encuentra en el diccionario. Quiero saber cuál es la palabra más larga y cuántas letras tiene?*
>
> *Voy a darles un pequeño ejemplo modesto: Sprite. ¿Okay? Empiezas con un sprite, sacas una carta, una del interior de la palabra, la quitas, y nos queda la palabra rencor, luego tomamos la e del final, nos quedamos con saliva , nos quitamos la s, nos quedamos con pit, it y yo*

Escribe un programa para encontrar todas las palabras que se pueden reducir de esta manera, y luego encuentra la más larga.

1. Este ejercicio es un poco más desafiante que la mayoría, así que aquí hay algunas sugerencias:

2. Es posible que desee escribir una función que tome una palabra y calcule una lista de todas las palabras que pueden formarse eliminando una letra. Estos son los "hijos" de la palabra.

3. Recursivamente, una palabra es reducible si alguno de sus hijos es reducible. Como caso base, puede considerar la cadena vacía reducible.

4. La lista de palabras que proporcioné, words.txtno contiene palabras de una sola letra. Por lo tanto, es posible que desee agregar "I", "a" y la cadena vacía.

Para mejorar el rendimiento de su programa, es posible que desee memorizar las palabras que se sabe que son reducibles.

Solución: *http://thinkpython2.com/code/reducible.py*.

Capítulo 13

Caso de estudio: selección de estructura de datos

En este punto, ha aprendido sobre las estructuras de datos centrales de Python, y ha visto algunos de los algoritmos que las usan. Si desea obtener más información sobre los algoritmos, este podría ser un buen momento para leer el Capítulo 21. Pero no tienes que leerlo antes de continuar; puedes leerlo cuando estés interesado.

Este capítulo presenta un caso de estudio con ejercicios que le permiten pensar en elegir estructuras de datos y practicar su uso.

13.1 Análisis de frecuencia de palabras

Como de costumbre, al menos debes intentar los ejercicios antes de leer mis soluciones.

Ejercicio 13-1.

Escriba un programa que lea un archivo, divida cada línea en palabras, quita el espacio en blanco y la puntuación de las palabras, y las convierte en minúsculas.

Sugerencia: el módulo `string` proporciona una cadena denominada `whitespace`, que contiene espacio, tabulación, nueva línea, etc., y `punctuation` que contiene los caracteres de puntuación. Veamos si podemos hacer que Python jure:

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

Además, podría considerar usar los métodos de cadena `strip`, `replace` y `translate`.

Ejercicio 13-2.

Vaya a Project Gutenberg (http://gutenberg.org) y descargue su libro favorito fuera de los derechos de autor en formato de texto plano.

Modifique su programa del ejercicio anterior para leer el libro que descargó, omita la información del encabezado al principio del archivo y procese el resto de las palabras como antes.

Luego modifique el programa para contar el número total de palabras en el libro y el número de veces que se usa cada palabra.

Imprime la cantidad de palabras diferentes usadas en el libro. Compare diferentes libros de diferentes autores, escritos en diferentes épocas. ¿Qué autor usa el vocabulario más extenso?

Ejercicio 13-3.

Modifique el programa del ejercicio anterior para imprimir las 20 palabras más utilizadas en el libro.

Ejercicio 13-4.

Modifique el programa anterior para leer una lista de palabras (consulte "Lectura de listas de palabras") y luego imprima todas las palabras del libro que no están en la lista de palabras. ¿Cuántos de ellos 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 al azar

Dadas las mismas entradas, la mayoría de los programas de computadora generan los mismos resultados cada vez, por lo que se dice que son deterministas. El **determinismo** suele ser algo bueno, ya que esperamos que el mismo cálculo arroje el mismo resultado. Sin embargo, para algunas aplicaciones, queremos que la computadora sea impredecible. Los juegos son un ejemplo obvio, pero hay más.

Hacer que un programa sea verdaderamente no determinista resulta ser difícil, pero hay formas de hacerlo que al menos parezca no determinista. Una de ellas es usar algoritmos que generan números **pseudoaleatorios**. Los números pseudoaleatorios no son verdaderamente aleatorios porque se generan mediante un cálculo determinista, pero con solo mirar los números, es casi imposible distinguirlos de los aleatorios.

El módulo `random` proporciona funciones que generan números pseudoaleatorios (que a partir de ahora llamaré "aleatorios").

La función `random` devuelve un flotante aleatorio entre 0.0 y 1.0 (incluyendo 0.0 pero no 1.0). Cada vez que llamas `random`, obtienes el siguiente número en una larga serie. Para ver una muestra, ejecuta este ciclo:

```
import random


for i in range(10):
    x = random.random()
    print(x)
```

La función de `randint` toma parámetros `low` y `high` y devuelve un número entero entre `low` y `high` (incluyendo ambos):

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Para elegir un elemento de una secuencia al azar, puede 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, incluidas gaussianas, exponenciales, gamma y algunas más.

Ejercicio 13-5.

Escriba una función llamada `choose_from_hist` que toma un histograma como se define en "Diccionario como una colección de contadores" y devuelve un valor aleatorio del histograma, elegido con probabilidad en proporción a la frecuencia. Por ejemplo, para este histograma:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

su función debería regresar `'a'` con probabilidad 2/3 y `'b'` con probabilidad 1/3.

13.3 Histograma de palabras

Debes intentar los ejercicios previos antes de continuar. Puede descargar mi solución desde *http://thinkpython2.com/code/analyze_book1.py*. También 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 process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist


def process_line(line, hist):
    line = line.replace('-', ' ')


    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1
```

```
hist = process_file('emma.txt')
```

Este programa lee emma.txt, que contiene el texto de *Emma* de Jane Austen.

`process_file` pasa por las líneas del archivo, pasándolos de uno en uno a `process_line`. El histograma `hist` se está utilizando como un acumulador.

`process_line` utiliza el método de cadena `replace` para reemplazar guiones con espacios antes de usar `split` para dividir la línea en una lista de cadenas. Se recorre la lista de palabras y uso `strip`, y `lower` para eliminar puntuacion y convertir a minúsculas. (Es taquigrafía decir que las cadenas se "convierten", recuerde que las cadenas son inmutables, por lo que los métodos como `strip` y `lower` devuelven nuevas cadenas).

Finalmente, `process_line` actualiza el histograma creando un nuevo elemento o incrementando uno existente.

Para contar el número total de palabras en el archivo, podemos sumar las frecuencias en el histograma:

```
def total_words(hist):
    return sum(hist.values())
```

La cantidad de palabras diferentes es solo la cantidad de elementos en el diccionario:

```
def different_words(hist):
    return len(hist)
```

Aquí hay un código para imprimir los resultados:

```
print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))
```

Y los resultados:

```
Total number of words: 161080
Number of different words: 7214
```

13.4 Palabras más comunes

Para encontrar las palabras más comunes, podemos hacer una lista de tuplas, donde cada tupla contiene una palabra y su frecuencia, y ordenarla.

La siguiente función toma un histograma y devuelve una lista de tuplas de frecuencia de palabras:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
```

```
    return t
```

En cada tupla, la frecuencia aparece primero, por lo que la lista resultante se ordena por frecuencia. Aquí hay un ciclo que imprime las 10 palabras más comunes:

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

Utilizo el argumento de palabra clave `sep` para decir que `print` use un carácter de tabulación como un "separador", en lugar de un espacio, por lo que la segunda columna se alinea. Aquí están los resultados de *Emma*:

The most common words are:

```
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 tiene curiosidad, puede leer sobre esto en *https://wiki.python.org/moin/HowTo/Sorting*.

13.5 Parámetros opcionales

Hemos visto funciones y métodos integrados que toman argumentos opcionales. También es posible 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 print_most_common(hist, num=10):
    t = most_common(hist)
    print('The most common words are:')
    for freq, word in t[:num]:
        print(word, freq, sep='\t')
```

El primer parámetro es obligatorio; el segundo es opcional El **valor predeterminado** de `num` es 10.

Si solo proporciona un argumento:

```
print_most_common(hist)
```

num obtiene el valor predeterminado. Si proporciona dos argumentos:

```
print_most_common(hist, 20)
```

num obtiene el valor del argumento en su lugar. En otras palabras, el argumento opcional anula el valor predeterminado.

Si una función tiene parámetros requeridos y opcionales, todos los parámetros requeridos tienen que ser los primeros, seguidos por los opcionales.

13.6 Resta del diccionario

Encontrar las palabras del libro que no están en la lista de palabras `words.txt` es un problema que podría reconocer como una resta establecida; 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).

`subtract` toma diccionarios `d1` y `d2` y devuelve un nuevo diccionario que contiene todas las claves de las `d1` que no están en `d2`. Como realmente no nos importan los valores, los configuramos todos en Ninguno:

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

Para encontrar las palabras en el libro que no están `words.txt`, podemos usar `process_file` para construir un histograma para `words.txt`, y luego restar:

```
words = process_file('words.txt')
diff = subtract(hist, words)


print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

Estos son algunos de los resultados de *Emma*:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Algunas de estas palabras son nombres y posesivos. Otros, como " rencontre", ya no son de uso común. ¡Pero algunas son palabras comunes que realmente deberían estar en la lista!

Ejercicio 13-6.

Python proporciona una estructura de datos llamada `set` que proporciona muchas operaciones de conjunto común. Puede leer sobre ellos en "Conjuntos" o leer la documentación en *http://docs.python.org/3/library/stdtypes.html#types-set*.

Escriba un programa que use la resta establecida para encontrar palabras en el libro que no estén en la lista de palabras.

Solución: *http://thinkpython2.com/code/analyze_book2.py*.

13.7 Palabras aleatorias

Para elegir una palabra aleatoria del histograma, el algoritmo más simple es crear una lista con múltiples copias de cada palabra, de acuerdo con la frecuencia observada, y luego elegir de la lista:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

La expresión `[word] * freq` crea una lista con copias de la cadena `word`. 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 eliges una palabra al azar, reconstruye la lista, que es tan grande como el libro original. Una mejora obvia es construir la lista una vez y luego hacer selecciones múltiples, pero la lista sigue siendo grande.

Una alternativa es:

1. Use `keys` para obtener una lista de las palabras en el libro.

2. Elabore una lista que contenga la suma acumulada de las frecuencias de las palabras (consulte el Ejercicio 10-2). El último elemento en esta lista es el número total de palabras en el libro, `n`.

3. Elija un número aleatorio de 1 a `n`. Utilice una búsqueda de bisección (consulte el Ejercicio 10-10) para encontrar el índice donde se insertaría el número aleatorio en la suma acumulativa.

4. Usa el índice para encontrar la palabra correspondiente en la lista de palabras.

Ejercicio 13-7.

Escriba un programa que use este algoritmo para elegir una palabra aleatoria del libro.

Solución: *http://thinkpython2.com/code/analyze_book3.py*.

13.8 Análisis de Markov

Si eliges palabras del libro al azar, puedes tener 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 las palabras sucesivas. Por ejemplo, en una oración real esperaría que un artículo como "the" sea seguido por un adjetivo o un sustantivo, y probablemente no como un verbo o un adverbio.

Una forma de medir este tipo de relaciones es el análisis de Markov, que caracteriza, para una secuencia dada de palabras, la probabilidad de las palabras que pueden venir después. Por ejemplo, comienza la canción "Eric, the Half a Bee":

- 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" siempre va seguida de la palabra "bee", pero la frase "the bee " puede ir seguida por "has" o "be".

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" y "be").

Dado este mapeo, puede generar un texto aleatorio comenzando con cualquier prefijo y eligiendo al azar entre los posibles sufijos. A continuación, puede combinar el final del prefijo y el nuevo sufijo para formar el siguiente prefijo y repetir.

Por ejemplo, si comienza con el prefijo "Half a", la siguiente palabra debe 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 siempre es dos, pero puede hacer el análisis de Markov con cualquier longitud de prefijo.

Ejercicio 13-8.

Análisis de Markov:

1. Escribe un programa para leer un texto de un archivo y realizar un análisis de Markov. El resultado debería ser un diccionario que correlacione desde prefijos hasta una colección de posibles sufijos. La colección puede ser una lista, tupla o diccionario; Depende de usted hacer una elección adecuada. Puedes probar tu programa con la longitud de prefijo 2, pero deberías escribir el programa de una manera que haga que sea más fácil probar otras longitudes.

2. Agregue una función al programa anterior para generar texto aleatorio basado en el análisis de Markov. Aquí hay un ejemplo de Emma con la longitud del prefijo 2:

   Él fue muy inteligente, ya sea dulzura o estar enojado, avergonzado o solo entretenido, ante tal golpe. ¿Nunca había pensado en Hannah hasta que nunca fuiste para mí? "" No puedo pronunciar discursos, Emma: "él mismo pronto lo cortó todo.

   Para este ejemplo, dejé la puntuación adjunta a las palabras. El resultado es casi sintácticamente correcto, pero no del todo. Semánticamente, casi tiene sentido, pero no del todo.

   Qué sucede si aumenta la longitud del prefijo? ¿El texto al azar tiene más sentido?

3. Una vez que el programa esté funcionando, es posible que desee probar un mash-up: si combina texto de dos o más libros, el texto aleatorio que genere combinará el vocabulario y las frases de las fuentes de maneras interesantes.

Crédito: Este estudio de caso se basa en un ejemplo de Kernighan y Pike, *The Practice of Programming*, Addison-Wesley, 1999.

Debe intentar este ejercicio antes de continuar; entonces puede descargar mi solución desde *http://thinkpython2.com/code/markov.py*. También necesitarás *http://thinkpython2.com/code/emma.txt*.

13.9 Estructuras de datos

Usar el análisis de Markov para generar texto aleatorio es divertido, pero también hay un punto en este ejercicio: selección de estructura de datos. En su solución a los ejercicios anteriores, tenía que elegir:

- Cómo representar los prefijos.
- Cómo representar la colección de posibles sufijos.
- Cómo representar el mapeo desde cada prefijo a la colección de posibles sufijos.

El último es fácil: un diccionario es la elección obvia para un mapeo de las claves a los valores correspondientes.

Para los prefijos, las opciones más obvias son cadena, lista de cadenas o tupla de cadenas.

Para los sufijos, una opción es una lista; otro es un histograma (diccionario).

¿Cómo deberías elegir? El primer paso es pensar en las operaciones que deberá implementar para cada estructura de datos. Para los prefijos, debemos ser capaces de eliminar palabras desde el principio y agregarlas al final. Por ejemplo, si el prefijo actual es "Half a", y la siguiente palabra es "bee", debe ser capaz de formar el siguiente prefijo, "a bee".

Su primera opción podría ser una lista, ya que es fácil agregar y eliminar elementos, pero también debemos poder utilizar los prefijos como claves en un diccionario, por lo que eso excluye las listas. Con tuplas, no puede agregar o eliminar, pero puede usar el operador de suma para formar una nueva tupla:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` toma una tupla de palabras, `prefix` y una cadena, `word` y forma una nueva tupla que tiene todas las palabras, excepto la primera, y se agrega `word` al final.

Para la recopilación de sufijos, las operaciones que debemos realizar incluyen agregar un nuevo sufijo (o aumentar la frecuencia de uno existente) y elegir un sufijo aleatorio.

Agregar un nuevo sufijo es igualmente fácil para la implementación de la lista o el histograma. Elegir un elemento aleatorio de una lista es fácil; elegir de un histograma es más difícil de hacer de manera eficiente (ver Ejercicio 13-7).

Hasta ahora hemos estado hablando principalmente sobre la facilidad de implementación, pero hay otros factores a considerar al elegir las 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 otras; por ejemplo, mencioné que el operador `in` es más rápido para los diccionarios que para las listas, al menos cuando la cantidad de elementos es grande.

Pero a menudo no se sabe de antemano qué implementación será más rápida. Una opción es implementarlos y ver cuál es mejor. Este enfoque se llama **evaluación comparativa**. Una alternativa práctica es elegir la estructura de datos que sea más fácil de implementar y luego ver si es lo suficientemente rápida para la aplicación deseada. Si es así, no hay necesidad de continuar. De lo contrario, hay herramientas, como el módulo `profile`, que pueden identificar los lugares en un programa que requieren más tiempo.

El otro factor a considerar es el espacio de almacenamiento. Por ejemplo, usar un histograma para la colección de sufijos puede tomar menos espacio porque solo tiene que almacenar cada palabra una vez, sin importar cuántas veces aparezca en el texto. En algunos casos, ahorrar espacio también puede hacer que su programa se ejecute más rápido y, en el extremo, su programa podría no ejecutarse en absoluto si se queda sin memoria. Pero para muchas aplicaciones, el espacio es una consideración secundaria después del tiempo de ejecución.

Un pensamiento final: en esta discusión, he dado a entender que deberíamos usar una estructura de datos para el análisis y la generación. Pero dado que estas son fases separadas, también sería posible usar una estructura para el análisis y luego convertir a otra estructura para la generación. Esto sería una ganancia neta si el tiempo ahorrado durante la generación excediera el tiempo empleado en la conversión.

13.8 Depuración

Cuando está depurando un programa, y especialmente si está trabajando en un error, hay cinco cosas que debe intentar:

leerlo:

> Examine su código, léalo de nuevo y verifique que diga lo que quería decir.

ejecutarlo:

> Experimenta haciendo cambios y ejecutando diferentes versiones. A menudo, si muestra lo correcto en el lugar correcto del programa, el problema se vuelve obvio, pero a veces debe construir andamios.

rumiarlo:

> ¡Tómate un tiempo para pensar! ¿Qué tipo de error es: sintaxis, tiempo de ejecución o semántica? ¿Qué información puede obtener de los mensajes de error o de la salida del programa? ¿Qué tipo de error podría causar el problema que estás viendo? ¿Qué cambiaste por última vez antes de que apareciera el problema?

Rubberducking:

Si le explicas el problema a alguien más, a veces encuentras la respuesta antes de que termines de hacer la pregunta. A menudo no necesitas a la otra persona; Podrías hablar con un pato de goma. Y ese es el origen de la estrategia conocida llamada **depuración de pato de goma**. No estoy inventando esto; ver *https://en.wikipedia.org/wiki/Rubber_duck_debugging*.

Retirada:

En algún momento, lo mejor que puede hacer es retroceder y deshacer los cambios recientes hasta que regrese a un programa que funciona y que usted comprende. Entonces puedes comenzar a reconstruir.

Los programadores principiantes a veces se atascan en una de estas actividades y se olvidan de los demás. Cada actividad viene con su propio modo de falla.

Por ejemplo, leer su código podría ayudar si el problema es un error tipográfico, pero no si el problema es un malentendido conceptual. Si no comprende lo que hace su programa, puede leerlo 100 veces y nunca ver el error, porque el error está en su cabeza.

Ejecutar experimentos puede ser útil, especialmente si realiza pruebas pequeñas y simples. Pero si ejecuta experimentos sin pensar o leer su código, puede caer en un patrón que llamo "programación aleatoria", que es el proceso de hacer cambios aleatorios hasta que el programa haga lo correcto. Huelga decir que la programación aleatoria puede llevar mucho tiempo.

Tienes que tomar tiempo para pensar. La depuración es como una ciencia experimental. Debería tener al menos una hipótesis sobre cuál es el problema. Si hay dos o más posibilidades, trate de pensar en una prueba que eliminaría una de ellas.

Pero incluso las mejores técnicas de depuración fallarán si hay demasiados errores o si el código que intentas corregir es demasiado grande y complicado. A veces, la mejor opción es retirarse, simplificando el programa hasta que llegue a algo que funcione y que usted entienda.

Los programadores principiantes a menudo son reacios a retirarse porque no pueden soportar eliminar una línea de código (incluso si está mal). Si te hace sentir mejor, copia tu programa en otro archivo antes de comenzar a desmantelarlo. Luego puedes copiar las piezas una a una.

Encontrar un error difícil requiere leer, ejecutar, reflexionar y, a veces, retirarse. Si te quedas atascado en una de estas actividades, prueba con los demás.

13.9 Glosario

determinista:

Perteneciente a un programa que hace lo mismo cada vez que se ejecuta, con las mismas entradas.

pseudoaleatorio:

Perteneciente a una secuencia de números que parece ser aleatoria, pero que se genera mediante un programa determinista.

valor por defecto:

El valor otorgado a un parámetro opcional si no se proporciona ningún argumento.

anular:

Para reemplazar un valor predeterminado con un argumento.

evaluación comparativa:

El proceso de elegir entre estructuras de datos implementando alternativas y probándolas en una muestra de las posibles entradas.

depuración de patos de goma:

Depuración explicando su problema a un objeto inanimado como un pato de goma. Articular el problema puede ayudarte a resolverlo, incluso si el pato de goma no conoce de Python.

13.10 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 las frecuencias de las palabras en los idiomas naturales (*http://en.wikipedia.org/wiki/Zipf's_law*). Específicamente, predice que la frecuencia, *f*, de la palabra con rango *r* es:

$$f = cr^{-s}$$

donde *s* y *c* son parámetros que dependen del idioma y el texto. Si toma el logaritmo de ambos lados de esta ecuación, obtiene:

$$\log f = \log c - s \log r$$

Entonces, si trazas *log f* frente a *log r*, deberías obtener una línea recta con pendiente *-s* e intercepto *log c*.

Escriba un programa que lea un texto de un archivo, cuente las frecuencias de palabras e imprima una línea para cada palabra, en orden descendente de frecuencia, con *log f* y *log r*. Use el programa de gráficos de su elección para trazar 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 trazado. Si instaló Anaconda, ya lo tiene; de lo contrario, es posible que deba instalarlo `matplotlibmatplotlib`.

Capítulo 14

Archivos

Este capítulo presenta la idea de programas "persistentes" que mantienen los datos en almacenamiento permanente y muestra cómo usar diferentes tipos de almacenamiento permanente, como archivos y bases de datos.

14.1 Persistencia

La mayoría de los programas que hemos visto hasta ahora son transitorios en el sentido de que se ejecutan durante un corto período de tiempo y producen algunos resultados, pero cuando terminan, sus datos desaparecen. Si ejecuta el programa nuevamente, comienza con un borrón y cuenta nueva.

Otros programas son **persistentes**: funcionan durante mucho tiempo (o todo el tiempo); mantienen al menos algunos de sus datos en el almacenamiento permanente (un disco duro, por ejemplo); y si se apagan y reinician, retoman donde lo dejaron.

Los ejemplos de programas persistentes son los sistemas operativos, que se ejecutan prácticamente siempre que una computadora está encendida, y los servidores web, que se ejecutan todo el tiempo, esperando que entren solicitudes en la red.

Una de las maneras más simples para que los programas mantengan sus datos es leyendo y escribiendo archivos de texto. Ya hemos visto programas que leen archivos de texto; en este capítulo veremos programas que los escriben.

Una alternativa es almacenar el estado del programa en una base de datos. En este capítulo, presentaré una base de datos simple y un módulo `pickle`, que hace que sea más fácil almacenar los datos del programa.

14.2 Leyendo y escribiendo

Un archivo de texto es una secuencia de caracteres almacenados en un medio permanente como un disco duro, memoria flash o CD-ROM. Vimos cómo abrir y leer un archivo en "Lectura de listas de palabras".

Para escribir un archivo, debe abrirlo con modo `'w'` como un segundo parámetro:

