Contents

1.1 Introducción **1**
1.1.1 Programación en Python 2
1.1.2 Instalación de Python 3 3
1.1.3 Sesiones interactivas 3
1.1.4 Primer ejemplo 4
1.1.5 Orientación práctica: Errores 6

\tableofcontents

1.1 Introducción

La informática es una disciplina académica tremendamente amplia. Las áreas de sistemas distribuidos globalmente, inteligencia artificial, robótica, gráficos, seguridad, computación científica, arquitectura informática y docenas de subcampos emergentes se expanden con nuevas técnicas y descubrimientos cada año. El rápido progreso de la informática ha dejado pocos aspectos de la vida humana intactos. El comercio, la comunicación, la ciencia, el arte, el ocio y la política se han reinventado como dominios computacionales.

La enorme productividad de la informática sólo es posible porque se basa en un conjunto elegante y potente de ideas fundamentales. Toda computación comienza con la representación de la información, la especificación de la lógica para procesarla y el diseño de abstracciones que gestionen la complejidad de esa lógica. Para dominar estos fundamentos será necesario que entendamos con precisión cómo interpretan los ordenadores los programas informáticos y llevan a cabo los procesos computacionales.

Estas ideas fundamentales se han enseñado durante mucho tiempo en Berkeley utilizando el libro de texto clásico._Estructura e interpretación de programas informáticos_ (SICPI) de Harold Abelson y Gerald Jay Sussman con Julie Sussman. Estas notas de clase se basan en gran medida en ese libro de texto, que los autores originales han licenciado amablemente para su adaptación y reutilización.

El inicio de nuestro viaje intelectual no requiere revisión, ni debemos esperar que alguna vez la requiera.

> Estamos a punto de estudiar la idea de una *proceso computacional* Los procesos computacionales son seres abstractos que habitan en las computadoras. A medida que evolucionan, los procesos manipulan otras cosas abstractas llamadas datos. La evolución de un proceso está dirigida por un patrón de reglas llamado programa. Las personas crean programas para dirigir los procesos. En efecto, conjuramos a los espíritus de la computadora con nuestros hechizos.
>
> Los programas que utilizamos para conjurar procesos son como los

> hechizos de un brujo. Están cuidadosamente compuestos a partir de expresiones simbólicas en lenguaje arcano y esotérico._lenguajes de programación_ que prescriben las tareas que queremos que nuestros procesos realicen.
>
> Un proceso computacional, en un ordenador que funciona correctamente, ejecuta los programas con precisión y exactitud. Por lo tanto, como el aprendiz de brujo, los programadores novatos deben aprender a comprender y prever las consecuencias de sus conjuros.
>
> —Abelson y Sussman,SICPI(1993)

1.1.1 Programación en Python

> Un idioma no es algo que se aprende sino algo a lo que te unes.
>
> —Arika Okrent

Para definir procesos computacionales, necesitamos un lenguaje de programación; preferiblemente uno que muchos humanos y una gran variedad de computadoras puedan entender. En este curso, aprenderemos losPitónidioma.

Python es un lenguaje de programación ampliamente utilizado que ha reclutado entusiastas de muchas profesiones: programadores web, ingenieros de juegos, científicos, académicos e incluso diseñadores de nuevos lenguajes de programación. Cuando aprendes Python, te unes a una comunidad de desarrolladores de un millón de personas. Las comunidades de desarrolladores son instituciones tremendamente importantes: los miembros se ayudan entre sí a resolver problemas, comparten su código y experiencias y desarrollan software y herramientas de forma colectiva. Los miembros dedicados suelen alcanzar la celebridad y el reconocimiento generalizado por sus contribuciones. Tal vez algún día te nombren entre estos Pythonistas de élite.

El lenguaje Python en sí es el producto de unagran comunidad de voluntariosque se enorgullece de ladiversidadde sus colaboradores. El lenguaje fue concebido e implementado por primera vez porGuido van Rossuma finales de los años 1980. El primer capítulo de suTutorial de Python 3explica por qué Python es tan popular entre los muchos lenguajes disponibles hoy en día.

Python se destaca como lenguaje instruccional porque, a lo largo de su historia, los desarrolladores de Python han enfatizado la interpretabilidad humana del código Python, reforzada por laZen de PythonLos principios rectores de la belleza, la simplicidad y la legibilidad son los que guían este curso. Python es particularmente apropiado para este curso porque su amplio conjunto de características admite una variedad de estilos de programación diferentes, que exploraremos. Si bien no existe una única forma de programar en Python, hay un conjunto de convenciones compartidas en toda la comunidad de desarrolladores que facilitan el proceso de lectura, comprensión y extensión de los programas existentes. Por lo tanto, la combinación de gran flexibilidad y accesibilidad de

Python permite a los estudiantes explorar muchos paradigmas de programación y luego aplicar sus conocimientos recién adquiridos a miles deProyectos en curso.

Estas notas mantienen el espíritu deSICPIPresentando las características de Python en sintonía con técnicas de diseño de abstracción y un modelo riguroso de computación. Además, estas notas brindan una introducción práctica a la programación en Python, incluidas algunas características avanzadas del lenguaje y ejemplos ilustrativos. Aprender Python será algo natural a medida que avance en el curso.