```
>>> fout = open('output.txt', 'w')
```

Si el archivo ya existe, al abrirlo en modo de escritura borra los datos antiguos y comienza de nuevo, ¡así que tenga cuidado! Si el archivo no existe, se crea uno nuevo.

`open` devuelve un objeto de archivo que proporciona métodos para trabajar con el archivo. El método `write` pone datos en el archivo:

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

```
24
```

El valor de retorno es la cantidad de caracteres que se escribieron. El objeto de archivo realiza un seguimiento de dónde está, por lo que si `write` se vuelve a llamar, agrega los datos nuevos al final del archivo:

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

Cuando haya terminado de escribir, debe cerrar el archivo:

```
>>> fout.close ()
```

Si no cierra el archivo, Python lo cierra por usted cuando finaliza el programa.

14.3 Operador de formato

El argumento de `write` tiene que ser una cadena, por lo que si queremos poner otros valores en un archivo, debemos convertirlos en cadenas. La forma más fácil de hacerlo es con `str`:

```
>>> x = 52
>>> fout.write (str (x))
```

Una alternativa es utilizar el **operador de formato**, %. Cuando se aplica a enteros, % es el operador de módulo. Pero cuando el primer operando es una cadena, % es el operador de formato.

El primer operando es la **cadena de formato**, que contiene una o más **secuencias de formato**, que especifican cómo se formatea el segundo operando. El resultado es una cadena.

Por ejemplo, la secuencia de formateo `'%d'` significa que el segundo operando debe formatearse como un entero decimal:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

El resultado es la cadena `'42'`, que no debe confundirse con el valor entero 42.

Una secuencia de formato puede aparecer en cualquier lugar de la cadena, por lo que puede incrustar un valor en una oración:

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

Si hay más de una secuencia de formato en la cadena, el segundo argumento tiene que ser una tupla. Cada secuencia de formato se empareja con un elemento de la tupla, en orden.

El siguiente ejemplo se utiliza `'%d'` para formatear un entero, `'%g'` formatear un número de coma flotante y `'%s'` formatear una cadena:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
```

```
'In 3 years I have spotted 0.1 camels.'
```

El número de elementos en la tupla debe coincidir con el número de secuencias de formato en la cadena. Además, los tipos de elementos tienen que coincidir con las secuencias de formato:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

En el primer ejemplo, no hay suficientes elementos; en el segundo, el elemento es del tipo incorrecto.

Para obtener más información sobre el operador de formato, consulte *https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting*. Una alternativa más poderosa es el método de formato de cadena, que puede leer en *https://docs.python.org/3/library/stdtypes.html#str.format*.

14.4 Nombres de archivos y rutas

Los archivos están organizados en **directorios** (también llamados "carpetas"). Cada programa en ejecución tiene un "directorio actual", que es el directorio predeterminado para la mayoría de las operaciones. Por ejemplo, cuando abre un archivo para leer, Python lo busca en el directorio actual.

El módulo `os` proporciona funciones para trabajar con archivos y directorios ("os" significa "sistema operativo"). `os.getcwd` devuelve el nombre del directorio actual:

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

`cwd` significa "directorio de trabajo actual". El resultado en este ejemplo es /home/dinsdale, que es el directorio de inicio de un usuario nombrado `dinsdale`.

Una cadena como `'/home/dinsdale'` esa identifica un archivo o directorio y se llama **ruta**.

Un nombre de archivo simple, como `memo.txt`, también se considera una ruta, pero es una ruta relativa porque se relaciona con el directorio actual. Si el directorio actual es /home/dinsdale, el nombre de archivo `memo.txt` se referirá a `/home/dinsdale/memo.txt`.

Una ruta que comienza con / no depende del directorio actual; se llama una ruta **absoluta**. Para encontrar la ruta absoluta a un archivo, puede usar `os.path.abspath`:

```
>>> os.path.abspath ('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` proporciona otras funciones para trabajar con nombres de archivos y rutas. Por ejemplo, `os.path.exists` comprueba si existe un archivo o directorio:

```
>>> os.path.exists ('memo.txt')
```

```
True
```

Si existe, `os.path.isdir` verifica si se trata de un directorio:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

Del mismo modo, `os.path.isfile` verifica si se trata de un archivo.

`os.listdir` devuelve una lista de los archivos (y otros directorios) en el directorio dado:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

Para demostrar estas funciones, el siguiente ejemplo "recorre" un directorio, imprime los nombres de todos los archivos y se llama recursivamente en todos los directorios:

```
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` toma un directorio y un nombre de archivo y los une en una ruta completa.

El módulo `os` proporciona una función llamada `walk` que es similar a esta, pero más versátil. Como ejercicio, lea la documentación y úselo para imprimir los nombres de los archivos en un directorio determinado y sus subdirectorios. Puede descargar mi solución desde *http://thinkpython2.com/code/walk.py*.

14.5 Captura de excepciones

Muchas cosas pueden salir mal cuando intenta leer y escribir archivos. Si intentas abrir un archivo que no existe, obtienes un `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

Si no tienes permiso para acceder a un archivo:

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

Y si intentas abrir un directorio para leer, obtienes

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

Para evitar estos errores, podría usar funciones como `os.path.exists` y `os.path.isfile`, pero tomaría mucho tiempo y código para verificar todas las posibilidades (si `" Errno 21"` es una indicación, hay al menos 21 cosas que pueden salir mal).

Es mejor seguir adelante y tratar, y tratar con los problemas si ocurren, que es exactamente lo que hace la declaración `try`. La sintaxis es similar a una declaración `if...else`:

```
try:
    fin = open('bad_file')
except:
    print('Something went wrong.')
```

Python comienza ejecutando la cláusula `try`. Si todo va bien, omite la cláusula `except` y continúa. Si ocurre una excepción, salta de la cláusula `try` y ejecuta la cláusula `except`.

Manejar una excepción con una declaración `try` se llama capturar una excepción. En este ejemplo, la cláusula `except` imprime un mensaje de error que no es muy útil. En general, atrapar una excepción te da la oportunidad de solucionar el problema, o intentarlo nuevamente, o al menos finalizar el programa correctamente.

14.6 Bases de datos

Una base de datos es un archivo que está organizado para almacenar datos. Muchas bases de datos están organizadas como un diccionario en el sentido de que se asignan desde claves a valores. La mayor diferencia entre una base de datos y un diccionario es que la base de datos está en el disco (u otro almacenamiento permanente), por lo que persiste una vez que finaliza el programa.

El módulo `dbm` proporciona una interfaz para crear y actualizar archivos de base de datos. Como ejemplo, crearé una base de datos que contiene subtítulos para archivos de imagen.

Abrir una base de datos es similar a abrir otros archivos:

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

El modo `'c'` significa que la base de datos debe crearse si aún no existe. El resultado es un objeto de base de datos que se puede usar (para la mayoría de las operaciones) como un diccionario.

Cuando crea un nuevo elemento, `dbm` actualiza el archivo de la base de datos:

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

Cuando accede a uno de los elementos, `dbm` lee el archivo:

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

El resultado es un **objeto bytes**, por lo que comienza con b. Un objeto de bytes es similar a una cadena de muchas maneras. Cuando te adentras en Python, la diferencia se vuelve importante, pero por ahora podemos ignorarla.

Si realiza otra asignación a una clave existente, dbm reemplaza el valor anterior:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Algunos métodos de diccionario, como keys y items, no funcionan con objetos de base de datos. Pero la iteración con un bucle for funciona:

```
for key in db:
    print(key, db[key])
```

Al igual que con otros archivos, debe cerrar la base de datos cuando haya terminado:

```
>>> db.close ()
```

14.7 Decapado

Una limitación de esto dbm es que las claves y los valores tienen que ser cadenas o bytes. Si intenta utilizar cualquier otro tipo, obtendrá un error.

El módulo pickle puede ayudar. Traduce casi cualquier tipo de objeto en una cadena adecuada para el almacenamiento en una base de datos, y luego traduce cadenas en objetos.

pickle.dumps toma un objeto como parámetro y devuelve una representación de cadena (dumps es la abreviatura de "cadena de volcado"):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

El formato no es obvio para los lectores humanos; está destinado a ser fácil de interpretar. pickle.loads ("Cadena de carga") reconstituye el objeto:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps (t1)
>>> t2 = pickle.loads (s)
>>> t2
[1, 2, 3]
```

Aunque el nuevo objeto tiene el mismo valor que el anterior, no es (en general) el mismo objeto:

```
>>> t1 == t2
True
```

```
>>> t1 is t2
False
```

En otras palabras, decapado y luego descortezado tiene el mismo efecto que copiar el objeto.

Puede usar `pickle` para almacenar cadenas en una base de datos. De hecho, esta combinación es tan común que se ha encapsulado en un módulo llamado `shelve`.

14.8 Shell

La mayoría de los sistemas operativos proporcionan una interfaz de línea de comandos, también conocida como **shell**. Las shells generalmente proporcionan comandos para navegar por el sistema de archivos y ejecutar aplicaciones. Por ejemplo, en Unix puede cambiar directorios cd, mostrar los contenidos de un directorio `ls` e iniciar un navegador web escribiendo (por ejemplo) `firefox`.

Cualquier programa que pueda iniciarse desde el shell también se puede iniciar desde Python utilizando un **objeto pipe**, que representa un programa en ejecución.

Por ejemplo, el comando Unix `ls -l` normalmente muestra los contenidos del directorio actual en formato largo. Puede iniciar `ls` con `os.popen1`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

El argumento es una cadena que contiene un comando de shell. El valor de retorno es un objeto que se comporta como un archivo abierto. Puede leer el resultado del proceso `ls` una línea a la vez con `readline` o obtener todo de una vez con `read`:

```
>>> res = fp.read ()
```

Cuando termine, cierre la shell como un archivo:

```
>>> stat = fp.close ()
>>> print (stat)
None
```

El valor de retorno es el estado final del proceso `ls`; `None` significa que terminó normalmente (sin errores).

Por ejemplo, la mayoría de los sistemas Unix proporcionan un comando llamado `md5sum` que lee el contenido de un archivo y calcula una "suma de comprobación". Puede leer sobre MD5 en *http://en.wikipedia.org/wiki/Md5*. Este comando proporciona una forma eficiente de verificar si dos archivos tienen el mismo contenido. La probabilidad de que diferentes contenidos produzcan la misma suma de comprobación es muy pequeña (es decir, es poco probable que suceda antes de que el universo colapse).

Puede usar una Shell para ejecutar `md5sum` desde Python y obtener el resultado:

```
>>> 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 Escritura de módulos

Cualquier archivo que contenga código Python se puede importar como un módulo. Por ejemplo, supongamos que tiene un archivo nombrado `wc.py` con el siguiente código:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count


print(linecount('wc.py'))
```

Si ejecuta este programa, se lee e imprime el número de líneas en el archivo, que es 7. También puede importarlo así:

```
>>> import wc
7
```

Ahora tienes un objeto de módulo `wc`:

```
>>> wc
<module 'wc' from 'wc.py'>
```

El objeto del módulo proporciona `linecount`:

```
>>> wc.linecount ('wc.py')
7
```

El único problema con este ejemplo es que cuando importa el módulo, ejecuta el código de prueba en la parte inferior. Normalmente, cuando importa un módulo, define nuevas funciones pero no las ejecuta.

Los programas que se importarán como módulos a menudo usan el siguiente modismo:

```
if __name__ == '__main__':
   print(linecount('wc.py'))
```

`__name__` es una variable incorporada que se establece cuando se inicia el programa. Si el programa se ejecuta como un script, `__name__` tiene el valor `'__main__'`; en ese caso, se ejecuta el código de prueba. De lo contrario, si el módulo se está importando, el código de prueba se omite.

Como ejercicio, escriba este ejemplo en un archivo llamado `wc.py` y ejecútelo como un script. Luego ejecuta el intérprete de Python y import `wc`. ¿Cuál es el valor de `__name__` cuándo se está importando el módulo?

Advertencia: si importa un módulo que ya ha sido importado, Python no hace nada. No vuelve a leer el archivo, incluso si ha cambiado.

Si desea volver a cargar un módulo, puede usar la función incorporada `reload`, pero puede ser complicado, por lo que lo más seguro es reiniciar el intérprete y luego importar el módulo nuevamente.

14.10 Depuración

Cuando está leyendo y escribiendo archivos, puede tener problemas con el espacio en blanco. Estos errores pueden ser difíciles de depurar porque los espacios, las pestañas y las nuevas líneas son normalmente invisibles:

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2          3
 4
```

La función incorporada `repr` puede ayudar. Toma cualquier objeto como argumento y devuelve una representación de cadena del objeto. Para cadenas, representa caracteres de espacios en blanco con secuencias de barra inclinada invertida:

```
>>> print (repr (s))
'1 2 \ t 3 \ n 4'
```

Esto puede ser útil para la depuración.

Otro problema con el que puede encontrarse es que diferentes sistemas usan caracteres diferentes para indicar el final de una línea. Algunos sistemas usan una línea nueva, representada \n. Otros usan un personaje de retorno, representado `\r`. Algunos usan ambos. Si mueve archivos entre diferentes sistemas, estas incoherencias pueden causar problemas.

Para la mayoría de los sistemas, hay aplicaciones para convertir de un formato a otro. Puede encontrarlos (y leer más sobre este tema) en *http://en.wikipedia.org/wiki/Newline*. O, por supuesto, podrías escribir uno tú mismo.

14.11 Glosario

persistente:

> Perteneciente a un programa que se ejecuta indefinidamente y mantiene al menos algunos de sus datos en el almacenamiento permanente.

operador de formato:

> Un operador, `%` que toma una cadena de formato y una tupla y genera una cadena que incluye los elementos de la tupla formateados como se especifica en la cadena de formato.

cadena de formato:

Una cadena, utilizada con el operador de formato, que contiene secuencias de formato.

secuencia de formateo:

Una secuencia de caracteres en una cadena de formato, como `%d`, que especifica cómo debe formatearse un valor.

Archivo de texto:

Una secuencia de caracteres almacenados en un almacenamiento permanente como un disco duro.

directorio:

Una colección de archivos con nombre, también llamada carpeta.

path:

Una cadena que identifica un archivo.

path relativo:

Una ruta que comienza desde el directorio actual.

path absoluto:

Una ruta que comienza desde el directorio más alto en el sistema de archivos.

captura:

Para evitar que una excepción termine un programa usando las instrucciones `try` y *except*.

base de datos:

Un archivo cuyos contenidos están organizados como un diccionario con claves que corresponden a valores.

Objeto de bytes:

Un objeto similar a una cadena.

shell:

Un programa que permite a los usuarios escribir comandos y luego los ejecuta iniciando otros programas.

objeto de shell:

Un objeto que representa un programa en ejecución, que permite que un programa de Python ejecute comandos y lea los resultados.

14.12 Ejercicios

Ejercicio 14-1.

Escriba una función llamada sed que toma como argumentos una cadena de patrón, una cadena de reemplazo y dos nombres de archivo; debería leer el primer archivo y escribir el contenido en el segundo archivo (crearlo si es necesario). Si la cadena del patrón aparece en alguna parte del archivo, debe reemplazarse con la cadena de reemplazo.

Si se produce un error al abrir, leer, escribir o cerrar archivos, su programa debería detectar la excepción, imprimir un mensaje de error y salir.

Solución: *http://thinkpython2.com/code/sed.py*.

Ejercicio 14-2.

Si descarga mi solución al ejercicio 12-2 de *http://thinkpython2.com/code/anagram_sets.py*, verá que crea un diccionario que mapea desde una cadena de letras ordenada a la lista de palabras que pueden ser deletreado con esas letras. Por ejemplo, mapas a la lista. `'opst'['opts', 'post', 'pots', 'spot', 'stop', 'tops']`

Escriba un módulo que importe `anagram_sets` y proporcione dos nuevas funciones: `store_anagrams` debe almacenar el diccionario de anagramas en un "estante"; `read_anagrams` debería buscar una palabra y devolver una lista de sus anagramas.

Solución: *http://thinkpython2.com/code/anagram_db.py*

Ejercicio 14-3.

En una gran colección de archivos MP3, puede haber más de una copia de la misma canción, almacenada en diferentes directorios o con diferentes nombres de archivo. El objetivo de este ejercicio es buscar duplicados.

4. Escriba un programa que busque un directorio y todos sus subdirectorios, recursivamente, y devuelva una lista de rutas completas para todos los archivos con un sufijo dado (como `.mp3`). Sugerencia: `os.path` proporciona varias funciones útiles para manipular nombres de archivos y rutas.

5. Para reconocer duplicados, puede usar `md5sum` para calcular una "suma de comprobación" para cada archivo. Si dos archivos tienen la misma suma de comprobación, probablemente tengan el mismo contenido.

6. Para volver a verificar, puede usar el comando Unix `diff`.

Solución: *http://thinkpython2.com/code/find_duplicates.py*.

Capítulo 15

Clases y objetos

En este punto, usted sabe cómo usar las funciones para organizar el código y los tipos incorporados para organizar los datos. El siguiente paso es aprender "programación orientada a objetos", que usa tipos definidos por programador para organizar código y datos. La programación orientada a objetos es un gran tema; tomará algunos capítulos para llegar allí.

Los ejemplos de código de este capítulo están disponibles en *http://thinkpython2.com/code/Point1.py*; las soluciones para los ejercicios están disponibles en *http://thinkpython2.com/code/Point1_soln.py*.

15.1 Tipos definidos por programador

Hemos utilizado muchos de los tipos incorporados de Python; ahora vamos a definir un nuevo tipo. Como ejemplo, crearemos un tipo llamado `Point` que representa un punto en el espacio bidimensional.

En notación matemática, los puntos a menudo se escriben entre paréntesis con una coma que separa las coordenadas. Por ejemplo, *(0,0)* representa el origen y *(x, y)* representa el punto *x* unidades a la derecha y y unidades a partir del origen.

Hay varias formas en que podríamos representar puntos en Python:

- Podríamos almacenar las coordenadas por separado en dos variables, `x` y `y`.
- Podríamos almacenar las coordenadas como elementos en una lista o tupla.
- Podríamos crear un nuevo tipo para representar puntos como objetos.

Crear un nuevo tipo es más complicado que las otras opciones, pero tiene ventajas que serán evidentes pronto.

Un tipo definido por el programador también se llama **clase**. Una definición de clase se ve así:

```
class Point:
    """Represents a point in 2-D space."""
```

El encabezado indica que se llama a la nueva clase `Point`. El cuerpo es un docstring que explica para qué es la clase. Puede definir variables y métodos dentro de una definición de clase, pero volveremos sobre eso más adelante.

La definición de una clase llamada `Point` y crea un objeto de **clase**:

```
>>> Point
<class '__main__.Point'>
```

Porque `Point` se define en el nivel superior, su "nombre completo" es `__main__.Point`.

El objeto de clase es como una fábrica para crear objetos. Para crear un Punto, llama `Point` como si fuera una función:

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

El valor de retorno es una referencia a un objeto `Point`, que le asignamos blank.

Crear un nuevo objeto se llama **instanciación**, y el objeto es una **instancia** de la clase.

Cuando imprime una instancia, Python le dice a qué clase pertenece y dónde está almacenada en la memoria (el prefijo 0x significa que el siguiente número está en hexadecimal).

Cada objeto es una instancia de alguna clase, por lo que "objeto" e "instancia" son intercambiables. Pero en este capítulo utilizo "instancia" para indicar que estoy hablando de un tipo definido por el programador.

15.2 Atributos

Puede asignar valores a una instancia usando notación de puntos:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

Esta sintaxis es similar a la sintaxis para seleccionar una variable de un módulo, como `math.pi` o `string.whitespace`. En este caso, sin embargo, estamos asignando valores a los elementos con nombre de un objeto. Estos elementos se llaman **atributos**.

Como sustantivo, "AT-trib-ute" se pronuncia con énfasis en la primera sílaba, en oposición a "a-TRIB-ute", que es un verbo.

El siguiente diagrama muestra el resultado de estas asignaciones. Un diagrama de estado que muestra un objeto y sus atributos se llama **diagrama de objeto**; ver la Figura 15-1.

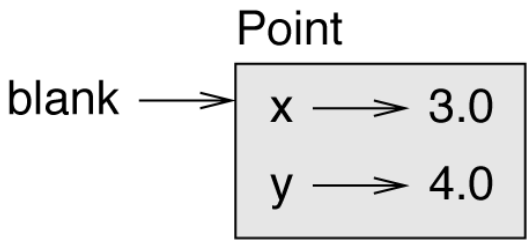

Figura 15-1. Diagrama de objeto.

La variable se `blank` refiere a un objeto Point, que contiene dos atributos. Cada atributo se refiere a un número de coma flotante.

Puede leer el valor de un atributo usando la misma sintaxis:

```
>>> blank.y
4.0
```

```
>>> x = blank.x
>>> x
3.0
```

La expresión `blank.x` significa, "Ir al objeto `blank` hace referencia y obtener el valor de x." En el ejemplo, asignamos ese valor a una variable nombrada x. No hay conflicto entre la variable x y el atributo x.

Puede usar la notación de puntos como parte de cualquier expresión. Por ejemplo:

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

Puede pasar una instancia como un argumento de la manera habitual. Por ejemplo:

```
def print_point(p):
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` toma un punto como argumento y lo muestra en notación matemática. Para invocarlo, puede pasar `blank` como argumento:

```
>>> print_point(blank)
(3.0, 4.0)
```

Dentro de la función, `p` es un alias para `blank`, por lo que si la función se modifica `p`, `blank` cambia.

Como ejercicio, escriba una función llamada `distance_between_points` que toma dos puntos como argumentos y devuelve la distancia entre ellos.

15.3 Rectángulos

A veces es obvio cuáles deberían ser los atributos de un objeto, pero otras veces debes tomar decisiones. Por ejemplo, imagine que está diseñando una clase para representar rectángulos. ¿Qué atributos usarías para especificar la ubicación y el tamaño de un rectángulo? Puedes ignorar el ángulo; para mantener las cosas simples, suponga que el rectángulo es vertical u horizontal.

Hay al menos dos posibilidades:

- Puede especificar una esquina del rectángulo (o el centro), el ancho y la altura.
- Podrías especificar dos esquinas opuestas.

En este punto, es difícil decir si alguno es mejor que el otro, así que implementaremos el primero, solo como un ejemplo.

Aquí está la definición de clase:

```
class Rectangle:
```

```
    """Represents a rectangle.


    attributes: width, height, corner.
    """
```

La docstring enumera los atributos: `width` y `height` son números; `corner` es un objeto Point que especifica la esquina inferior izquierda.

Para representar un rectángulo, debe instanciar un objeto Rectangle y asignar valores a los atributos:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

La expresión `box.corner.x` significa, "Ir al objeto se `box` refiere y seleccionar el atributo llamado `corner`; luego ve a ese objeto y selecciona el atributo llamado x".

La figura 15-2 muestra el estado de este objeto. Un objeto que es un atributo de otro objeto está **incrustado**.

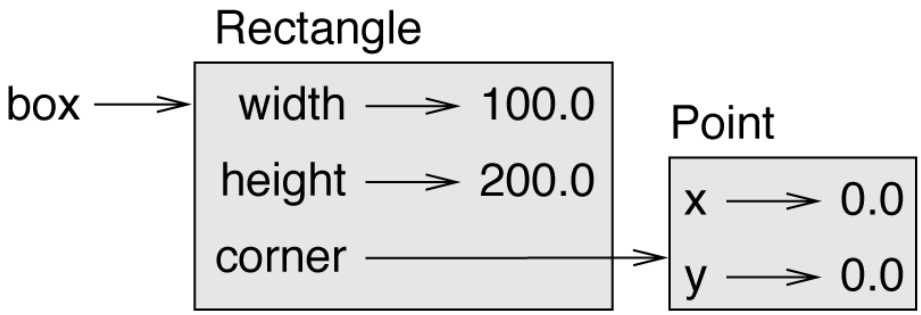

Figura 15-2. Diagrama de objeto.

15.4 Instancias como valores de retorno

Las funciones pueden devolver instancias. Por ejemplo, `find_center` toma a `Rectangle` como argumento y devuelve a `Point` que contiene las coordenadas del centro de `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
```

Aquí hay un ejemplo que pasa `box` como argumento y asigna el punto resultante a `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

15.5 Los objetos son mutables

Puede cambiar el estado de un objeto haciendo una asignación a uno de sus atributos. Por ejemplo, para cambiar el tamaño de un rectángulo sin cambiar su posición, puede modificar los valores de `width` y `height`:

```
box.width = box.width + 50
box.height = box.height + 100
```

También puede escribir funciones que modifican objetos. Por ejemplo, `grow_rectangle` toma un objeto rectangular y dos números, `dwidth` y `dheight`, y añade los números para la anchura y la altura del rectángulo:

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Aquí hay un ejemplo que demuestra el efecto:

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

Dentro de la función, `rect` es un alias para `box`, así que cuando la función se modifica `rect`, `box` cambia.

Como ejercicio, escriba una función llamada `move_rectangle` que toma un Rectángulo y dos números nombrados `dx` y `dy`. Debería cambiar la ubicación del rectángulo al agregar `dx` a la coordenada x de `corner` y agregar `dy` a la coordenada y de `corner`.

15.6 Proceso de copiar

Aliasing puede hacer que un programa sea difícil de leer porque los cambios en un lugar pueden tener efectos inesperados en otro lugar. Es difícil hacer un seguimiento de todas las variables que pueden referirse a un objeto dado.

Copiar un objeto suele ser una alternativa al aliasing. El copymódulo contiene una función llamada copyque puede duplicar cualquier objeto:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0


>>> import copy
```

```
>>> p2 = copy.copy(p1)
```

p1 y p2 contienen los mismos datos, pero no son el mismo Punto:

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

El operador `is` lo indica `p1` y `p2` no son el mismo objeto, que es lo que esperábamos. Pero es posible que haya esperado == ceder `True` porque estos puntos contienen los mismos datos. En ese caso, se sentirá decepcionado al saber que, por ejemplo, el comportamiento predeterminado del operador == es el mismo que el del is operador; comprueba la identidad del objeto, no la equivalencia del objeto. Esto se debe a que para los tipos definidos por el programador, Python no sabe qué debería considerarse equivalente. Al menos no todavía.

Si usa `copy.copy` para duplicar un rectángulo, encontrará que copia el objeto Rectangle pero no el punto incrustado:

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

La figura 15-3 muestra cómo se ve el diagrama de objetos. Esta operación se denomina **copia superficial** porque copia el objeto y las referencias que contiene, pero no los objetos incrustados.

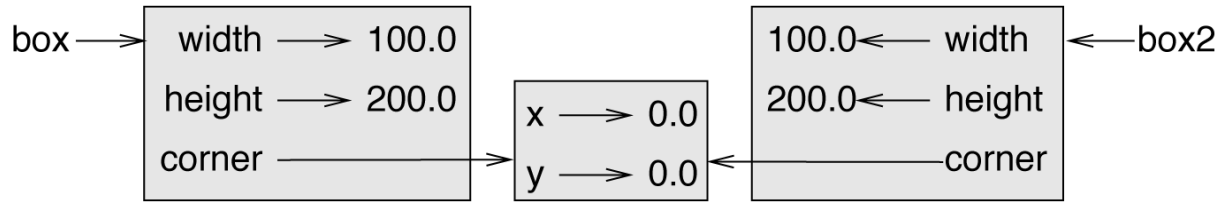

Figura 15-3. Diagrama de objeto.

Para la mayoría de las aplicaciones, esto no es lo que quieres. En este ejemplo, invocar `grow_rectangle` en uno de los Rectángulos no afectaría al otro, ¡pero invocar `move_rectangle` en cualquiera afectaría a ambos! Este comportamiento es confuso y propenso a errores.

Afortunadamente, el módulo `copy` proporciona un método llamado `deepcopy` que copia no solo el objeto sino también los objetos a los que hace referencia, y los objetos a los que se refieren, y así sucesivamente. No se sorprenderá al saber que esta operación se llama **copia profunda**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 y box son objetos completamente separados.

Como ejercicio, escribe una versión de `move_rectangle` eso crea y devuelve un nuevo rectángulo en lugar de modificar el anterior.

15.7 Depuración

Cuando empiezas a trabajar con objetos, es probable que encuentres algunas excepciones nuevas. Si intenta acceder a un atributo que no existe, obtiene un `AttributeError`:

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

Si no está seguro de qué tipo es un objeto, puede preguntar:

```
>>> type(p)
<class '__main__.Point'>
```

También puede usar `isinstance` para verificar si un objeto es una instancia de una clase:

```
>>> isinstance(p, Point)
True
```

Si no está seguro de si un objeto tiene un atributo en particular, puede usar la función incorporada `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

El primer argumento puede ser cualquier objeto; el segundo argumento es una cadena que contiene el nombre del atributo.

También puede usar una declaración `try` para ver si el objeto tiene los atributos que necesita:

```
try:
    x = p.x
```

```
except AttributeError:
    x = 0
```

Este enfoque puede hacer que sea más fácil escribir funciones que funcionen con diferentes tipos; más sobre ese tema está apareciendo en "Polimorfismo".

15.8 Glosario

clase:

> Un tipo definido por el programador. Una definición de clase crea un nuevo objeto de clase.

objeto de clase:

> Un objeto que contiene información sobre un tipo definido por el programador. El objeto de clase se puede usar para crear instancias del tipo.

ejemplo:

> Un objeto que pertenece a una clase.

instanciar:

> Para crear un nuevo objeto.

atributo:

> Uno de los valores nombrados asociados con un objeto.

objeto incrustado:

> Un objeto que se almacena como un atributo de otro objeto.

copia superficial:

> Para copiar el contenido de un objeto, incluidas las referencias a objetos incrustados; implementado por la función `copy` en el módulo `copy`.

copia profunda

> Para copiar el contenido de un objeto, así como cualquier objeto incrustado, y cualquier objeto incrustado en ellos, y así sucesivamente; implementado por la función `deepcopy` en el módulo `copy`.

Diagrama de objeto:

> Un diagrama que muestra los objetos, sus atributos y los valores de los atributos.

15.9 Ejercicios

Ejercicio 15-1.

Escriba una definición para una clase nombrada `Circle` con atributos `center` y `radius`, donde `center` es un objeto Punto y el radio es un número.

Crea una instancia de un objeto Circle que represente un círculo con su centro en (150, 100) y radio 75.

Escriba una función llamada `point_in_circle` que toma un Círculo y un Punto y devuelve Verdadero si el Punto se encuentra dentro o sobre el límite del círculo.

Escribe una función llamada `rect_in_circle` que toma un Círculo y un Rectángulo y devuelve Verdadero si el Rectángulo se encuentra completamente dentro o sobre el límite del círculo.

Escribe una función llamada `rect_circle_overlap` que toma un Círculo y un Rectángulo y devuelve Verdadero si alguna de las esquinas del Rectángulo cae dentro del círculo. O como una versión más desafiante, devuelve True si alguna parte del Rectángulo cae dentro del círculo.

Solución: *http://thinkpython2.com/code/Circle.py*.

Ejercicio 15-2.

Escribe una función llamada `draw_rect` que toma un objeto Tortuga y un Rectángulo y usa la Tortuga para dibujar el Rectángulo. Ver el Capítulo 4 para ejemplos usando objetos Tortuga.

Escribe una función llamada `draw_circle` que toma una Tortuga y un Círculo y dibuja el Círculo.

Solución: *http://thinkpython2.com/code/draw.py*.

Capítulo 16

Clases y funciones

Ahora que sabemos cómo crear nuevos tipos, el siguiente paso es escribir funciones que tomen objetos definidos por el programador como parámetros y devolverlos como resultados. En este capítulo también presento "estilo de programación funcional" y dos nuevos planes de desarrollo de programa.

Los ejemplos de código de este capítulo están disponibles en *http://thinkpython2.com/code/Time1.py*. Las soluciones para los ejercicios están en *http://thinkpython2.com/code/Time1_soln.py*.

16.1 Hora

Como otro ejemplo de un tipo definido por el programador, definiremos una clase llamada `Time` que registra la hora del día. La definición de clase se ve así:

```
class Time:
    """Represents the time of day.


    attributes: hour, minute, second
    """
```

Podemos crear un nuevo objeto `Time` y asignar atributos para horas, minutos y segundos:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

El diagrama de estado para el objeto `Time` se parece a la Figura 16-1.

Como ejercicio, escriba una función llamada `print_time` que toma un objeto `Time` e imprime en el formulario `hour:minute:second`. Sugerencia: la secuencia de formato `'%.2d'` imprime un número entero utilizando al menos dos dígitos, incluido un cero inicial si es necesario.

Escribir una función booleana llamada `is_after` que toma dos objetos Tiempo, `t1` y `t2`, y devuelve `True` si `t1` la siguiente manera `t2` cronológica y `False` de otra manera. Desafío: no use una declaración `if`.

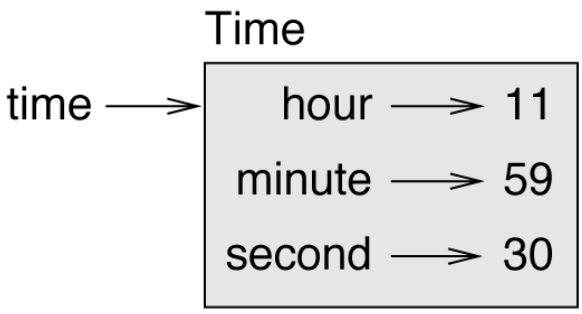

Figura 16-1. Diagrama de objeto.

16.2 Funciones Puras

En las siguientes secciones, escribiremos dos funciones que agregan valores de tiempo. Demuestran dos tipos de funciones: funciones puras y modificadores. También demuestran un plan de desarrollo al que llamaré **prototipo y parche**, que es una forma de abordar un problema complejo comenzando con un prototipo simple y abordando de forma incremental las complicaciones.

Aquí hay un prototipo simple de `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
```

La función crea un nuevo objeto `Time`, inicializa sus atributos y devuelve una referencia al nuevo objeto. Esto se denomina **función pura** porque no modifica ninguno de los objetos que se le pasan como argumentos y no tiene ningún efecto, como mostrar un valor o recibir una entrada del usuario, que no sea devolver un valor.

Para probar esta función, crearé dos objetos de Time: `start` contiene la hora de inicio de una película, como *Monty Python* y el *Santo Grial*, y `duration` contiene el tiempo de ejecución de la película, que es de 1 hora y 35 minutos.

`add_time` se da cuenta cuando la película estará lista:

```
>>> 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
```

El resultado, `10:80:00` puede que no sea lo que esperabas. El problema es que esta función no se ocupa de casos en los que la cantidad de segundos o minutos suma más de sesenta. Cuando eso sucede, tenemos que "cargar" los segundos adicionales en la columna de minutos o los minutos adicionales en la columna de la hora.

Aquí hay una versión mejorada:

```
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
```

Aunque esta función es correcta, está empezando a ser grande. Veremos una alternativa más corta más adelante.

16.3 Modificadores

Algunas veces es útil para una función modificar los objetos que obtiene como parámetros. En ese caso, los cambios son visibles para la persona que llama. Las funciones que funcionan de esta manera se llaman **modificadores**.

`increment`, que agrega un número dado de segundos a un objeto `Time`, se puede escribir naturalmente como un modificador. Aquí hay un borrador:

```
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
```

La primera línea realiza la operación básica; el resto trata de los casos especiales que vimos antes.

¿Es esta función correcta? ¿Qué pasa si `second` ses mucho mayor que 60?

En ese caso, no es suficiente llevar una vez; tenemos que seguir haciéndolo hasta que `time.second` sea menor a 60. Una solución es reemplazar las declaraciones `if` con declaraciones `while`. Eso haría que la función sea correcta, pero no muy eficiente. Como ejercicio, escribe una versión correcta `increment` que no contenga ningún bucle.

Todo lo que se puede hacer con modificadores también se puede hacer con funciones puras. De hecho, algunos lenguajes de programación solo permiten funciones puras. Existe cierta evidencia de que los programas que usan funciones puras son más rápidos de desarrollar y menos propensos a errores que los programas que usan modificadores. Pero los modificadores son convenientes a veces, y los programas funcionales tienden a ser menos eficientes.

En general, le recomiendo que escriba funciones puras siempre que sea razonable y que recurra a modificadores solo si hay una ventaja convincente. Este enfoque podría llamarse un estilo de **programación funcional**.

Como ejercicio, escriba una versión "pura" de `increment` eso crea y devuelve un nuevo objeto Time en lugar de modificar el parámetro.

16.4 Prototipos versus planificación

El plan de desarrollo que estoy demostrando se llama "prototipo y parche". Para cada función, escribí un prototipo que realizó el cálculo básico y luego lo probé, corrigiendo errores en el camino.

Este enfoque puede ser efectivo, especialmente si aún no tiene una comprensión profunda del problema. Pero las correcciones incrementales pueden generar código que es innecesariamente complicado (dado que se trata de muchos casos especiales) y no confiable (ya que es difícil saber si ha encontrado todos los errores).

Una alternativa es el **desarrollo diseñado**, en el que una visión de alto nivel del problema puede facilitar la programación. En este caso, la idea es que un objeto Time es realmente un número de tres dígitos en la base 60 (ver *http://en.wikipedia.org/wiki/Sexagesimal*)! El atributo es la "columna de unos", el atributo es la "columna de los sesenta", y el atributo es la "columna de treinta y seis cientos" `secondminutehour`.

Cuando escribimos `add_time` y `increment`, efectivamente, estábamos haciendo una adición en la base 60, que es por lo que tuvimos que llevar de una columna a la siguiente.

Esta observación sugiere otro enfoque para todo el problema: podemos convertir objetos de tiempo en enteros y aprovechar el hecho de que la computadora sabe cómo hacer aritmética de enteros.

Aquí hay una función que convierte Times en enteros:

```
def time_to_int(time):
```

```
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

Y aquí hay una función que convierte un entero en un Tiempo (recuerdo que `divmod` divide el primer argumento por el segundo y devuelve el cociente y el resto como una tupla):

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

Es posible que tenga que pensar un poco y realizar algunas pruebas para convencerse de que estas funciones son correctas. Una forma de probarlos es verificar que `time_to_int (int_to_time(x)) == x` para muchos valores de `x`. Este es un ejemplo de un control de coherencia.

Una vez que esté convencido de que son correctos, puede usarlos para reescribir `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Esta versión es más corta que la original y más fácil de verificar. Como ejercicio, vuelva a escribir `increment` usando `time_to_int` y `int_to_time`.

De alguna manera, la conversión de la base 60 a la base 10 y viceversa es más difícil que simplemente lidiar con los tiempos. La conversión de base es más abstracta; nuestra intuición para lidiar con los valores de tiempo es mejor.

Pero si tenemos la idea de tratar los tiempos como números base 60 y hacer la inversión de escribir las funciones de conversión (`time_to_int` y `int_to_time`), obtenemos un programa que es más corto, más fácil de leer y depurar, y más confiable.

También es más fácil agregar características más adelante. Por ejemplo, imagine restar dos veces para encontrar la duración entre ellos. El enfoque ingenuo sería implementar la resta con endeudamiento. Usar las funciones de conversión sería más fácil y más probable que sea correcto.

Irónicamente, a veces hacer que un problema sea más difícil (o más general) lo hace más fácil (porque hay menos casos especiales y menos oportunidades de error).

16.5 Depuración

Un objeto Time está bien formado si los valores de `minute` y `second` están entre 0 y 60 (incluido 0 pero no 60) y si `hour` es positivo. `hour` y `minute` deberían ser valores integrales, pero podríamos permitir secondtener una parte de fracción.

Los requisitos como estos se llaman **invariantes** porque siempre deben ser ciertos. Para decirlo de otra manera, si no son ciertas, algo salió mal.

Escribir código para verificar invariantes puede ayudar a detectar errores y encontrar sus causas. Por ejemplo, puede tener una función como la `valid_time` que toma un objeto Time y devuelve `False` si viola un invariante:

```
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
```

Al comienzo de cada función, puede verificar los argumentos para asegurarse de que sean válidos:

```
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)
```

O puede usar una **declaración de afirmación**, que verifica un invariante dado y genera una excepción si falla:

```
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` las declaraciones son útiles porque distinguen el código que trata las condiciones normales del código que busca errores.

16.6 Glosario

prototipo y parche:

> Un plan de desarrollo que implica escribir un borrador de un programa, probar y corregir los errores a medida que se encuentran.

desarrollo diseñado:

> Un plan de desarrollo que implica una visión de alto nivel del problema y más planificación que el desarrollo incremental o el desarrollo de prototipos.

función pura:

> Una función que no modifica ninguno de los objetos que recibe como argumentos. La mayoría de las funciones puras son fructíferas.

modificador:

Una función que cambia uno o más de los objetos que recibe como argumentos. La mayoría de los modificadores son nulos; es decir, vuelven None.

programación funcional:

Un estilo de diseño de programa en el que la mayoría de las funciones son puras.

invariante:

Una condición que siempre debería ser cierta durante la ejecución de un programa.

declaración afirmativa:

Una declaración que verifica una condición y genera una excepción si falla.

16.7 Ejercicios

Los ejemplos de código de este capítulo están disponibles en *http://thinkpython2.com/code/Time1.py*; las soluciones para los ejercicios están disponibles en *http://thinkpython2.com/code/Time1_soln.py*.

Ejercicio 16-1.

Escriba una función llamada `mul_time` que toma un objeto de Tiempo y un número y devuelve un nuevo objeto de Tiempo que contiene el producto del Tiempo original y el número.

Luego use `mul_time` para escribir una función que tome un objeto Time que represente el tiempo de finalización en una carrera, y un número que represente la distancia, y devuelva un objeto Time que represente el ritmo promedio (tiempo por milla).

Ejercicio 16-2.

El `datetime` módulo proporciona `time` objetos que son similares a los objetos de Tiempo en este capítulo, pero proporcionan un amplio conjunto de métodos y operadores. Lea la documentación en *http://docs.python.org/3/library/datetime.html*.

1. Use el módulo `datetime` para escribir un programa que obtenga la fecha actual e imprima el día de la semana.

2. Escriba un programa que tome un cumpleaños como entrada e imprima la edad del usuario y la cantidad de días, horas, minutos y segundos hasta su próximo cumpleaños.

3. Para dos personas nacidas en días diferentes, hay un día en que uno tiene el doble de edad que el otro. Ese es su Día Doble. Escriba un programa que tome dos cumpleaños y calcule su doble día.

4. Para un poco más de desafío, escriba la versión más general que calcula el día en que una persona es n veces mayor que la otra.

Solución: *http://thinkpython2.com/code/double.py*.

Capítulo 17

Clases y métodos

Aunque estamos utilizando algunas de las características orientadas a objetos de Python, los programas de los dos últimos capítulos no están realmente orientados a objetos porque no representan las relaciones entre los tipos definidos por el programador y las funciones que operan en ellos. El siguiente paso es transformar esas funciones en métodos que hagan que las relaciones sean explícitas.

Los ejemplos de código de este capítulo están disponibles en http://thinkpython2.com/code/Time2.py, y las soluciones para los ejercicios están en http://thinkpython2.com/code/Point2_soln.py.

17.1 Características orientadas a objetos

Python es un **lenguaje de programación orientado a objetos**, lo que significa que proporciona características que admiten programación orientada a objetos, que tiene estas características definitorias:

- Los programas incluyen definiciones de clases y métodos.
- La mayor parte del cálculo se expresa en términos de operaciones en objetos.
- Los objetos a menudo representan cosas en el mundo real, y los métodos a menudo se corresponden con las formas en que las cosas en el mundo real interactúan.

Por ejemplo, la clase `Time` definida en el Capítulo 16 corresponde a la forma en que las personas registran la hora del día, y las funciones que definimos corresponden a los tipos de cosas que las personas hacen con los tiempos. Del mismo modo, las clases `Point` y `Rectangle` en el Capítulo 15 corresponden a los conceptos matemáticos de un punto y un rectángulo.

Hasta ahora, no hemos aprovechado las características que proporciona Python para admitir programación orientada a objetos. Estas características no son estrictamente necesarias; la mayoría de ellos proporciona una sintaxis alternativa para cosas que ya hemos hecho. Pero en muchos casos, la alternativa es más concisa y transmite con mayor precisión la estructura del programa.

Por ejemplo, `Time1.py` no existe una conexión obvia entre la definición de clase y las definiciones de función que siguen. Con algún examen, es evidente que cada función toma al menos un objeto `Time` como argumento.

Esta observación es la motivación de los **métodos**; un método es una función que está asociada con una clase particular. Hemos visto métodos para cadenas, listas, diccionarios y tuplas. En este capítulo, definiremos métodos para tipos definidos por programador.

Los métodos son semánticamente lo mismo que las funciones, pero hay dos diferencias sintácticas:

- Los métodos se definen dentro de una definición de clase para hacer explícita la relación entre la clase y el método.
- La sintaxis para invocar un método es diferente de la sintaxis para llamar a una función.

En las siguientes secciones, tomaremos las funciones de los dos capítulos anteriores y las transformaremos en métodos. Esta transformación es puramente mecánica; puedes hacerlo siguiendo una secuencia de pasos. Si te sientes cómodo convirtiendo de un formulario a otro, podrás elegir la mejor forma para lo que sea que estés haciendo.

17.2 Imprimir objetos

En el Capítulo 16, definimos una clase nombrada `Time` y en "Time", escribiste una función llamada `print_time`:

```
class Time:
    """Represents the time of day."""


def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Para llamar a esta función, debe pasar un objeto `Time` como argumento:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

Para hacer print_time un método, todo lo que tenemos que hacer es mover la definición de la función dentro de la definición de la clase. Observe el cambio en la sangría.

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

Ahora hay dos formas de llamar print_time. La primera (y menos común) forma es utilizar la sintaxis de la función:

```
>>> Time.print_time (inicio)
09:45:00
```

En este uso de la notación de punto, `Time` es el nombre de la clase, y `print_time` es el nombre del método. `start` se pasa como un parámetro.

La segunda forma (y más concisa) es usar la sintaxis del método:

```
>>> start.print_time ()
09:45:00
```

En este uso de la notación de punto, `print_time` es el nombre del método (de nuevo), y `start` es el objeto en el que se invoca el método, que se llama **sujeto**. Así como el sujeto de una oración es de lo que se trata la oración, el tema de la invocación de un método es de qué se trata el método.

Dentro del método, el sujeto se asigna al primer parámetro, por lo que en este caso startse le asigna time.

Por convención, se llama al primer parámetro de un método **self**, por lo que sería más común escribir `print_time` de esta manera:

```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

El motivo de esta convención es una metáfora implícita:

- La sintaxis para una llamada de función, `print_time(start)` sugiere que la función es el agente activo. Dice algo así como, " `print_time` ¡Oye! Aquí hay un objeto para que imprimas ".

- En la programación orientada a objetos, los objetos son los agentes activos. Una invocación a un método como `start.print_time()` "start ¡Hey ! Por favor, imprima usted mismo".

Este cambio de perspectiva puede ser más cortés, pero no es obvio que sea útil. En los ejemplos que hemos visto hasta ahora, puede que no lo sea. Pero a veces el cambio de responsabilidad de las funciones a los objetos hace posible escribir funciones (o métodos) más versátiles, y hace que sea más fácil mantener y reutilizar el código.

Como ejercicio, reescriba `time_to_int` (desde "Prototipos versus Planificación") como un método. También podría estar tentado a reescribir `int_to_time` como método, pero eso realmente no tiene sentido porque no habría ningún objeto para invocarlo.

17.3 Otro ejemplo

Aquí hay una versión de `increment` (de "Modificadores") reescrita como método:

```
# inside class Time:


    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Esta versión asume que `time_to_int` está escrito como un método. Además, tenga en cuenta que es una función pura, no un modificador.

Así es como invocarías `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
```

```
10:07:17
```

El sujeto, `start`, se le asigna al primer parámetro, `self`. El argumento, `1337`, se le asigna al segundo parámetro, seconds.