Sin embargo, Python es un lenguaje rico con muchas características y usos, y los vamos introduciendo de forma consciente y paulatina a medida que incorporamos conceptos fundamentales de informática. Para los estudiantes experimentados que quieran asimilar todos los detalles del lenguaje rápidamente, recomendamos leer el libro de Mark PilgrimSumérgete en Python 3, que está disponible gratuitamente en línea. Los temas de ese libro difieren sustancialmente de los temas de este curso, pero el libro contiene información práctica muy valiosa sobre el uso del lenguaje Python. Tenga cuidado: a diferencia de estas notas, Dive Into Python 3 supone una experiencia sustancial en programación.

La mejor forma de comenzar a programar en Python es interactuar directamente con el intérprete. En esta sección se describe cómo instalar Python 3, iniciar una sesión interactiva con el intérprete y comenzar a programar.

1.1.2 Instalación de Python 3

Como sucede con todo buen software, Python tiene muchas versiones. Este curso utilizará la versión estable más reciente de Python 3 (actualmente Python 3.2). Muchas computadoras ya tienen instaladas versiones anteriores de Python, pero no serán suficientes para este curso. Debería poder usar cualquier computadora para este curso, pero deberá instalar Python 3. No se preocupe, Python es gratuito.

Dive Into Python 3 tiene información detalladaInstrucciones de instalaciónpara todas las plataformas principales. Estas instrucciones mencionan Python 3.1 varias veces, pero es mejor usar Python 3.2 (aunque las diferencias son insignificantes para este curso). Todas las máquinas de instrucción en el departamento de Ingeniería Eléctrica, Electrónica y Computación ya tienen instalado Python 3.2.

1.1.3 Sesiones interactivas

En una sesión interactiva de Python, escribes algo de código Python *código* Después de la *inmediato* , `>>>`. La pitón *intérprete* Lee y evalúa lo que escribes, ejecutando tus diversos comandos.

Existen varias formas de iniciar una sesión interactiva y difieren en sus propiedades. Pruébelas todas para descubrir cuál prefiere. Todas utilizan

exactamente el mismo intérprete detrás de escena.

- La forma más sencilla y común es ejecutar la aplicación Python 3. Tipo`python3`en un indicador de terminal (Mac/Unix/Linux) o abra la aplicación Python 3 en Windows.

- Una aplicación más fácil de usar para quienes están aprendiendo el idioma se llama Idle 3 (`idle3`). Idle colorea tu código (lo que se denomina resaltado de sintaxis), muestra sugerencias de uso y marca la fuente de algunos errores. Idle siempre se incluye con Python, por lo que ya lo tienes instalado. - El editor de Emacs puede ejecutar una sesión interactiva dentro de uno de sus búferes. Si bien es un poco más difícil de aprender, Emacs es un editor potente y versátil para cualquier lenguaje de programación. Lee el Tutorial de Emacs 61A para comenzar. Muchos programadores que invierten tiempo en aprender Emacs nunca vuelven a cambiar de editor.

En cualquier caso, si ves el mensaje de Python,`>>>`, entonces ha iniciado con éxito una sesión interactiva. Estas notas muestran interacciones de ejemplo utilizando el mensaje, seguido de algunos datos de entrada.

```
2 + 2
```

Controles: Cada sesión guarda un historial de lo que has escrito. Para acceder a ese historial, pulsa`<Control>-P`(anterior) y`<Control>-N`(próximo).`<Control>-D`sale de una sesión, lo que descarta este historial.

1.1.4 Primer ejemplo

> Y, a medida que la imaginación se materializa
>
> Las formas de las cosas desconocidas y la pluma del poeta
>
> Los convierte en formas y no les da nada a lo aéreo.
>
> Una vivienda local y un nombre.
>
> —William Shakespeare, El sueño de una noche de verano

Para darle a Python la introducción que se merece, comenzaremos con un ejemplo que utiliza varias características del lenguaje. En la siguiente sección, tendremos que empezar desde cero y construir el lenguaje pieza por pieza. Piense en esta sección como un adelanto de las potentes características que vendrán.

Python tiene soporte integrado para una amplia gama de actividades de programación comunes, como manipular texto, mostrar gráficos y comunicarse a través de Internet. La declaración import

```
from urllib.request import urlopen
```

carga funcionalidad para acceder a datos en Internet. En particular, pone a disposición una función llamada`urlopen`, que puede acceder al contenido en un localizador uniforme de recursos (URL), que es la ubicación de algo en Internet.

Declaraciones y expresiones El código Python consta de declaraciones y expresiones. En términos generales, los programas informáticos constan de instrucciones para

1. Calcular algún valor
2. Realizar alguna acción

Las sentencias suelen describir acciones. Cuando el intérprete de Python ejecuta una sentencia, lleva a cabo la acción correspondiente. Por otro lado, las expresiones suelen describir cálculos que generan valores. Cuando Python evalúa una expresión, calcula su valor. En este capítulo se presentan varios tipos de sentencias y expresiones.

La declaración de asignación

```
shakespeare = urlopen('http://inst.eecs.berkeley.edu/~cs61a/fa11/shakespeare.txt')
```

asocia el nombre`shakespeare`con el valor de la expresión que sigue. Esa expresión aplica el`urlopen`función a una URL que contiene el texto completo de las 37 obras de William Shakespeare, todo en un solo documento de texto.

Funciones Las funciones encapsulan la lógica que manipula los datos. Una dirección web es un fragmento de datos, y el texto de las obras de Shakespeare es otro. El proceso por el cual la primera conduce a la segunda puede ser complejo, pero podemos aplicar ese proceso utilizando solo una expresión simple porque esa complejidad está oculta dentro de una función. Las funciones son el tema principal de este capítulo.