Este mecanismo puede ser confuso, especialmente si comete un error. Por ejemplo, si invoca `increment` con dos argumentos, obtiene:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were given
```

El mensaje de error inicialmente es confuso, porque solo hay dos argumentos entre paréntesis. Pero el tema también se considera un argumento, por lo que todos juntos son tres.

Por cierto, un **argumento posicional** es un argumento que no tiene un nombre de parámetro; es decir, no es un argumento de palabra clave. En esta función, llame a:

```
sketch(parrot, cage, dead=True)
```

`parrot` y `cage` son posicionales, y `dead` es un argumento clave.

17.4 Un ejemplo más complicado

Reescribir `is_after` (desde "Time") es un poco más complicado porque toma dos objetos de Tiempo como parámetros. En este caso, es convencional nombrar el primer parámetro `self` y el segundo parámetro `other`:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Para utilizar este método, debe invocarlo en un objeto y pasar el otro como argumento:

```
>>> end.is_after(start)
True
```

Una cosa buena de esta sintaxis es que casi se lee en inglés: "end is after start?"

17.5 El método init

El método init (abreviatura de " initialization") es un método especial que se invoca cuando se crea una instancia de un objeto. Su nombre completo es `__init__` (dos caracteres de subrayado, seguidos de init, y luego otros dos guiones bajos). Un método init para la clase `Time` podría verse así:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
```

```
        self.second = second
```

Es común que los parámetros __init__ tengan los mismos nombres que los atributos. La declaración

```
        self.hour = hour
```

almacena el valor del parámetro `hour` como un atributo de `self`.

Los parámetros son opcionales, por lo que si llama `Time` sin argumentos, obtendrá los valores predeterminados:

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

Si proporciona un argumento, anula `hour`:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

Si proporciona dos argumentos, anulan `hour` y `minute`:

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

Y si proporciona tres argumentos, anulan los tres valores predeterminados.

Como ejercicio, escriba un método init para la clase `Point` que toma x y y como parámetros opcionales y los asigna a los atributos correspondientes.

17.6 El método __str__

`__str__` es un método especial, como `__init__`, que se supone que devuelve una representación de cadena de un objeto.

Por ejemplo, aquí hay un método `str` para objetos de tiempo:

```
# inside class Time:


    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Cuando eres un objeto `print`, Python invoca el método `str`:

```
>>> time = Time(9, 45)
>>> print(time)
```

```
09:45:00
```

Cuando escribo una nueva clase, casi siempre empiezo escribiendo `__init__`, lo que hace que sea más fácil crear instancias de objetos, y es útil para la depuración.

Como ejercicio, escribe un método `str` para la clase `Point`. Crea un objeto `Point` e imprímelo.

17.7 Sobrecarga del operador

Al definir otros métodos especiales, puede especificar el comportamiento de los operadores en los tipos definidos por el programador. Por ejemplo, si define un método llamado `__add__` para la clase `Time`, puede usar el operador `+` en objetos `Time`.

Aquí está cómo se vería la definición:

```
# inside class Time:


    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

Y aquí es cómo puedes usarlo:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

Cuando aplica el operador `+` a objetos `Time`, Python invoca `__add__`. Cuando imprimes el resultado, Python invoca `__str__`. ¡Así que están sucediendo muchas cosas detrás de escena!

Cambiar el comportamiento de un operador para que funcione con los tipos definidos por el programador se denomina sobrecarga del operador. Para cada operador en Python hay un método especial correspondiente, como `__add__`. Para obtener más información, consulte *http://docs.python.org/3/reference/datamodel.html#specialnames*.

Como ejercicio, escribe un método `add` para la clase Punto.

17.8 Despacho basado en el tipo

En la sección anterior agregamos dos objetos Time, pero también es posible que desee agregar un número entero a un objeto Time. La siguiente es una versión de `__add__` que comprueba el tipo de `other` e invoca cualquiera `add_time` o `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)
```

La función incorporada `isinstance` toma un valor y un objeto de clase, y retorna `True` si el valor es una instancia de la clase.

Si otheres un objeto Time, `__add__` invoca `add_time`. De lo contrario, se supone que el parámetro es un número e invoca `increment`. Esta operación se denomina **despacho basado en tipo** porque distribuye el cálculo a diferentes métodos en función del tipo de los argumentos.

Aquí hay ejemplos que usan el operador + con diferentes tipos:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Desafortunadamente, esta implementación de adición no es conmutativa. Si el entero es el primer operando, obtienes

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

El problema es que, en lugar de pedirle al objeto Time que agregue un entero, Python le pide a un entero que agregue un objeto Time, y no sabe cómo hacerlo. Pero hay una solución inteligente para este problema: el método especial `__radd__`, que significa "right-side add". Este método se invoca cuando aparece un objeto Time en el lado derecho del operador +. Aquí está la definición:

```
# inside class Time:


    def __radd __ (self, other):
        return self .__ add __ (other)
```

Y así es como se usa:

```
>>> print(1337 + start)
10:07:17
```

Como ejercicio, escriba un método `add` para Puntos que funcione con un objeto `Point` o una tupla:

- Si el segundo operando es un punto, el método debe devolver un nuevo punto cuya coordenada x es la suma de las coordenadas x de los operandos, y lo mismo ocurre con las coordenadas y.
- Si el segundo operando es una tupla, el método debe agregar el primer elemento de la tupla a la coordenada x y el segundo elemento a la coordenada y, y devolver un nuevo Punto con el resultado.

17.9 Polimorfismo

El despacho basado en el tipo es útil cuando es necesario, pero (afortunadamente) no siempre es necesario. A menudo puede evitarlo escribiendo funciones que funcionen correctamente para argumentos con diferentes tipos.

Muchas de las funciones que escribimos para cadenas también funcionan para otros tipos de secuencias. Por ejemplo, en "Diccionario como una colección de contadores" solíamos contar con `histogram` el número de veces que cada letra aparece en una palabra:

```
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
```

Esta función también funciona para listas, tuplas e incluso diccionarios, siempre que los elementos de los elementos `s` sean aptos para que puedan utilizarse como claves en `d`:

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Las funciones que funcionan con varios tipos se llaman **polimórficas**. El polimorfismo puede facilitar la reutilización del código. Por ejemplo, la función incorporada `sum`, que agrega los elementos de una secuencia, funciona siempre que los elementos de la secuencia admitan la adición.

Como los objetos Time proporcionan un método `add`, trabajan con `sum`:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
```

```
>>> print(total)
23:01:00
```

En general, si todas las operaciones dentro de una función funcionan con un tipo dado, la función funciona con ese tipo.

El mejor tipo de polimorfismo es el tipo involuntario, donde descubres que una función que ya escribiste puede aplicarse a un tipo que nunca planificaste.

17.10 Interfaz e Implementación

Uno de los objetivos del diseño orientado a objetos es hacer que el software sea más fácil de mantener, lo que significa que puede mantener el programa funcionando cuando otras partes del sistema cambian y modificar el programa para cumplir con los nuevos requisitos.

Un principio de diseño que ayuda a lograr ese objetivo es mantener las interfaces separadas de las implementaciones. Para los objetos, eso significa que los métodos que proporciona una clase no deben depender de cómo se representan los atributos.

Por ejemplo, en este capítulo desarrollamos una clase que representa una hora del día. Métodos proporcionados por esta clase incluyen `time_to_int`, `is_after`, y `add_time`.

Podríamos implementar esos métodos de varias maneras. Los detalles de la implementación dependen de cómo representemos el tiempo. En este capítulo, los atributos de un objeto Time son `hour`, `minute` y `second`.

Como alternativa, podríamos reemplazar estos atributos con un solo entero que represente el número de segundos desde la medianoche. Esta implementación haría que algunos métodos, como `is_after`, sean más fáciles de escribir, pero hace que otros métodos sean más difíciles.

Después de implementar una nueva clase, es posible que descubra una mejor implementación. Si otras partes del programa están utilizando su clase, puede llevar mucho tiempo y ser propenso a errores cambiar la interfaz.

Pero si diseña la interfaz con cuidado, puede cambiar la implementación sin cambiar la interfaz, lo que significa que otras partes del programa no tienen que cambiar.

17.11 Depuración

Es legal agregar atributos a los objetos en cualquier punto de la ejecución de un programa, pero si tiene objetos del mismo tipo que no tienen los mismos atributos, es fácil cometer errores. Se considera una buena idea inicializar todos los atributos de un objeto en el método init.

Si no está seguro de si un objeto tiene un atributo en particular, puede usar la función incorporada `hasattr` (consulte "Depuración").

Otra forma de acceder a los atributos es la función integrada `vars`, que toma un objeto y devuelve un diccionario que correlaciona desde los nombres de los atributos (como cadenas) a sus valores:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

Para fines de depuración, puede resultarle útil mantener esta función a mano:

```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

`print_attributes` atraviesa el diccionario e imprime cada nombre de atributo y su valor correspondiente.

La función incorporada `getattr` toma un objeto y un nombre de atributo (como una cadena) y devuelve el valor del atributo.

17.12 Glosario

lenguaje orientado a objetos:

> Un lenguaje que proporciona funciones, como tipos y métodos definidos por programador, que facilitan la programación orientada a objetos.

programación orientada a objetos:

> Un estilo de programación en el que los datos y las operaciones que lo manipulan se organizan en clases y métodos.

método:

> Una función que se define dentro de una definición de clase y se invoca en instancias de esa clase.

tema:

> El objeto en el que se invoca un método.

argumento posicional:

> Un argumento que no incluye un nombre de parámetro, por lo que no es un argumento de palabra clave.

sobrecarga del operador:

> Cambiando el comportamiento de un operador como + para que funcione con un tipo definido por el programador.

despacho basado en tipo:

> Un patrón de programación que verifica el tipo de un operando e invoca diferentes funciones para diferentes tipos.

polimórfico:

> Perteneciente a una función que puede funcionar con más de un tipo.

ocultación de información:

> El principio de que la interfaz proporcionada por un objeto no debe depender de su implementación, en particular la representación de sus atributos.

17.13 Ejercicios

Ejercicio 17-1.

Descargue el código de este capítulo de *http://thinkpython2.com/code/Time2.py*. Cambia los atributos de a ser un entero único que representa los segundos desde la medianoche. Luego modifique los métodos (y la función) para trabajar con la nueva implementación. No debería tener que modificar el código de prueba en. Cuando haya terminado, la salida debería ser la misma que antes `Timeint_to_timemain`.

Solución: *http://thinkpython2.com/code/Time2_soln.py*.

Ejercicio 17-2.

Este ejercicio es una historia de advertencia sobre uno de los errores más comunes y difíciles de encontrar en Python. Escriba una definición para una clase nombrada `Kangaroo` con los siguientes métodos:

1. Un método `__init__` que inicializa un atributo nombrado `pouch_contents` a una lista vacía.

2. Un método llamado `put_in_pouch` que toma un objeto de cualquier tipo y lo agrega a `pouch_contents`.

3. Un método `__str__` que devuelve una representación de cadena del objeto `Kangaroo` y el contenido de la bolsa.

Pruebe su código creando dos objetos `Kangaroo`, asignándolos a las variables nombradas `kanga` y `roo`, luego, agregando `roo` el contenido de `kanga` la valija.

Descargue *http://thinkpython2.com/code/BadKangaroo.py*. Contiene una solución al problema anterior con un error grande y desagradable. Encuentra y arregla el error.

Si te quedas atascado, puedes descargar *http://thinkpython2.com/code/GoodKangaroo.py*, que explica el problema y muestra una solución.

Capítulo 18

Herencia

La característica de lenguaje más comúnmente asociada con la programación orientada a objetos es la **herencia**. La herencia es la capacidad de definir una nueva clase que es una versión modificada de una clase existente. En este capítulo, demuestro herencia usando clases que representan naipes, mazos de cartas y manos de póker.

Si no juegas al póker, puedes leer sobre él en *http://en.wikipedia.org/wiki/Poker*, pero no tienes que hacerlo; Te diré lo que necesitas saber para los ejercicios.

Los ejemplos de código de este capítulo están disponibles en *http://thinkpython2.com/code/Card.py*.

18.1 Objeto de tarjeta

Hay 52 cartas en una baraja, cada una de las cuales pertenece a 1 de 4 palos y 1 de 13 rangos. Los trajes son Picas, Corazones, Diamantes y Tréboles (en orden descendente en el puente). Los rangos son Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen y King. Dependiendo del juego que estés jugando, un As puede ser más alto que King o menor que 2.

Si queremos definir un nuevo objeto para representar un naipe, es obvio cuáles deberían ser los atributos: `rank` y `suit`. No es tan obvio qué tipo deberían ser los atributos. Una posibilidad es usar cadenas que contengan palabras como `'Spade'` para trajes y `'Queen'` rangos. Un problema con esta implementación es que no sería fácil comparar las tarjetas para ver cuál tenía un rango o palo más alto.

Una alternativa es usar enteros para **codificar** los rangos y los trajes. En este contexto, "codificar" significa que vamos a definir un mapeo entre números y palos, o entre números y rangos. Este tipo de codificación no pretende ser un secreto (eso sería "encriptación").

Por ejemplo, esta tabla muestra los trajes y los códigos enteros correspondientes:

Espadas ↦ 3

Copas ↦ 2

Diamantes ↦ 1

Clubs ↦ 0

Este código hace que sea fácil comparar tarjetas; Debido a que los trajes más altos se asignan a números más altos, podemos comparar los trajes comparando sus códigos.

El mapeo de rangos es bastante obvio; cada uno de los rangos numéricos se correlaciona con el entero correspondiente, y para las cartas de cara:

Jack ↦ 11

Reina ↦ 12

Rey ↦ 13

Estoy utilizando el símbolo ↦ para dejar claro que estas asignaciones no son parte del programa Python. Son parte del diseño del programa, pero no aparecen explícitamente en el código.

La definición de clase `Card` se ve así:

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

Como de costumbre, el método init toma un parámetro opcional para cada atributo. La carta predeterminada es el 2 de Tréboles.

Para crear una Tarjeta, llame `Card` con el palo y el rango de la tarjeta que desea:

```
queen_of_diamonds = Card(1, 12)
```

18.2 Atributos de clase

Para imprimir los objetos de la Carta de forma que las personas puedan leer fácilmente, necesitamos una asignación de los códigos enteros a los rangos y palos correspondientes. Una forma natural de hacerlo es con listas de cadenas. Asignamos estas listas a los **atributos de clase**:

```
# 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])
```

Las variables como `suit_names` y `rank_names`, que se definen dentro de una clase pero fuera de cualquier método, se llaman atributos de clase porque están asociados con el objeto de la clase `Card`.

Este término los distingue de las variables como `suit` y `rank`, que se llaman atributos de instancia porque están asociados con una instancia particular.

Se accede a ambos tipos de atributos usando notación de puntos. Por ejemplo, in `__str__`, `self` es un objeto Card, y `self.rank` es su rango. Del mismo modo, `Card` es un objeto de clase, y `Card.rank_names` es una lista de cadenas asociadas con la clase.

Cada carta tiene su propia `suit` y `rank`, pero solo hay una copia de `suit_names` y `rank_names`.

Poniendo todo junto, la expresión `Card.rank_names[self.rank]` significa "usar el atributo `rank` del objeto `self` como un índice en la lista `rank_names` de la clase `Card`, y seleccionar la cadena apropiada".

El primer elemento de `rank_names` es `None` porque no hay ninguna carta con rango cero. Al incluir `None` como un guardián de lugar, obtenemos un mapeo con la propiedad agradable que el índice 2 asigna a la cadena `'2'`, y así sucesivamente. Para evitar esta modificación, podríamos haber usado un diccionario en lugar de una lista.

Con los métodos que tenemos hasta ahora, podemos crear e imprimir tarjetas:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

La figura 18-1 es un diagrama del objeto `Card` de clase y una instancia de tarjeta. `Card` es un objeto de clase; su tipo es `type`. `card1` es una instancia de `Card`, por lo que su tipo es Card. Para ahorrar espacio, no dibujé el contenido de `suit_names` y `rank_names`.

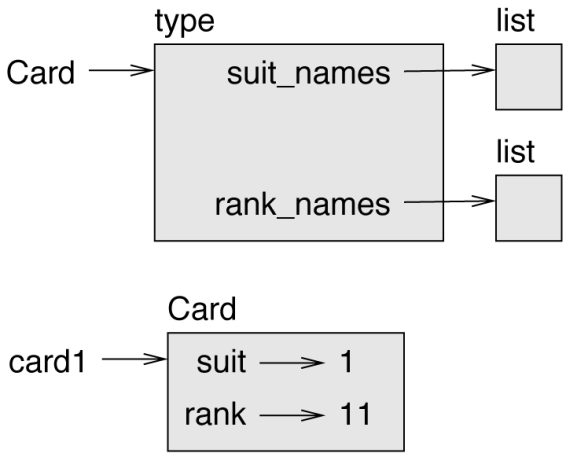

Figura 18-1. Diagrama de objeto.

18.3 Comparando tarjetas

Para los tipos incorporados, hay operadores relacionales (<, >, ==, etc.) que comparan los valores y determinan cuando uno es mayor que, menor que, o igual a otro. Para los tipos definidos por el programador, podemos anular el comportamiento de los operadores incorporados al proporcionar un método denominado `__lt__`, que significa "menor que".

`__lt__` toma dos parámetros, `self` y `other`, `True` si `self` es estrictamente menor que `other`.

El orden correcto para las cartas no es obvio. Por ejemplo, ¿cuál es mejor, el 3 de Tréboles o el 2 de Diamantes? Uno tiene un rango más alto, pero el otro tiene un palo más alto. Para comparar cartas, debes decidir si el rango o el palo es más importante.

La respuesta puede depender del juego que estés jugando, pero para hacerlo simple, haremos la elección arbitraria de ese palo más importante, de modo que todas las Picas superarán a todos los Diamantes, y así sucesivamente.

Con eso decidido, podemos escribir `__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
```

Puede escribir esto de manera más concisa usando la comparación de tuplas:

```
# inside class Card:

    def __lt__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return t1 < t2
```

Como ejercicio, escribe un método `__lt__` para objetos de Tiempo. Puede usar la comparación de tuplas, pero también podría considerar la comparación de enteros.