Otra declaración de asignación

```
words = set(shakespeare.read().decode().split())
```

asocia el nombre`words`al conjunto de todas las palabras únicas que aparecen en las obras de Shakespeare, las 33.721. La cadena de órdenes para`read`, `decode`, y`split`, cada uno opera sobre una entidad computacional intermedia: se leen los datos de la URL abierta, esos datos se decodifican en texto y ese texto se divide en palabras. Todas esas palabras se colocan en un`set`.

Objetos Un conjunto es un tipo de objeto que admite operaciones de conjunto como el cálculo de intersecciones y la comprobación de la pertenencia. Un objeto agrupa de forma transparente los datos y la lógica que los manipula, de forma que oculta la complejidad de ambos. Los objetos son el tema principal del Capítulo 2.

La expresión

```
{w for w in words if len(w) >= 5 and w[::-1] in words}
```

```
{'madam', 'stink', 'leets', 'rever', 'drawer', 'stops', 'sessa',
'repaid', 'speed', 'redder', 'devil', 'minim', 'spots', 'asses',
'refer', 'lived', 'keels', 'diaper', 'sleek', 'steel', 'leper',
'level', 'deeps', 'repel', 'reward', 'knits'}
```

es una expresión compuesta que evalúa el conjunto de palabras shakespearianas que aparecen tanto al derecho como al revés. La notación crípticaw[::-1]enumera cada letra de una palabra, pero la-1dice dar un paso atrás::(Aquí significa que las posiciones del primer y último carácter a enumerar son las predeterminadas). Cuando ingresa una expresión en una sesión interactiva, Python imprime su valor en la siguiente línea, como se muestra.

Intérpretes La evaluación de expresiones compuestas requiere un procedimiento preciso que interprete el código de una manera predecible. Un programa que implementa dicho procedimiento, evaluando expresiones y sentencias compuestas, se denomina intérprete. El diseño y la implementación de intérpretes es el tema principal del Capítulo 3.

En comparación con otros programas informáticos, los intérpretes de lenguajes de programación son únicos por su generalidad. Python no fue diseñado con Shakespeare ni palíndromos en mente. Sin embargo, su gran flexibilidad nos permitió procesar una gran cantidad de texto con solo unas pocas líneas de código.

Al final, descubriremos que todos estos conceptos básicos están estrechamente relacionados: las funciones son objetos, los objetos son funciones y los intérpretes son instancias de ambos. Sin embargo, desarrollar una comprensión clara de cada uno de estos conceptos y su papel en la organización del código es fundamental para dominar el arte de la programación.

1.1.5 Orientación práctica: Errores

Python está esperando tu orden. Te animamos a experimentar con el lenguaje, aunque quizás aún no conozcas su vocabulario y estructura completos. Sin embargo, prepárate para cometer errores. Si bien las computadoras son tremendamente rápidas y flexibles, también son extremadamente rígidas. La naturaleza de las computadoras se describe enCurso introductorio de Stanfordcomo

> La ecuación fundamental de las computadoras es:`computer = powerful + stupid`
>
> Las computadoras son muy potentes y pueden analizar grandes cantidades de datos con gran rapidez. Pueden realizar miles de millones de operaciones por segundo, cada una de las cuales es bastante simple.
>
> Los ordenadores también son sorprendentemente estúpidos y frágiles. Las operaciones que pueden realizar son extremadamente rígidas, simples y mecánicas. El ordenador carece de cualquier cosa que se parezca a una percepción real… no se parece en nada al HAL 9000 de las películas. Por lo menos, no deberías dejarte intimidar por el ordenador como si fuera una especie de cerebro. En el fondo, es muy mecánico.

> La programación consiste en que una persona utilice su verdadero conocimiento para construir algo útil, construido a partir de operaciones pequeñas y simples que la computadora puede realizar.
>
> —Francisco Cai y Nick Parlante, Stanford CS101

La rigidez de las computadoras se hará evidente inmediatamente al experimentar con el intérprete de Python: incluso los cambios más pequeños de ortografía y formato provocarán resultados y errores inesperados.

Aprender a interpretar errores y diagnosticar la causa de errores inesperados se llama *depuración* Algunos principios rectores de la depuración son:

1. **Prueba incrementalmente** :Todo programa bien escrito se compone de pequeños componentes modulares que se pueden probar individualmente. Pruebe todo lo que escriba lo antes posible para detectar errores de forma temprana y ganar confianza en sus componentes.
2. **Aislar errores** :Un error en la salida de un programa, expresión o declaración compuesta se puede atribuir normalmente a un componente modular en particular. Al intentar diagnosticar un problema, rastree el error hasta el fragmento de código más pequeño que pueda antes de intentar corregirlo.
3. **Comprueba tus suposiciones** :Los intérpretes cumplen sus instrucciones al pie de la letra, ni más ni menos. Su resultado es inesperado cuando el comportamiento de algún código no coincide con lo que el programador cree (o supone) que es ese comportamiento. Conozca sus suposiciones y luego concentre su esfuerzo de depuración en verificar que sus suposiciones realmente se cumplan.
4. **Consultar a otros** : ¡No estás solo! Si no entiendes un mensaje de error, pregúntale a un amigo, instructor o motor de búsqueda. Si has aislado un error, pero no puedes averiguar cómo corregirlo, pídele a otra persona que lo revise. Se comparte una gran cantidad de conocimientos valiosos sobre programación en el contexto de la resolución de problemas en equipo.

Las pruebas incrementales, el diseño modular, las suposiciones precisas y el trabajo en equipo son temas que persisten a lo largo de este curso. Esperamos que también persistan a lo largo de su carrera en informática.