18.4 Mazos

Ahora que tenemos tarjetas, el siguiente paso es definir mazos. Como un mazo está compuesto de cartas, es natural que cada mazo contenga una lista de cartas como atributo.

La siguiente es una definición de clase para `Deck`. El método init crea el atributo `cards` y genera el conjunto estándar de 52 cartas:

```
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)
```

La forma más fácil de llenar el mazo es con un bucle anidado. El bucle externo enumera los palos de 0 a 3. El bucle interno enumera los rangos de 1 a 13. Cada iteración crea una nueva carta con el palo y rango actual, y lo agrega a `self.cards`.

18.5 Imprimir el mazo

Aquí hay un método `__str__` para Deck:

```
#inside class Deck:


    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Este método demuestra una forma eficiente de acumular una cadena grande: crear una lista de cadenas y luego usar el método de cadena `join`. La función incorporada `str` invoca el método `__str__` en cada tarjeta y devuelve la representación de cadena.

Como invocamos `join` en un carácter de nueva línea, las tarjetas están separadas por líneas nuevas. Así es como se ve el resultado:

```
>>> 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
```

Aunque el resultado aparece en 52 líneas, es una cadena larga que contiene líneas nuevas.

18.6 Agregar, Eliminar, Mezclar y Ordenar

Para repartir cartas, nos gustaría un método que elimine una carta del mazo y la devuelva. El método de lista pop proporciona una forma conveniente de hacerlo:

```
#inside class Deck:


    def pop_card(self):
        return self.cards.pop()
```

Como pop elimina la última carta de la lista, estamos tratando desde la parte inferior del mazo.

Para agregar una tarjeta, podemos usar el método de lista `append`:

```
#inside class Deck:


    def add_card(self, card):
        self.cards.append(card)
```

Un método como este que usa otro método sin hacer mucho trabajo a veces se llama **chapa**. La metáfora proviene de la carpintería, donde una chapa es una fina capa de madera de buena calidad pegada a la superficie de una pieza de madera más barata para mejorar la apariencia.

En este caso, `add_card` es un método "delgado" que expresa una operación de lista en términos apropiados para mazos. Mejora la apariencia o interfaz de la implementación.

Como otro ejemplo, podemos escribir un método de plataforma llamado `shuffle` usando la función `shuffle` del módulo `random`:

```
# inside class Deck:


    def shuffle(self):
        random.shuffle(self.cards)
```

No te olvides de importar `random`.

Como ejercicio, escriba un método de cubierta llamado `sort` que usa el método de lista `sort` para ordenar las tarjetas en `Deck`. `sort` usa el método `__lt__` que definimos para determinar el orden.

18.7 Herencia

La herencia es la capacidad de definir una nueva clase que es una versión modificada de una clase existente. Como ejemplo, digamos que queremos que una clase represente una "mano", es decir, las cartas que posee un jugador. Una mano es similar a una baraja: ambas están formadas por una colección de cartas, y ambas requieren operaciones como agregar y eliminar cartas.

Una mano también es diferente de una baraja; hay operaciones que queremos para manos que no tienen sentido para un mazo. Por ejemplo, en el poker podemos comparar dos manos para ver cuál gana. En bridge, podemos calcular un puntaje de una mano para hacer una oferta.

Esta relación entre clases-similar, pero diferente-se presta a la herencia. Para definir una nueva clase que hereda de una clase existente, coloque el nombre de la clase existente entre paréntesis:

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

Esta definición indica que `Hand` hereda de `Deck`; eso significa que podemos usar métodos como `pop_card` y `add_card` para `Hand` así como para `Decks`.

Cuando una clase nueva hereda de una existente, la existente se llama **padre** y la nueva clase se llama **hijo**.

En este ejemplo, Handhereda `__init__` de `Deck`, pero realmente no hace lo que queremos: en lugar de poblar la mano con 52 nuevas tarjetas, el método init para Hands debe inicializarse `cards` con una lista vacía.

Si proporcionamos un método init en la clase `Hand`, anula el de la clase `Deck`:

```
# inside class Hand:


    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

Cuando creas una Mano, Python invoca este método de inicio, no el que está en `Deck`.

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

Los otros métodos se heredan de `Deck`, entonces podemos usar `pop_card` y `add_card` tratar una carta:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

Un siguiente paso natural es encapsular este código en un método llamado `move_cards`:

```
#inside class Deck:


    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` toma dos argumentos, un objeto Mano y la cantidad de cartas para repartir. Modifica ambos `self` y `hand`, y regresa `None`.

En algunos juegos, las cartas se mueven de una mano a otra, o de una mano al mazo. Puede utilizar `move_cards` para cualquiera de estas operaciones: `self` puede ser una baraja o una mano, y `hand`, a pesar del nombre, también puede ser a `Deck`.

La herencia es una característica útil. Algunos programas que serían repetitivos sin herencia se pueden escribir con mayor elegancia. La herencia puede facilitar la reutilización del código, ya que puede personalizar el comportamiento de las clases principales sin tener que modificarlas. En algunos casos, la estructura de herencia refleja la estructura natural del problema, lo que hace que el diseño sea más fácil de entender.

Por otro lado, la herencia puede hacer que los programas sean difíciles de leer. Cuando se invoca un método, a veces no está claro dónde encontrar su definición. El código relevante puede extenderse a través de varios módulos. Además, muchas de las cosas que se pueden hacer usando la herencia se pueden hacer tan bien o mejor sin eso.

18.8 Diagramas de clase

Hasta ahora hemos visto diagramas de pila, que muestran el estado de un programa, y diagramas de objetos, que muestran los atributos de un objeto y sus valores. Estos diagramas representan una instantánea en la ejecución de un programa, por lo que cambian a medida que se ejecuta el programa.

También son muy detallados; para algunos propósitos, demasiado detallado. Un diagrama de clase es una representación más abstracta de la estructura de un programa. En lugar de mostrar objetos individuales, muestra las clases y las relaciones entre ellos.

Hay varios tipos de relación entre clases:

- Los objetos en una clase pueden contener referencias a objetos en otra clase. Por ejemplo, cada Rectángulo contiene una referencia a un Punto, y cada Deck contiene referencias a muchas Tarjetas. Este tipo de relación se llama **HAS-A**, como en, "un Rectángulo tiene un Punto".

- Una clase puede heredar de otra. Esta relación se llama **IS-A**, como en, "una Mano es una especie de Baraja".

- Una clase puede depender de otra en el sentido de que los objetos en una clase toman los objetos en la segunda clase como parámetros, o usan objetos en la segunda clase como parte de un cálculo. Este tipo de relación se llama **dependencia**.

Un **diagrama de clase** es una representación gráfica de estas relaciones. Por ejemplo, la Figura 18-2 muestra las relaciones entre `Card`, `Deck` y `Hand`.

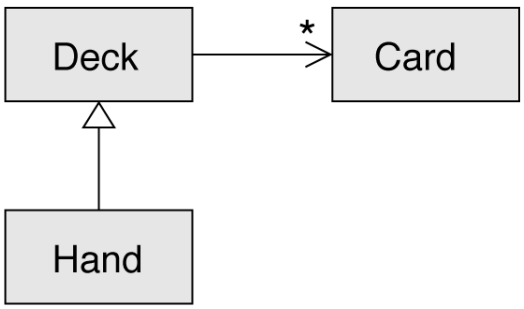

Figura 18-2. Diagrama de clase.

La flecha con una cabeza triangular hueca representa una relación IS-A; en este caso, indica que Hand hereda de Deck.

La punta de flecha estándar representa una relación HAS-A; en este caso, un Deck tiene referencias a objetos Card.

La estrella (*) cerca de la punta de flecha es una **multiplicidad**; indica cuántas cartas tiene una baraja. Una multiplicidad puede ser un número simple 52, como un rango 5..7 o una estrella, que indica que un Deck puede tener cualquier cantidad de Cartas.

No hay dependencias en este diagrama. Normalmente se mostrarían con una flecha discontinua. O si hay muchas dependencias, a veces se omiten.

Un diagrama más detallado podría mostrar que un Deck en realidad contiene una lista de Tarjetas, pero los tipos incorporados como list y dict generalmente no están incluidos en los diagramas de clases.

18.9 Encapsulación de datos

Los capítulos anteriores demuestran un plan de desarrollo que podríamos llamar "diseño orientado a objetos". Identificamos objetos que necesitábamos, como `Point`, `Rectangle` y `Time`, definimos clases para representarlos. En cada caso hay una correspondencia obvia entre el objeto y alguna entidad en el mundo real (o al menos un mundo matemático).

Pero a veces es menos obvio qué objetos necesitas y cómo deberían interactuar. En ese caso, necesita un plan de desarrollo diferente. De la misma manera que descubrimos interfaces de funciones por encapsulación y generalización, podemos descubrir interfaces de clase por **encapsulación de datos**.

El análisis de Markov, del "Análisis de Markov" , proporciona un buen ejemplo. Si descarga mi código de *http://thinkpython2.com/code/markov.py*, verá que usa dos variables globales y que se leen y escriben desde varias funciones como `suffix_map` y `prefix`.

```
suffix_map = {}
prefix = ()
```

Debido a que estas variables son globales, solo podemos ejecutar un análisis a la vez. Si leemos dos textos, sus prefijos y sufijos se agregarán a las mismas estructuras de datos (lo que lo convierte en un texto generado interesante).

Para ejecutar análisis múltiples y mantenerlos separados, podemos encapsular el estado de cada análisis en un objeto. Esto es lo que parece:

```
class Markov:


    def __init__(self):
        self.suffix_map = {}
        self.prefix = ()
```

A continuación, transformamos las funciones en métodos. Por ejemplo, aquí está `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)
```

Transformar un programa como este -cambiar el diseño sin cambiar el comportamiento- es otro ejemplo de refactorización (ver "Refactorización").

Este ejemplo sugiere un plan de desarrollo para diseñar objetos y métodos:

1. Comience escribiendo funciones que lean y escriban variables globales (cuando sea necesario).

2. Una vez que tenga el programa funcionando, busque asociaciones entre las variables globales y las funciones que las usan.

3. Encapsula las variables relacionadas como atributos de un objeto.

4. Transforma las funciones asociadas en métodos de la nueva clase.

Como ejercicio, descargue mi código de Markov de *http://thinkpython2.com/code/markov.py* y siga los pasos descritos anteriormente para encapsular las variables globales como atributos de una nueva clase llamada `Markov`.

Solución: *http://thinkpython2.com/code/Markov.py* (tenga en cuenta la mayúscula M).

18.10 Depuración

La herencia puede dificultar la depuración porque cuando invocas un método en un objeto, puede ser difícil averiguar qué método se invocará.

Supongamos que está escribiendo una función que funciona con objetos Mano. Que le gustaría que para trabajar con todo tipo de manos, como PokerHands, BridgeHands, etc. Si se invoca un método como `shuffle`, podría obtener la definida en `Deck`, pero si cualquiera de las subclases reemplazar este método, podrás obtener la versión en lugar. Este comportamiento suele ser algo bueno, pero puede ser confuso.

Cada vez que no esté seguro acerca del flujo de ejecución a través de su programa, la solución más simple es agregar instrucciones de impresión al comienzo de los métodos relevantes. Si `Deck.shuffle` imprime un mensaje que dice algo así `Running Deck.shuffle`, a medida que el programa se ejecuta, rastrea el flujo de ejecución.

Como alternativa, puede usar esta función, que toma un objeto y un nombre de método (como una cadena) y devuelve la clase que proporciona la definición del método:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Aquí hay un ejemplo:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
```

```
<class 'Card.Deck'>
```

Entonces el método `shuffle` para esta Mano es el que está adentro `Deck`.

`find_defining_class` usa el método `mro` para obtener la lista de objetos de clase (tipos) en los que se buscarán métodos. "MRO" significa "orden de resolución de método", que es la secuencia de clases que Python busca para "resolver" un nombre de método.

Aquí hay una sugerencia de diseño: cuando anula un método, la interfaz del nuevo método debe ser la misma que la anterior. Debería tomar los mismos parámetros, devolver el mismo tipo y obedecer las mismas condiciones previas y posteriores. Si sigue esta regla, encontrará que cualquier función diseñada para funcionar con una instancia de una clase principal, como una cubierta, también funcionará con instancias de clases secundarias como Hand y PokerHand.

Si viola esta regla, que se llama el "principio de sustitución Liskov", su código colapsará como (lo siento) un castillo de naipes.

18.11 Glosario

codificar:

> Para representar un conjunto de valores usando otro conjunto de valores mediante la construcción de un mapeo entre ellos.

atributo de clase:

> Un atributo asociado con un objeto de clase. Los atributos de clase se definen dentro de una definición de clase pero fuera de cualquier método.

atributo de instancia:

> Un atributo asociado con una instancia de una clase.

chapa:

> Un método o función que proporciona una interfaz diferente a otra función sin hacer mucho cálculo.

herencia:

> La capacidad de definir una nueva clase que es una versión modificada de una clase previamente definida.

clase de padres:

> La clase de la cual una clase hija hereda.

clase de hijos:

> Una nueva clase creada al heredar de una clase existente; también llamado una "subclase".

Relación IS-A:

> Una relación entre una clase infantil y su clase principal.

Relación HAS-A:

Una relación entre dos clases donde las instancias de una clase contienen referencias a instancias de la otra.

dependencia:

Una relación entre dos clases donde las instancias de una clase usan instancias de la otra clase, pero no las almacenan como atributos.

diagrama de clase:

Un diagrama que muestra las clases en un programa y las relaciones entre ellos.

multiplicidad:

Una notación en un diagrama de clase que muestra, para una relación HAS-A, cuántas referencias hay a instancias de otra clase.

encapsulación de datos:

Un plan de desarrollo de programa que involucra un prototipo usando variables globales y una versión final que convierte las variables globales en atributos de instancia.

18.12 Ejercicios

Ejercicio 18-1.

Para el siguiente programa, dibuje un diagrama de clase UML que muestre estas clases y las relaciones entre ellas.

```
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.

Escribe un método llamado Deck `deal_hands` que toma dos parámetros: el número de manos y el número de cartas por mano. Debería crear el número apropiado de objetos de Mano, repartir el número apropiado de cartas por mano y devolver una lista de Manos.

Ejercicio 18-3.

Las siguientes son las manos posibles en el póker, en orden creciente de valor y decreciente orden de probabilidad:

par:

> Dos cartas con el mismo rango.

dos pares:

> Dos pares de cartas con el mismo rango.

tres de un tipo:

> Tres cartas con el mismo rango.

Derecho:

> Cinco cartas con rangos en secuencia (los ases pueden ser altos o bajos, por lo que Ace-2-3-4-5es una escalera y lo es `10-Jack-Queen-King-Ace`, pero `Queen-King-Ace-2-3` no lo es).

enjuagar:

> Cinco cartas con el mismo palo.

casa llena:

> Tres cartas con un rango, dos cartas con otro.

Cuatro de un tipo:

> Cuatro cartas con el mismo rango.

escalera recta:

> Cinco cartas en secuencia (como se define arriba) y con el mismo palo.

El objetivo de estos ejercicios es estimar la probabilidad de dibujar estas diversas manos.

1. Descargue los siguientes archivos de *http://thinkpython2.com/code*:

   `Card.py`:

   Una versión completa de las clases `Card`, `Deck` y `Hand` en este capítulo.

   `PokerHand.py`:

   Una implementación incompleta de una clase que representa una mano de póquer, y algún código que la prueba.

2. Si corres `PokerHand.py`, reparte siete manos de poker de 7 cartas y comprueba si alguno de ellos contiene un color. Lea este código cuidadosamente antes de continuar.

3. Añadir a métodos `PokerHand.py` llamado `has_pair`, `has_twopair` etc. que devuelven verdadero o falso dependiendo de si o no la mano satisface los criterios pertinentes. Su código debería funcionar correctamente para "manos" que contengan cualquier número de tarjetas (aunque 5 y 7 son los tamaños más comunes).

4. Escriba un método llamado `classify` que calcule la clasificación de mayor valor para una mano y establezca el atributo `label` en consecuencia. Por ejemplo, una mano de 7 cartas puede contener un color y un par; debe etiquetarse como "flush".

5. Cuando esté convencido de que sus métodos de clasificación funcionan, el próximo paso es estimar las probabilidades de las distintas manos. Escribe una función en la `PokerHand.py` que baraje un mazo de cartas, lo divida en manos, clasifique las manos y cuente el número de veces que aparecen varias clasificaciones.

6. Imprime una tabla de las clasificaciones y sus probabilidades. Ejecute su programa con un número mayor y mayor de manos hasta que los valores de salida converjan con un grado razonable de precisión. Compare sus resultados con los valores en *http://en.wikipedia.org/wiki/Hand_rankings*.

Solución: *http://thinkpython2.com/code/PokerHandSoln.py*.

Capítulo 19

Los extras

Uno de mis objetivos para este libro ha sido enseñarte el menor Python posible. Cuando había dos formas de hacer algo, elegí uno y evité mencionar el otro. O a veces pongo el segundo en un ejercicio.

Ahora quiero volver para algunas de las cosas buenas que quedaron atrás. Python proporciona una serie de características que no son realmente necesarias, puede escribir un buen código sin ellas, pero con ellas a veces puede escribir un código que sea más conciso, legible o eficiente, y a veces las tres.