Contents

1.2 Los elementos de la programación **1**
1.2.1 Expresiones 1
1.2.2 Expresiones de llamada 2
1.2.3 Importación de funciones de biblioteca 3
1.2.4 Nombres y entorno 4
1.2.5 Evaluación de expresiones anidadas 5
1.2.6 Diagramas de funciones 7

\tableofcontents

1.2 Los elementos de la programación

Un lenguaje de programación es más que un simple medio para ordenar a una computadora que realice tareas. El lenguaje también sirve como marco dentro del cual organizamos nuestras ideas sobre los procesos. Los programas sirven para comunicar esas ideas entre los miembros de una comunidad de programación. Por lo tanto, los programas deben escribirse para que los lean las personas y, solo incidentalmente, para que los ejecuten las máquinas.

Cuando describimos un lenguaje, debemos prestar especial atención a los medios que éste proporciona para combinar ideas simples y formar ideas más complejas. Todo lenguaje potente tiene tres mecanismos para lograrlo:

- **Expresiones y afirmaciones primitivas** , que representan los bloques de construcción más simples que proporciona el lenguaje,
- **medios de combinación** , por el cual los elementos compuestos se construyen a partir de otros más simples, y
- **medios de abstracción** , mediante el cual los elementos compuestos pueden nombrarse y manipularse como unidades.

En programación, trabajamos con dos tipos de elementos: funciones y datos (pronto descubriremos que en realidad no son tan distintos). De manera informal, los datos son cosas que queremos manipular y las funciones describen las reglas para manipular los datos. Por lo tanto, cualquier lenguaje de programación potente debería ser capaz de describir datos primitivos y funciones primitivas y debería tener métodos para combinar y abstraer tanto funciones como datos.

1.2.1 Expresiones

Después de experimentar con el intérprete completo de Python, ahora debemos comenzar de nuevo, desarrollando metódicamente el lenguaje Python pieza por pieza. Tenga paciencia si los ejemplos parecen simplistas; pronto habrá material más interesante.

Comenzamos con expresiones primitivas. Un tipo de expresión primitiva es un número. Más precisamente, la expresión que escribes consta de los números que representan el número en base 10.

```
42
```

Las expresiones que representan números pueden combinarse con operadores matemáticos para formar una expresión compuesta, que el intérprete evaluará:

```
-1 - -1
```

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

Estas expresiones matemáticas utilizan *infijo* notación, donde la *operador*(p.ej.,`+`, `-`, `*`, o`/`) aparece entre los *operandos*(números). Python incluye muchas formas de formar expresiones compuestas. En lugar de intentar enumerarlas todas de inmediato, presentaremos nuevas formas de expresión a medida que avanzamos, junto con las características del lenguaje que admiten.

1.2.2 Expresiones de llamada

El tipo más importante de expresión compuesta es una *expresión de llamada* , que aplica una función a algunos argumentos. Recordemos que, en álgebra, la noción matemática de una función es una aplicación de algunos argumentos de entrada a un valor de salida. Por ejemplo, la`max`Una función asigna sus entradas a una única salida, que es la más grande de las entradas. Una función en Python es más que una asignación de entrada-salida; describe un proceso computacional. Sin embargo, la forma en que Python expresa la aplicación de una función es la misma que en matemáticas.

```
max(7.5, 9.5)
```

Esta expresión de llamada tiene subexpresiones: el operador precede a los paréntesis, que encierran una lista de operandos delimitada por comas. El operador debe ser una función. Los operandos pueden ser cualquier valor; en este caso son números. Cuando se evalúa esta expresión de llamada, decimos que la función`max`es *llamado* con los argumentos 7.5 y 9.5, y *devoluciones* un valor de 9,5.

El orden de los argumentos en una expresión de llamada es importante. Por ejemplo, la función`pow`eleva su primer argumento a la potencia de su segundo argumento.

```
pow(100, 2)
```

```
pow(2, 100)
```

La notación de funciones tiene varias ventajas sobre la convención matemática de la notación infija. En primer lugar, las funciones pueden aceptar una cantidad arbitraria de argumentos:

```
max(1, -2, 3, -4)
```

No puede surgir ninguna ambigüedad, porque el nombre de la función siempre precede a sus argumentos.

En segundo lugar, la notación de función se extiende de manera sencilla a *anidado* Expresiones, donde los elementos son en sí mismos expresiones compuestas. En las expresiones de llamada anidadas, a diferencia de las expresiones infijas compuestas, la estructura de la anidación es completamente explícita entre paréntesis.

```
max(min(1, -2), min(pow(3, 5), -4))
```

```
-2
```

En principio, no hay límite para la profundidad de dicha anidación ni para la complejidad general de las expresiones que el intérprete de Python puede evaluar. Sin embargo, los humanos se confunden rápidamente con la anidación de varios niveles. Un papel importante para usted como programador es estructurar las expresiones de modo que sigan siendo interpretables para usted mismo, sus compañeros de programación y otras personas que puedan leer su código en el futuro.

Finalmente, la notación matemática tiene una gran variedad de formas: la multiplicación aparece entre términos, los exponentes aparecen como superíndices, la división como una barra horizontal y una raíz cuadrada como un techo con revestimiento inclinado. ¡Algunas de estas notaciones son muy difíciles de escribir! Sin embargo, toda esta complejidad se puede unificar mediante la notación de expresiones de llamada. Si bien Python admite operadores matemáticos comunes que utilizan notación infija (como`+`y`-`), cualquier operador puede expresarse como una función con un nombre.

1.2.3 Importación de funciones de biblioteca

Python define una gran cantidad de funciones, incluidas las funciones de operador mencionadas en la sección anterior, pero no pone a disposición sus nombres de forma predeterminada, para evitar un caos total. En su lugar, organiza las funciones y otras cantidades que conoce en módulos, que juntos forman la biblioteca de Python. Para utilizar estos elementos, se importan. Por ejemplo, la bibliotecamathEl módulo proporciona una variedad de funciones matemáticas familiares:

```
from math import sqrt, exp
sqrt(256)
```

```
exp(1)
```

y eloperatorEl módulo proporciona acceso a funciones correspondientes a los operadores infijos:

```
-1 - -1
```

```
42
```

```
-1 - -1
```

```
16
```

Un`import`La declaración designa un nombre de módulo (por ejemplo,`operator`o`math`) y luego enumera los atributos nombrados de ese módulo para importar (por ejemplo,`sqrt`o`exp`).

ElDocumentación de la biblioteca de Python 3enumerar las funciones definidas por cada módulo, como por ejemplo:módulo de matemáticasSin embargo, esta documentación está escrita para desarrolladores que conocen bien todo el lenguaje. Por ahora, es posible que descubra que experimentar con una función le brinda más información sobre su comportamiento que leer la documentación. A medida que se familiarice con el lenguaje y el vocabulario de Python, esta documentación se convertirá en una valiosa fuente de referencia.

1.2.4 Nombres y entorno

Un aspecto fundamental de un lenguaje de programación es el medio que proporciona para utilizar nombres para referirse a objetos computacionales. Si se le ha dado un nombre a un valor, decimos que el nombre *se une* al valor.

En Python, podemos establecer nuevos enlaces utilizando la declaración de asignación, que contiene un nombre a la izquierda de`=`y un valor a la derecha:

```
-1 - -1
```

```
10
```

```
-1 - -1
```

```
20
```

Los nombres también están vinculados mediante`import`declaraciones.

```
-1 - -1
```

```
1.0002380197528042
```

También podemos asignar múltiples valores a múltiples nombres en una sola declaración, donde los nombres y las expresiones están separados por comas.

```
-1 - -1
```

```
314.1592653589793
```

Número de serie 16

```
62.83185307179586
```

El`=`El símbolo se llama *asignación* operador en Python (y muchos otros lenguajes). La asignación es el medio más simple de Python para *abstracción* , ya que nos permite utilizar nombres simples para referirnos a los resultados de operaciones compuestas, como por ejemplo`area`calculados anteriormente. De esta

manera, se construyen programas complejos construyendo, paso a paso, objetos computacionales de complejidad creciente.

La posibilidad de vincular nombres a valores y luego recuperar esos valores por nombre significa que el intérprete debe mantener algún tipo de memoria que lleve un registro de los nombres, valores y vinculaciones. Esta memoria se denomina *ambiente*.

Los nombres también pueden estar vinculados a funciones. Por ejemplo, el nombre`max`está ligado a la función máxima que hemos estado usando. Las funciones, a diferencia de los números, son difíciles de representar como texto, por lo que Python imprime una descripción de identificación en su lugar cuando se le pide que imprima una función:

```
-1 - -1
```

Podemos utilizar declaraciones de asignación para dar nuevos nombres a funciones existentes.

Número de serie 18

```
-1 - -1
```

```
4
```

Y las declaraciones de asignación sucesivas pueden volver a vincular un nombre a un nuevo valor.

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
2
```

En Python, los nombres vinculados mediante asignación a menudo se denominan *nombres de variables* porque pueden vincularse a una variedad de valores diferentes durante la ejecución de un programa.

1.2.5 Evaluación de expresiones anidadas

Uno de nuestros objetivos en este capítulo es aislar cuestiones relacionadas con el pensamiento procedimental. Como ejemplo, consideremos que, al evaluar expresiones de llamada anidadas, el intérprete está siguiendo un procedimiento.

Para evaluar una expresión de llamada, Python hará lo siguiente:

1. Evalúe las subexpresiones del operador y del operando, luego
2. Aplique la función que es el valor de la subexpresión del operador a los argumentos que son los valores de las subexpresiones del operando.

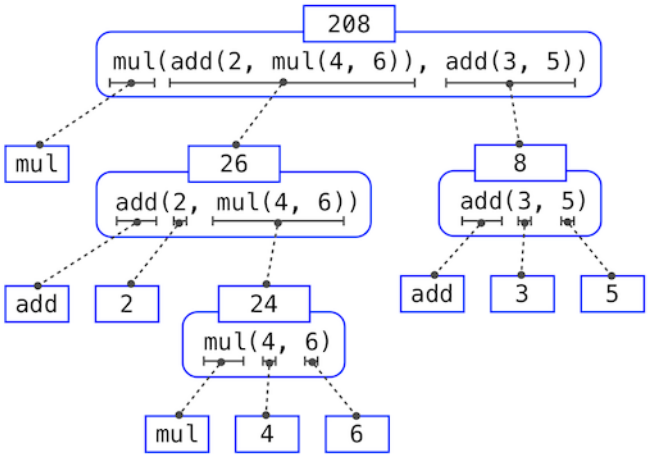
Incluso este procedimiento simple ilustra algunos puntos importantes sobre los procesos en general. El primer paso dicta que para llevar a cabo el proceso de evaluación de una expresión de llamada, primero debemos evaluar otras expresiones. Por lo tanto, el procedimiento de evaluación es *recursivo* en la naturaleza; es decir, incluye, como uno de sus pasos, la necesidad de invocar la norma misma.

Por ejemplo, evaluar

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
208
```

requiere que este procedimiento de evaluación se aplique cuatro veces. Si dibujamos cada expresión que evaluamos, podemos visualizar la estructura jerárquica de este proceso.