19.1 Expresiones condicionales

Vimos declaraciones condicionales en "Ejecución condicional". Los enunciados condicionales a menudo se utilizan para elegir uno de dos valores; por ejemplo:

```
si x> 0:
    y = math.log (x)
más:
    y = float ('nan')
```

Esta declaración verifica si `x` es positivo. Si es así, calcula `math.log`. Si no, `math.log` plantearía a `ValueError`. Para evitar detener el programa, generamos un "NaN", que es un valor especial de coma flotante que representa "No es un número".

Podemos escribir esta declaración de manera más concisa usando una **expresión condicional**:

```
y = math.log (x) si x> 0 else float ('nan')
```

Casi puede leer esta línea como en inglés: "y gets log-x if x es mayor que 0; de lo contrario, se convierte en NaN ".

Las funciones recursivas a veces pueden reescribirse utilizando expresiones condicionales. Por ejemplo, aquí hay una versión recursiva de `factorial`:

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

Podemos reescribirlo así:

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

Otro uso de expresiones condicionales es el manejo de argumentos opcionales. Por ejemplo, aquí está el método init de `GoodKangaroo` (ver Ejercicio 17-2):

```
    def __init__(self, name, contents=None):
        self.name = name
        if contents == None:
            contents = []
        self.pouch_contents = contents
```

Podemos reescribir este como este:

```
    def __init__(self, name, contents=None):
        self.name = name
        self.pouch_contents = [] if contents == None else contents
```

En general, puede reemplazar una instrucción condicional con una expresión condicional si ambas ramas contienen expresiones simples que se devuelven o se asignan a la misma variable.

19.2 Lista de comprensiones

En "Mapa, filtro y reducción", vimos el mapa y los patrones de filtro. Por ejemplo, esta función toma una lista de cadenas, asigna el método de cadena `capitalizea` los elementos y devuelve una nueva lista de cadenas:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

Podemos escribir esto de forma más concisa usando una **lista de comprensión**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

Los operadores de corchetes indican que estamos construyendo una nueva lista. La expresión dentro de los paréntesis especifica los elementos de la lista, y la cláusula `for` indica qué secuencia estamos atravesando.

La sintaxis de una lista de comprensión es un poco incómoda porque la variable de ciclo, `s` en este ejemplo, aparece en la expresión antes de llegar a la definición.

Las listas de comprensión también se pueden usar para filtrar. Por ejemplo, esta función selecciona solo los elementos `t` que están en mayúsculas y devuelve una nueva lista:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

Podemos reescribirlo usando una lista de comprensión:

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

Las listas de comprensión son concisas y fáciles de leer, al menos para expresiones simples. Y generalmente son más rápidos que el equivalente de los bucles, a veces mucho más rápido. Entonces, si estás enojado conmigo por no mencionarlos antes, lo entiendo.

Pero, en mi defensa, las listas de comprensión son más difíciles de depurar porque no se puede poner una declaración de impresión dentro del ciclo. Sugiero que los use solo si el cálculo es lo suficientemente simple como para hacerlo bien la primera vez. Y para los principiantes eso significa nunca.

19.3 Expresiones del generador

Las **expresiones de generador** son similares a las listas de comprensión, pero con paréntesis en lugar de corchetes:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

El resultado es un objeto generador que sabe cómo iterar a través de una secuencia de valores. Pero a diferencia de una lista de comprensión, no calcula los valores de una vez; espera ser preguntado. La función incorporada `next` obtiene el siguiente valor del generador:

```
>>> next(g)
0
>>> next(g)
1
```

Cuando llegue al final de la secuencia, se genera una excepción `StopIteration`. También puede usar un ciclo `for` para recorrer los valores:

```
>>> for val in g:
...     print(val)
4
9
16
```

El objeto del generador realiza un seguimiento de dónde se encuentra en la secuencia, por lo que el ciclo `for` continúa donde se detuvo `next`. Una vez que el generador está agotado, continúa aumentando `StopException`:

```
>>> next(g)
StopIteration
```

Las expresiones generadoras se utilizan a menudo con funciones como `sum`, `max` y `min`:

```
>>> sum(x**2 for x in range(5))
30
```

19.4 any y all

Python proporciona una función incorporada, `any` que toma una secuencia de valores booleanos y devuelve `True` si hay alguno de los valores `True`. Funciona en listas:

```
>>> any([False, False, True])
True
```

Pero a menudo se usa con expresiones de generador:

```
>>> any(letter == 't' for letter in 'monty')
True
```

Ese ejemplo no es muy útil porque hace lo mismo que el operador `in`. Pero podríamos usar anypara reescribir algunas de las funciones de búsqueda que escribimos en "Buscar". Por ejemplo, podríamos escribir `avoids` así:

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

La función casi se lee como en inglés: " word evita forbiddensi no hay letras prohibidas word".

Usar `any` con una expresión de generador es eficiente porque se detiene inmediatamente si encuentra un valor `True`, por lo que no tiene que evaluar toda la secuencia.

Python proporciona otra función incorporada, `all` que devuelve `True` si cada elemento de la secuencia es True. Como ejercicio, use `all` para reescribir `uses_all` de "Buscar".

19.5 Conjuntos

En "Resta Diccionario" uso diccionarios para encontrar las palabras que aparecen en un documento pero no en una lista de palabras. La función que escribí toma `d1`, que contiene las palabras del documento como claves, y `d2` que contiene la lista de palabras. Devuelve un diccionario que contiene las claves `d1` que no están en `d2`:

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
```

```
            res[key] = None
    return res
```

En todos estos diccionarios, los valores se deben a `None` que nunca los utilizamos. Como resultado, desperdiciamos algo de espacio de almacenamiento.

Python proporciona otro tipo incorporado, llamado a `set`, que se comporta como una colección de claves de diccionario sin valores. Agregar elementos a un conjunto es rápido; también lo es verificar la membresía. Y los conjuntos proporcionan métodos y operadores para calcular operaciones de conjuntos comunes.

Por ejemplo, establecer la resta está disponible como un método llamado `difference` o como un operador, -. Entonces podemos reescribir `subtract` de esta manera:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

El resultado es un conjunto en lugar de un diccionario, pero para operaciones como la iteración, el comportamiento es el mismo.

Algunos de los ejercicios de este libro se pueden realizar de manera concisa y eficiente con los sets. Por ejemplo, aquí hay una solución para `has_duplicates`, del Ejercicio 10-7, que usa un diccionario:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

Cuando aparece un elemento por primera vez, se agrega al diccionario. Si el mismo elemento aparece nuevamente, la función regresa `True`.

Usando sets, podemos escribir la misma función como esta:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

Un elemento solo puede aparecer en un conjunto una vez, de modo que si un elemento `t` aparece más de una vez, el conjunto será más pequeño que `t`. Si no hay duplicados, el conjunto tendrá el mismo tamaño que `t`.

También podemos usar conjuntos para hacer algunos de los ejercicios en el Capítulo 9. Por ejemplo, aquí hay una versión de `uses_only` con un bucle:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
```

```
    return True
```

`uses_only` comprueba si todas las letras en `word` están en `available`. Podemos reescribirlo así:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

El <= operador verifica si un conjunto es un subconjunto u otro, incluida la posibilidad de que sean iguales, lo cual es cierto si `word` aparecen todas las letras en `available`.

Como ejercicio, vuelva a escribir `avoids` usando conjuntos.

19.6 Contadores

Un contador es como un conjunto, excepto que si un elemento aparece más de una vez, el contador registra cuántas veces aparece. Si está familiarizado con la idea matemática de un `multiset`, un contador es una forma natural de representar un multiset.

El contador se define en un módulo estándar llamado `collections`, por lo que debe importarlo. Puede inicializar un contador con una cadena, lista o cualquier otra cosa que admita la iteración:

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Los contadores se comportan como diccionarios de muchas maneras; mapean desde cada tecla hasta la cantidad de veces que aparece. Como en los diccionarios, las claves tienen que ser lavables.

A diferencia de los diccionarios, los contadores no generan una excepción si accede a un elemento que no aparece. En cambio, devuelven 0:

```
>>> count['d']
0
```

Podemos usar Contadores para reescribir `is_anagram` del Ejercicio 10-6:

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

Si dos palabras son anagramas, contienen las mismas letras con los mismos conteos, por lo que sus contadores son equivalentes.

Los contadores proporcionan métodos y operadores para realizar operaciones tipo conjunto, que incluyen suma, resta, unión e intersección. Y proporcionan un método a menudo útil `most_common`, que devuelve una lista de pares valor-frecuencia, ordenados de los más comunes a los menos:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
```

```
r 2
p 1
a 1
```

19.7 defaultdict

El módulo `collections` también proporciona `defaultdict`, que es como un diccionario, excepto que si accede a una clave que no existe, puede generar un nuevo valor sobre la marcha.

Cuando crea a `defaultdict`, proporciona una función que se utiliza para crear nuevos valores. Una función utilizada para crear objetos a veces se llama **fábrica**. Las funciones integradas que crean listas, conjuntos y otros tipos se pueden usar como fábricas:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Observe que el argumento es `list`, que es un objeto de clase, no `list()`, que es una nueva lista. La función que proporciona no se invoca a menos que acceda a una clave que no existe:

```
>>> t = d['new key']
>>> t
[]
```

La nueva lista, a la que llamamos `t`, también se agrega al diccionario. Entonces, si modificamos `t`, el cambio aparece en `d`:

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

Si está creando un diccionario de listas, a menudo puede escribir código más simple usando `defaultdict`. En mi solución para el ejercicio 12-2, que puede obtener de *http://thinkpython2.com/code/anagram_sets.py*, hago un diccionario que mapea desde una cadena de letras ordenada a la lista de palabras que se pueden escribir con esas letras. Por ejemplo, mapas a la lista. `'opst'['opts', 'post', 'pots', 'spot', 'stop', 'tops']`

Aquí está el código original:

```
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
```

Esto se puede simplificar utilizando `setdefault`, que podría haber usado en el Ejercicio 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
```

Esta solución tiene el inconveniente de que hace una nueva lista cada vez, independientemente de si es necesaria. Para las listas, no es gran cosa, pero si la función de fábrica es complicada, podría ser.

Podemos evitar este problema y simplificar el código usando 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
```

Mi solución para el Ejercicio 18-3, que puedes descargar desde *http://thinkpython2.com/code/PokerHandSoln.py*, se usa `setdefaul` en la función `thas_straightflush`. Esta solución tiene el inconveniente de crear un objeto `Hand` en todo momento a través del ciclo, ya sea que se necesite o no. Como ejercicio, reescríbelo usando `defaultdict`.

19.8 Tuples nombrados

Muchos objetos simples son básicamente colecciones de valores relacionados. Por ejemplo, el objeto Point definido en el Capítulo 15 contiene dos números, `x` y `y`. Cuando defines una clase como esta, usualmente comienzas con un método init y un método str:

```
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)
```

Este es un gran código para transmitir una pequeña cantidad de información. Python proporciona una forma más concisa de decir lo mismo:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

El primer argumento es el nombre de la clase que desea crear. El segundo es una lista de los atributos que los objetos de puntos deben tener, como cadenas. El valor de retorno de `namedtuple` es un objeto de clase:

```
>>> Point
<class '__main__.Point'>
```

`Point` proporciona automáticamente métodos como `__init__` y `__str__` para que no tenga que escribirlos.

Para crear un objeto Point, usa la clase Point como una función:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

El método init asigna los argumentos a los atributos usando los nombres que proporcionó. El método str imprime una representación del objeto Point y sus atributos.

Puede acceder a los elementos de la tupla nombrada por su nombre:

```
>>> p.x, p.y
(1, 2)
```

Pero también puedes tratar una tupla nombrada como una tupla:

```
>>> p[0], p[1]
(1, 2)


>>> x, y = p
>>> x, y
(1, 2)
```

Las tuplas con nombre proporcionan una forma rápida de definir clases simples. El inconveniente es que las clases simples no siempre son simples. Más adelante puede decidir que desea agregar métodos a una tupla nombrada. En ese caso, podría definir una nueva clase que herede de la tupla nombrada:

```
class Pointier(Point):
    # add more methods here
```

O puede cambiar a una definición de clase convencional.

19.9 Recopilación de Args de palabras clave

En "Tuplas de argumento de longitud variable" , vimos cómo escribir una función que reúne sus argumentos en una tupla:

```
def printall(*args):
    print(args)
```

Puede llamar a esta función con cualquier número de argumentos posicionales (es decir, argumentos que no tienen palabras clave):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Pero el operador * no recoge los argumentos de palabra clave:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

Para reunir argumentos de palabra clave, puede usar el operador **:

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

Puede llamar al parámetro de recopilación de palabras clave todo lo que desee, pero `kwargs` es una opción común. El resultado es un diccionario que asigna palabras clave a valores:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

Si tiene un diccionario de palabras clave y valores, puede usar el operador scatter, **, para llamar a una función:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Sin que el operador de dispersión, la función de la trataría dcomo un solo argumento posicional, por lo que sería asignar `d` a `x` y se quejan porque no hay nada para asignar a `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'
```

Cuando trabaje con funciones que tienen una gran cantidad de parámetros, a menudo es útil crear y pasar diccionarios que especifiquen las opciones más utilizadas.

19.10 Glosario

expresión condicional:

Una expresión que tiene uno de dos valores, dependiendo de una condición.

lista de comprensión:

Una expresión con un bucle `for` entre corchetes que arroja una nueva lista.

expresión del generador:

Una expresión con un bucle `for` entre paréntesis que produce un objeto generador.

multiset:

Una entidad matemática que representa un mapeo entre los elementos de un conjunto y el número de veces que aparecen.

fábrica:

Una función, generalmente aprobada como un parámetro, utilizada para crear objetos.

19.11 Ejercicios

Ejercicio 19-1.

La siguiente es una función que calcula el coeficiente binomial recursivamente:

```
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
```

Reescribe el cuerpo de la función usando expresiones condicionales anidadas.

Una nota: esta función no es muy eficiente porque termina computando los mismos valores una y otra vez. Puede hacerlo más eficiente mediante la memorización (consulte "Memos"). Pero descubrirá que es más difícil de memorizar si lo escribe usando expresiones condicionales.

Capítulo 20

Depuración

Cuando está depurando, debe distinguir entre diferentes tipos de errores para rastrearlos más rápidamente:

- El intérprete descubre los errores de sintaxis cuando está traduciendo el código fuente a código de bytes. Indican que hay algo mal con la estructura del programa. Ejemplo: Omitir los dos puntos al final de una instrucción `def` genera el mensaje algo redundante `SyntaxError: invalid syntax`.

- Los errores de tiempo de ejecución son producidos por el intérprete si algo sale mal mientras el programa se está ejecutando. La mayoría de los mensajes de error de tiempo de ejecución incluyen información sobre dónde se produjo el error y qué funciones se estaban ejecutando. Ejemplo: una recursión infinita eventualmente causa el error de tiempo de ejecución `maximum recursion depth exceeded`.

- Los errores semánticos son problemas con un programa que se ejecuta sin generar mensajes de error pero que no hace lo correcto. Ejemplo: Es posible que una expresión no se evalúe en el orden esperado, lo que arroja un resultado incorrecto.

El primer paso en la depuración es descubrir qué tipo de error está tratando. Aunque las siguientes secciones están organizadas por tipo de error, algunas técnicas son aplicables en más de una situación.

20.1 Errores de sintaxis

Los errores de sintaxis suelen ser fáciles de corregir una vez que descubres cuáles son. Lamentablemente, los mensajes de error a menudo no son útiles. Los mensajes más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, ninguno de los cuales es muy informativo.

Por otro lado, el mensaje te dice en qué parte del programa se produjo el problema. En realidad, te dice dónde detectó Python un problema, que no es necesariamente donde está el error. A veces, el error es anterior a la ubicación del mensaje de error, a menudo en la línea anterior.

Si está construyendo el programa de forma incremental, debe tener una buena idea sobre dónde está el error. Estará en la última línea que agregó.

Si está copiando código de un libro, comience por comparar su código con el código del libro con mucho cuidado. Verifica cada personaje Al mismo tiempo, recuerde que el libro podría estar equivocado, de modo que si ve algo que se parece a un error de sintaxis, podría serlo.

Estas son algunas formas de evitar los errores de sintaxis más comunes:

1. Asegúrese de no estar usando una palabra clave de Python para un nombre de variable.

2. Compruebe que tiene dos puntos al final de la cabecera de cada sentencia compuesta, incluyendo `for`, `while`, `if`, y declaraciones `def`.

3. Asegúrese de que las cadenas del código tengan comillas coincidentes. Asegúrese de que todas las comillas sean citas rectas, no comillas.

4. Si tiene cadenas multilínea con comillas triples (simples o dobles), asegúrese de haber terminado la cadena correctamente. Una cadena no terminada puede causar un `invalid token` error al final de su programa, o puede tratar la siguiente parte del programa como una cadena hasta que llegue a la siguiente cadena. En el segundo caso, ¡podría no producir ningún mensaje de error!

5. Sin cerrar una abertura del operador (, {o [-hace Python continuar con la siguiente línea como parte de la instrucción actual. En general, se produce un error casi de inmediato en la siguiente línea.

6. Compruebe el clásico en =lugar de ==dentro de un condicional.

7. Compruebe la sangría para asegurarse de que se alinee de la forma en que se supone que debe. Python puede manejar el espacio y las pestañas, pero si los mezclas, puede causar problemas. La mejor forma de evitar este problema es usar un editor de texto que conozca Python y genere sangrías consistentes.

8. Si tiene caracteres que no son ASCII en el código (incluyendo cadenas y comentarios), eso podría causar un problema, aunque Python 3 normalmente maneja caracteres que no son ASCII. Tenga cuidado si pega texto en una página web u otra fuente.

Si nada funciona, pase a la siguiente sección ...

20.1.1 Sigo haciendo cambios y no hace ninguna diferencia.

Si el intérprete dice que hay un error y usted no lo ve, puede ser porque usted y el intérprete no están mirando el mismo código. Verifique su entorno de programación para asegurarse de que el programa que está editando es el que Python está tratando de ejecutar.