Esta ilustración se llama *árbol de expresión* En informática, los árboles crecen de arriba hacia abajo. Los objetos en cada punto de un árbol se denominan nodos; en este caso, son expresiones emparejadas con sus valores.

Para evaluar su raíz, la expresión completa, primero es necesario evaluar las ramas que son sus subexpresiones. Las expresiones hoja (es decir, los nodos sin ramas que se deriven de ellos) representan funciones o números. Los nodos interiores tienen dos partes: la expresión de llamada a la que se aplica nuestra regla de evaluación y el resultado de esa expresión. Si consideramos la evaluación en términos de este árbol, podemos imaginar que los valores de los operandos se filtran hacia arriba, comenzando desde los nodos terminales y luego combinándose en niveles cada vez más altos.

A continuación, observe que la aplicación repetida del primer paso nos lleva al punto en el que necesitamos evaluar, no expresiones de llamada, sino expresiones primitivas como numerales (por ejemplo,`2`) y nombres (por ejemplo,`add`). Nos ocupamos de los casos primitivos estipulando que

- Un numeral se evalúa como el número que nombra,
- Un nombre se evalúa como el valor asociado con ese nombre en el entorno actual.

Observe el importante papel que desempeña un entorno a la hora de determinar el significado de los símbolos en las expresiones. En Python, no tiene sentido

hablar del valor de una expresión como

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

Sin especificar ninguna información sobre el entorno que proporcione un significado al nombre.`x`(o incluso por el nombre`add`). Los entornos proporcionan el contexto en el que se lleva a cabo la evaluación, lo que juega un papel importante en nuestra comprensión de la ejecución del programa.

Este procedimiento de evaluación no es suficiente para evaluar todo el código Python, solo expresiones de llamada, numerales y nombres. Por ejemplo, no maneja sentencias de asignación. Ejecutar

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

no devuelve un valor ni evalúa una función en algunos argumentos, ya que el propósito de la asignación es, en cambio, vincular un nombre a un valor. En general, las declaraciones no se evalúan, sino que se *ejecutado* ; no producen un valor sino que realizan algún cambio. Cada tipo de declaración o expresión tiene su propio procedimiento de evaluación o ejecución, que iremos introduciendo de forma incremental a medida que avancemos.

Una nota pedante: cuando decimos que "un numeral se evalúa como un número", en realidad queremos decir que el intérprete de Python evalúa un numeral como un número. Es el intérprete el que le otorga significado al lenguaje de programación. Dado que el intérprete es un programa fijo que siempre se comporta de manera consistente, podemos decir libremente que los numerales (y las expresiones) se evalúan como valores en el contexto de los programas Python.

1.2.6 Diagramas de funciones

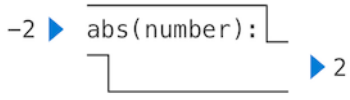
A medida que sigamos desarrollando un modelo formal de evaluación, descubriremos que la representación del estado interno del intérprete nos ayuda a seguir el progreso de nuestro procedimiento de evaluación. Una parte esencial de estos diagramas es la representación de una función.

Funciones puras. Las funciones tienen una entrada (sus argumentos) y devuelven una salida (el resultado de su aplicación). La función incorporada

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
2
```

Puede representarse como una pequeña máquina que toma una entrada y produce una salida.



La función`abs`es *puro* Las funciones puras tienen la propiedad de que su aplicación no tiene efectos más allá de devolver un valor.

Funciones no puras. Además de devolver un valor, la aplicación de una función no pura puede generar *efectos secundarios* , que realizan algún cambio en el estado del intérprete o la computadora. Un efecto secundario común es generar una salida adicional más allá del valor de retorno, utilizando el`print`función.

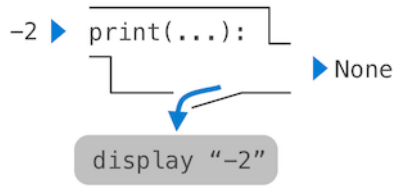
```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
-2
```

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
1 2 3
```

Mientras`print`y`abs`pueden parecer similares en estos ejemplos, pero funcionan de maneras fundamentalmente diferentes. El valor que`print`el retorno es siempre`None`, un valor especial de Python que no representa nada. El intérprete interactivo de Python no imprime automáticamente el valor`None`. En el caso de`print`, la función en sí misma imprime la salida como un efecto secundario de su llamada.



Una expresión anidada de llamadas a`print`destaca el carácter no puro de la función.

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
1
2
None None
```

Si considera que este resultado es inesperado, dibuje un árbol de expresiones para aclarar por qué la evaluación de esta expresión produce este resultado peculiar.

Ten cuidado con`print`! El hecho de que vuelva`None`significa que *No debería* sea la expresión en una declaración de asignación.

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
2
```

```
1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
```

```
None
```

Firmas. Las funciones difieren en la cantidad de argumentos que pueden aceptar. Para hacer un seguimiento de estos requisitos, dibujamos cada función de

manera que se muestre el nombre de la función y los nombres de sus argumentos. La función`abs`solo toma un argumento llamado`number`; si se proporciona más o menos, se producirá un error. La función`print`puede tomar un número arbitrario de argumentos, de ahí su representación como`print(...)`Una descripción de los argumentos que puede tomar una función se denomina función._firma__.

Contents

1.3 Definición de nuevas funciones **1**
1.3.1 Entornos 2
1.3.2 Llamada a funciones definidas por el usuario 4
1.3.3 Ejemplo: Llamar a una función definida por el usuario 6
1.3.4 Nombres locales 9
1.3.5 Orientación práctica: elección de nombres 9
1.3.6 Funciones como abstracciones 10
1.3.7 Operadores 11

\tableofcontents

1.3 Definición de nuevas funciones

Hemos identificado en Python algunos de los elementos que deben aparecer en cualquier lenguaje de programación potente:

1. Los números y las operaciones aritméticas son datos y funciones integrados.
2. La aplicación de funciones anidadas proporciona un medio para combinar operaciones.
3. Vincular nombres a valores proporciona un medio limitado de abstracción.

Ahora aprenderemos sobre *definiciones de funciones* , una técnica de abstracción mucho más poderosa mediante la cual un nombre puede vincularse a una operación compuesta, a la que luego se puede hacer referencia como una unidad.

Comenzaremos examinando cómo expresar la idea de “elevar al cuadrado”. Podríamos decir: “Para elevar al cuadrado algo, multiplícalo por sí mismo”. Esto se expresa en Python como

```
def square(x):
    return mul(x, x)
```

que define una nueva función a la que se le ha dado el nombre`square`Esta función definida por el usuario no está incorporada en el intérprete. Representa la operación compuesta de multiplicar algo por sí mismo.`x`En esta definición se llama *parámetro formal* , que proporciona un nombre para el elemento que se va a multiplicar. La definición crea esta función definida por el usuario y la asocia con el nombre`square`.

Las definiciones de funciones consisten en una`def`declaración que indica una`<name>`y una lista de nombres`<formal parameters>`, entonces un`return`declaración, llamada cuerpo de la función, que especifica la`<return expression>`de la función, que es una expresión que se evaluará siempre que se aplique la función.

```
def <name>(<formal parameters>):
return <return expression>
```

La segunda linea *debe* ¡Se debe sangrar! La convención dicta que se deben sangrar cuatro espacios en lugar de una tabulación. La expresión de retorno no se evalúa de inmediato; se almacena como parte de la función recién definida y se evalúa solo cuando la función se aplica finalmente. (Pronto veremos que la región sangrada puede abarcar varias líneas).

Habiendo definido`square`, podemos aplicarlo con una expresión de llamada:

```
square(21)
```

```
square(add(2, 5))
```

```
square(square(3))
```

También podemos utilizar`square`como elemento básico para definir otras funciones. Por ejemplo, podemos definir fácilmente una función`sum_squares`que, dados dos números como argumentos, devuelve la suma de sus cuadrados:

```
def sum_squares(x, y):
    return add(square(x), square(y))
```

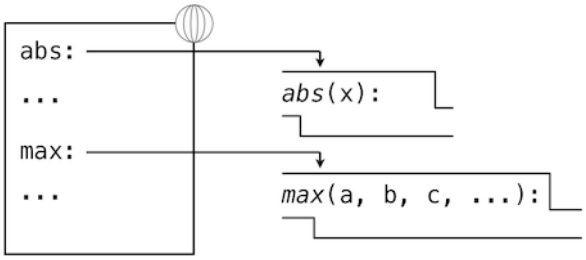
```
sum_squares(3, 4)
```

Las funciones definidas por el usuario se utilizan exactamente de la misma manera que las funciones integradas. De hecho, no se puede saber a partir de la definición de`sum_squares`si`square`está integrado en el intérprete, importado desde un módulo o definido por el usuario.

1.3.1 Entornos

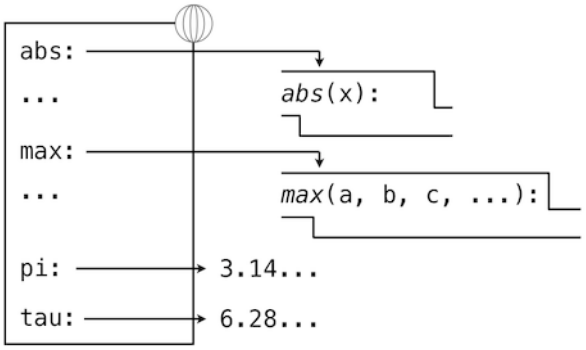
Nuestro subconjunto de Python es ahora lo suficientemente complejo como para que el significado de los programas no sea obvio. ¿Qué sucede si un parámetro formal tiene el mismo nombre que una función incorporada? ¿Pueden dos funciones compartir nombres sin que se produzcan confusiones? Para resolver estas preguntas, debemos describir los entornos con más detalle.

Un entorno en el que se evalúa una expresión consiste en una secuencia de *Marcos* , representados como cuadros. Cada cuadro contiene *fijaciones* , que asocian un nombre con su valor correspondiente. Hay un único *global* marco que contiene enlaces de nombres para todas las funciones integradas (solo`abs`y`max`Se muestran). Indicamos el marco global con un símbolo de globo.