Si no está seguro, intente poner un error de sintaxis obvio y deliberado al comienzo del programa. Ahora ejecútalo de nuevo. Si el intérprete no encuentra el nuevo error, no está ejecutando el nuevo código.

Hay algunos posibles culpables:

- Editó el archivo y olvidó guardar los cambios antes de volver a ejecutarlo. Algunos entornos de programación hacen esto por usted, pero otros no.

- Cambió el nombre del archivo, pero todavía está ejecutando el nombre anterior.

- Algo en su entorno de desarrollo está configurado incorrectamente.

- Si está escribiendo un módulo y lo está utilizando `import`, asegúrese de que no le dé a su módulo el mismo nombre que uno de los módulos estándar de Python.

- Si está utilizando `import` para leer un módulo, recuerde que debe reiniciar el intérprete o utilizarlo `reload` para leer un archivo modificado. Si vuelve a importar el módulo, no hace nada.

Si te quedas atascado y no puedes descubrir lo que está pasando, un enfoque es comenzar de nuevo con un nuevo programa como "¡Hola, Mundo!" Y asegurarte de que puedas ejecutar un programa conocido. Luego, gradualmente agregue las piezas del programa original al nuevo.

20.2 Errores en tiempo de ejecución

Una vez que su programa es sintácticamente correcto, Python puede leerlo y al menos comenzar a ejecutarlo. ¿Qué podría salir mal?

20.2.1 Mi programa no hace absolutamente nada.

Este problema es más común cuando su archivo consta de funciones y clases, pero en realidad no invoca una función para iniciar la ejecución. Esto puede ser intencional si solo planea importar este módulo para proporcionar clases y funciones.

Si no es intencional, asegúrese de que haya una llamada de función en el programa y asegúrese de que el flujo de ejecución llegue a ella (consulte "Flujo de ejecución" a continuación).

20.2.2 Mi programa se cuelga

Si un programa se detiene y parece que no está haciendo nada, está "colgando". A menudo eso significa que está atrapado en un ciclo infinito o recursión infinita.

- Si hay un bucle particular que sospecha que es el problema, agregue una declaración `print` inmediatamente antes del bucle que dice "ingresar al bucle" y otra inmediatamente después que diga "salir del bucle".

  Ejecuta el programa. Si obtienes el primer mensaje y no el segundo, tienes un ciclo infinito. Ve a la sección "Bucle infinito" a continuación.

- La mayoría de las veces, una recursión infinita causará que el programa se ejecute por un tiempo y luego produzca un error "RuntimeError: Maximum Recursion depth Excedido". Si eso sucede, vaya a la sección "Recursión infinita" a continuación.

  Si no obtiene este error pero sospecha que hay un problema con un método o función recursiva, puede seguir usando las técnicas en la sección "Recursión infinita".

- Si ninguno de esos pasos funciona, comience a probar otros bucles y otras funciones recursivas y métodos.

- Si eso no funciona, es posible que no comprenda el flujo de ejecución en su programa. Vaya a la sección "Flujo de ejecución" a continuación.

20.2.2.1 BUCLE INFINITO

Si cree que tiene un ciclo infinito y cree que sabe qué ciclo está causando el problema, agregue una instrucción `print` al final del ciclo que imprima los valores de las variables en la condición y el valor de la condición.

Por ejemplo:

```
while x > 0 and y < 0 :
    # do something to x
    # do something to y


    print('x: ', x)
```

```
print('y: ', y)
print("condition: ", (x > 0 and y < 0))
```

Ahora cuando ejecutas el programa, verás tres líneas de salida para cada vez a través del ciclo. La última vez a través del ciclo, la condición debería ser `False`. Si el ciclo continúa, podrá ver los valores de xy y, y puede averiguar por qué no se están actualizando correctamente.

20.2.2.2 RECURSIÓN INFINITA

La mayoría de las veces, la recursión infinita hace que el programa se ejecute por un tiempo y luego produce un error `Maximum recursion depth exceeded`.

Si sospecha que una función está causando una recursión infinita, asegúrese de que haya un caso base. Debería haber alguna condición que haga que la función regrese sin hacer una invocación recursiva. De lo contrario, debe reconsiderar el algoritmo e identificar un caso base.

Si hay un caso base pero el programa no parece estar llegando a él, agregue una declaración `print` al comienzo de la función que imprime los parámetros. Ahora cuando ejecuta el programa, verá algunas líneas de salida cada vez que se invoca la función, y verá los valores de los parámetros. Si los parámetros no se están moviendo hacia el caso base, obtendrá algunas ideas sobre por qué no.

20.2.2.3 FLUJO DE EJECUCIÓN

Si no está seguro de cómo se está moviendo el flujo de ejecución a través de su programa, agregue declaraciones `print` al comienzo de cada función con un mensaje como "ingresar función `foo`", donde está el nombre de la función.

Ahora cuando ejecuta el programa, imprimirá un rastro de cada función a medida que se invoca.

20.2.3 Cuando ejecuto el programa obtengo una excepción.

Si algo sale 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 llamó que, y así sucesivamente. En otras palabras, rastrea la secuencia de llamadas a funciones que lo llevaron a donde se encuentra, incluido el número de línea en su archivo donde ocurrió cada llamada.

El primer paso es examinar el lugar en el programa donde ocurrió el error y ver si puede averiguar qué sucedió. Estos son algunos de los errores de tiempo de ejecución más comunes:

NameError:

> Está intentando usar una variable que no existe en el entorno actual. Verifica si el nombre está bien escrito, o al menos consistentemente. Y recuerde que las variables locales son locales; no puede hacer referencia a ellos desde fuera de la función donde están definidos.

Error de tecleado:

Hay varias causas posibles:

- Está intentando usar un valor incorrectamente. Ejemplo: indizar una cadena, lista o tupla con algo que no sea un entero.

- Hay una falta de coincidencia entre los elementos en una cadena de formato y los elementos pasados para la conversión. Esto puede suceder si el número de elementos no coincide o si se solicita una conversión no válida.

- Estás pasando la cantidad de argumentos incorrecta a una función. Para los métodos, mira la definición del método y verifica que el primer parámetro sea `self`. Luego mira la invocación del método; asegúrese de invocar el método en un objeto con el tipo correcto y proporcionar los otros argumentos correctamente.

KeyError:

> Está intentando acceder a un elemento de un diccionario utilizando una clave que el diccionario no contiene. Si las claves son cadenas, recuerde que las mayúsculas son importantes.

AttributeError:

> Está intentando acceder a un atributo o método que no existe. ¡Revisar la ortografía! Puede usar la función incorporada `vars` para enumerar los atributos que existen.
>
> Si un AttributeError indica que un objeto tiene `NoneType`, eso significa que sí lo es `None`. Entonces, el problema no es el nombre del atributo, sino el objeto.
>
> La razón por la cual el objeto es ninguno podría ser que olvidó devolver un valor de una función; si llega al final de una función sin presionar un enunciado `return`, regresa `None`. Otra causa común es usar el resultado de un método de lista, como `sort`, que devuelve `None`.

IndexError:

> El índice que está utilizando para acceder a una lista, cadena o tupla es mayor que su longitud menos uno. Inmediatamente antes del sitio del error, agregue una print declaración para mostrar el valor del índice y la longitud de la matriz. ¿Es la matriz del tamaño correcto? ¿Es el índice el valor correcto?

El depurador de Python (`pdb`) es útil para rastrear excepciones porque le permite examinar el estado del programa inmediatamente antes del error. Puede leer `pdb` en *https://docs.python.org/3/library/pdb.html*.

20.2.4 Agregué tantas declaraciones impresas que me inundan de resultados.

Uno de los problemas con el uso de sentencias `print` para la depuración es que puede terminar enterrado en la salida. Hay dos formas de proceder: simplificar el resultado o simplificar el programa.

Para simplificar el resultado, puede eliminar o comentar declaraciones `print` que no ayudan, o combinarlas, o formatear el resultado para que sea más fácil de entender.

Para simplificar el programa, hay varias cosas que puede hacer. Primero, reduzca el problema en el que está trabajando el programa. Por ejemplo, si está buscando una lista, busque una pequeña lista. Si el programa recibe información del usuario, proporciónele la información más simple que causa el problema.

Segundo, limpia el programa. Elimine el código muerto y reorganice el programa para que sea tan fácil de leer como sea posible. Por ejemplo, si sospecha que el problema se encuentra en una parte profundamente anidada del programa, intente volver a escribir esa parte con una estructura más simple. Si sospecha que hay una función grande, intente dividirla en funciones más pequeñas y probarlas por separado.

A menudo, el proceso de encontrar el caso de prueba mínimo te lleva al error. Si encuentra que un programa funciona en una situación pero no en otra, eso le da una pista sobre lo que está sucediendo.

Del mismo modo, volver a escribir un fragmento de código puede ayudarlo a encontrar errores sutiles. Si realiza un cambio que cree que no debería afectar el programa, y lo hace, eso puede incitarlo.

20.3 Errores semánticos

De alguna manera, los errores semánticos son los más difíciles de depurar, porque el intérprete no proporciona información sobre lo que 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 está viendo. Necesitas una hipótesis sobre lo que el programa realmente está haciendo. Una de las cosas que lo hace tan difícil es que las computadoras funcionan tan rápido.

A menudo desearás poder ralentizar el programa a la velocidad humana, y con algunos depuradores puedes. Pero el tiempo que se tarda en insertar algunas print declaraciones bien ubicadas es a menudo corto en comparación con la configuración del depurador, la inserción y eliminación de puntos de interrupción y el "escalonamiento" del programa hasta donde se produce el error.

20.3.1 Mi programa no funciona

Debería hacerse estas preguntas:

- ¿Hay algo que se suponía que el programa debía hacer pero que no parece estar sucediendo? Busque la sección del código que realiza esa función y asegúrese de que se está ejecutando cuando lo crea oportuno.

- ¿Está sucediendo algo que no debería suceder? Busque el código en su programa que realiza esa función y vea si se está ejecutando cuando no debería.

- ¿Hay una sección de código produciendo un efecto que no es lo que esperabas? Asegúrese de comprender el código en cuestión, especialmente si implica funciones o métodos en otros módulos de Python. Lea la documentación de las funciones que llama. Pruébelos escribiendo casos de prueba simples y verificando los resultados.

Para programar, necesitas un modelo mental de cómo funcionan los programas. Si escribe un programa que no hace lo que espera, a menudo el problema no está en el programa; está en tu modelo mental.

La mejor manera de corregir su modelo mental es dividir el programa en sus componentes (normalmente las funciones y métodos) y probar cada componente de forma independiente. Una vez que encuentre la discrepancia entre su modelo y la realidad, puede resolver el problema.

Por supuesto, debe construir y probar componentes a medida que desarrolla el programa. Si encuentra un problema, solo debería haber una pequeña cantidad de código nuevo que no se sabe que es correcto.

20.3.2 Tengo una gran expresión peluda y no hace lo que espero.

Escribir expresiones complejas está bien siempre que sean legibles, pero puede ser difícil depurarlas. A menudo es una buena idea dividir una expresión compleja en una serie de asignaciones a variables temporales.

Por ejemplo:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

Esto puede ser reescrito 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 las variables proporcionan documentación adicional, y es más fácil de depurar porque puede verificar los tipos de las variables intermedias y mostrar sus valores.

Otro problema que puede ocurrir con las expresiones grandes es que el orden de evaluación puede no ser el esperado. Por ejemplo, si está traduciendo la expresión $\frac{x}{2\pi}$ a Python, puede escribir:

```
y = x / 2 * math.pi
```

Eso no es correcto porque la multiplicación y la división tienen la misma precedencia y se evalúan de izquierda a derecha. Entonces esta expresión computa $x\pi/2$.

Una buena forma de depurar expresiones es agregar paréntesis para hacer que el orden de evaluación sea explícito:

```
y = x / (2 * math.pi)
```

Cuando no esté seguro del orden de la evaluación, use paréntesis. No solo el programa será correcto (en el sentido de hacer lo que usted desea), sino que también será más legible para otras personas que no hayan memorizado el orden de las operaciones.

20.3.3 Tengo una función que no devuelve lo que esperaba.

Si tiene una declaración `return` con una expresión compleja, no tiene la posibilidad de imprimir el resultado antes de regresar. 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 tiene la oportunidad de mostrar el valor de `count` antes de retornar.

20.3.4 Estoy realmente atascado y necesito ayuda.

Primero, trate de alejarse de la computadora por unos minutos. Las computadoras emiten ondas que afectan el cerebro, causando estos síntomas:

- Frustración y rabia
- Creencias supersticiosas ("la computadora me odia") y pensamiento mágico ("el programa solo funciona cuando me pongo el sombrero hacia atrás").
- Programación de paseo aleatorio (el intento de programar escribiendo cada programa posible y eligiendo el que hace lo correcto).

Si te encuentras con alguno de estos síntomas, levántate y sal a caminar. Cuando estés tranquilo, piensa en el programa. ¿Qué está haciendo? ¿Cuáles son algunas causas posibles de ese comportamiento? ¿Cuándo fue la última vez que tenía un programa en funcionamiento y qué hizo a continuación?

A veces solo lleva tiempo encontrar un error. A menudo encuentro errores cuando estoy lejos de la computadora y dejo que mi mente divague. Algunos de los mejores lugares para encontrar errores son los trenes, la ducha y la cama justo antes de dormirse.

20.3.5 No, realmente necesito ayuda.

Sucede. Incluso los mejores programadores a veces se atascan. Algunas veces trabajas en un programa por tanto tiempo que no puedes ver el error. Necesitas un par de ojos frescos.

Antes de traer a alguien más, asegúrese de estar preparado. Su programa debe ser lo más simple posible, y debe estar trabajando en la entrada más pequeña que causa el error. Debe tener las declaraciones `print` en los lugares apropiados (y la salida que producen debe ser comprensible). Debe entender el problema lo suficientemente bien como para describirlo de manera concisa.

Cuando traiga a alguien para ayudar, asegúrese de darles la información que necesitan:

- Si hay un mensaje de error, ¿qué es y qué parte del programa indica?
- ¿Qué fue lo último que hizo antes de que ocurriera este error? ¿Cuáles fueron las últimas líneas de código que escribió 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 qué podrías haber hecho para encontrarlo más rápido. La próxima vez que vea algo similar, podrá encontrar el error más rápidamente.

Recuerde, el objetivo no es solo hacer que el programa funcione. El objetivo es aprender cómo hacer que el programa funcione.

Capítulo 21

Análisis de algoritmos

Este apéndice es un extracto editado de Think Complexity, de Allen B. Downey, también publicado por O'Reilly Media (2012). Cuando haya terminado con este libro, es posible que desee pasar a ese.

El análisis de algoritmos es una rama de la informática que estudia el rendimiento de los algoritmos, especialmente sus requisitos de tiempo de ejecución y espacio. Ver *http://en.wikipedia.org/wiki/Analysis_of_algorithms*.

El objetivo práctico del análisis de algoritmos es predecir el rendimiento de diferentes algoritmos para guiar las decisiones de diseño.

Durante la campaña presidencial de Estados Unidos en 2008, se le pidió al candidato Barack Obama que realizara un análisis improvisado cuando visitó Google. El presidente ejecutivo Eric Schmidt le preguntó en broma por "la forma más eficiente de clasificar un millón de enteros de 32 bits". Al parecer, Obama recibió una notificación, porque respondió rápidamente: "Creo que el tipo de burbuja sería el camino equivocado". Ver *http://bit.ly/1MpIwTf*.

Esto es cierto: el ordenamiento de burbujas es conceptualmente simple pero lento para grandes conjuntos de datos. La respuesta que Schmidt probablemente estaba buscando es "ordenar por radix" (*http://en.wikipedia.org/wiki/Radix_sort*).

El objetivo del análisis de algoritmos es hacer comparaciones significativas entre algoritmos, pero hay algunos problemas:

- El rendimiento relativo de los algoritmos puede depender de las características del hardware, por lo que un algoritmo puede ser más rápido en la Máquina A, otro en la Máquina B. La solución general a este problema es especificar un **modelo de máquina** y analizar el número de pasos u operaciones, un algoritmo requiere bajo un modelo dado.

- El rendimiento relativo puede depender de los detalles del conjunto de datos. Por ejemplo, algunos algoritmos de clasificación se ejecutan más rápido si los datos ya están parcialmente ordenados; otros algoritmos corren más lento en este caso. Una forma común de evitar este problema es analizar el **peor de los casos**. A veces es útil analizar el rendimiento de un caso promedio, pero eso suele ser más difícil, y puede que no sea obvio qué conjunto de casos promediar.

- El rendimiento relativo también depende del tamaño del problema. Un algoritmo de clasificación que sea rápido para listas pequeñas puede ser lento para listas largas. La solución habitual a este problema es expresar el tiempo de ejecución (o número de operaciones) como una función del tamaño del problema, y agrupar las funciones en categorías según cuán rápido crecen a medida que aumenta el tamaño del problema.

Lo bueno de este tipo de comparación es que se presta a la clasificación simple de algoritmos. Por ejemplo, si sé que el tiempo de ejecución del algoritmo A tiende a ser proporcional al tamaño de la entrada, n y que el

algoritmo B tiende a ser proporcional a n , entonces espero que A sea más rápido que B, al menos para grandes valores de n .

Este tipo de análisis viene con algunas advertencias, pero lo abordaremos más adelante.

21.1 Orden de crecimiento

Supongamos que ha analizado dos algoritmos y expresado sus tiempos de ejecución en términos del tamaño de la entrada: el algoritmo A toma 100 n + 1 pasos para resolver un problema con el tamaño n; El algoritmo B toma $n^2 + n + 1$ medidas.

La siguiente tabla muestra el tiempo de ejecución de estos algoritmos para diferentes tamaños de problema:

Tamaño de entrada	Tiempo de ejecución del algoritmo A	Tiempo de ejecución del algoritmo B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	> 10^{10}