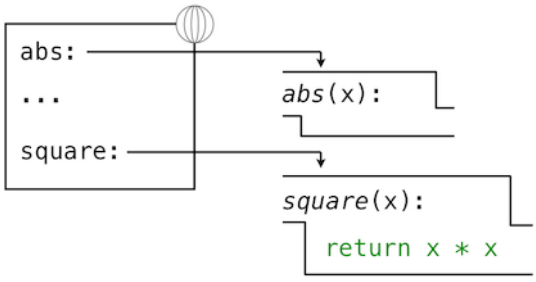


Las instrucciones de asignación e importación agregan entradas al primer marco del entorno actual. Hasta ahora, nuestro entorno consta únicamente del marco global.

```
from math import pi
tau = 2 * pi
```



AdefLa declaración también vincula un nombre a la función creada por la definición. El entorno resultante después de definir`square`aparece a continuación:



Estos *Diagramas de entorno* muestra los enlaces del entorno actual, junto con los valores (que no forman parte de ningún marco) a los que están enlazados los nombres. Observe que el nombre de una función se repite, una vez en el marco y una vez como parte de la función misma. Esta repetición es intencional: muchos nombres diferentes pueden referirse a la misma función, pero esa función en sí misma tiene solo un nombre intrínseco. Sin embargo, buscar el valor de un nombre en un entorno solo inspecciona los enlaces de nombres. El nombre

intrínseco de una función**no lo hace** desempeñan un papel en la búsqueda de nombres. En el ejemplo que vimos antes,

```
f = max
f
```

El nombre *máximo* es el nombre intrínseco de la función, y eso es lo que ves impreso como valor para`f`. Además, ambos nombres`max`y`f`están ligados a esa misma función en el entorno global.

A medida que vayamos introduciendo características adicionales de Python, tendremos que ampliar estos diagramas. Cada vez que lo hagamos, enumeraremos las nuevas características que pueden expresar nuestros diagramas.

Nuevas características del entorno: Asignación y definición de funciones definidas por el usuario.

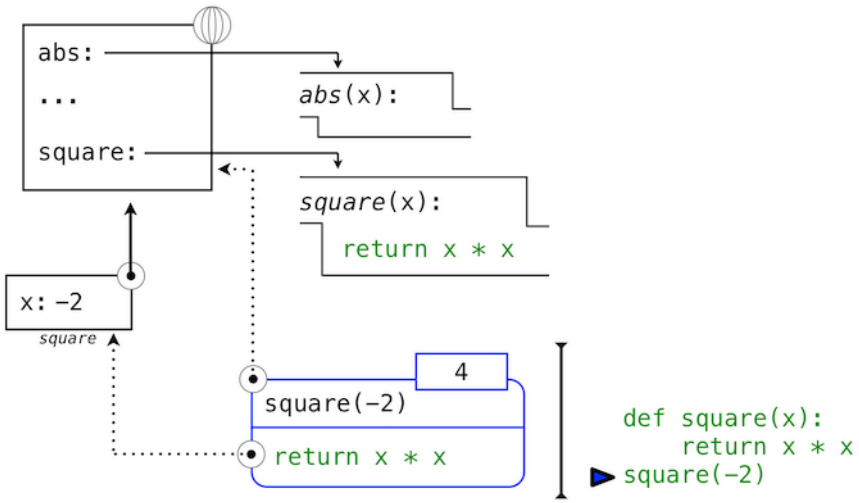
1.3.2 Llamada a funciones definidas por el usuario

Para evaluar una expresión de llamada cuyo operador nombra una función definida por el usuario, el intérprete de Python sigue un proceso similar al utilizado para evaluar expresiones con una función de operador incorporada. Es decir, el intérprete evalúa las expresiones de operandos y luego aplica la función nombrada a los argumentos resultantes.

El acto de aplicar una función definida por el usuario introduce una segunda *local* marco, al que solo puede acceder esa función. Para aplicar una función definida por el usuario a algunos argumentos:

1. Vincula los argumentos a los nombres de los parámetros formales de la función en una nueva *local* marco.
2. Evalúa el cuerpo de la función en el entorno comenzando en ese marco y terminando en el marco global.

El entorno en el que se evalúa el cuerpo consta de dos marcos: primero, el marco local que contiene los enlaces de argumentos y, luego, el marco global que contiene todo lo demás. Cada instancia de una aplicación de función tiene su propio marco local independiente.



Esta figura incluye dos aspectos diferentes del intérprete de Python: el entorno actual y una parte del árbol de expresiones relacionado con la línea de código actual que se está evaluando. Hemos representado la evaluación de una expresión de llamada que tiene una función definida por el usuario (en azul) como un rectángulo redondeado de dos partes. Las flechas punteadas indican qué entorno se utiliza para evaluar la expresión en cada parte.

- La mitad superior muestra la expresión de llamada que se está evaluando. Esta expresión de llamada no es interna a ninguna función, por lo que se evalúa en el entorno global. Por lo tanto, cualquier nombre dentro de ella (como`square`) se buscan en el marco global.
- La mitad inferior muestra el cuerpo de la`square`función. Su expresión de retorno se evalúa en el nuevo entorno introducido en el paso 1 anterior, que vincula el nombre de`square`parámetro formal de`x`al valor de su argumento,`-2`.

El orden de los marcos en un entorno afecta el valor que se obtiene al buscar un nombre en una expresión. Ya hemos indicado que un nombre se evalúa como el valor asociado a ese nombre en el entorno actual. Ahora podemos ser más precisos:

- Un nombre se evalúa como el valor vinculado a ese nombre en el primer cuadro del entorno actual en el que se encuentra ese nombre.

Nuestro marco conceptual de entornos, nombres y funciones constituye un *modelo de evaluación* ; aunque algunos detalles mecánicos aún no están especificados (por ejemplo, cómo se implementa un enlace), nuestro modelo describe de manera precisa y correcta cómo el intérprete evalúa las expresiones de llamada. En el Capítulo 3 veremos cómo este modelo puede servir como modelo para implementar un intérprete funcional para un lenguaje de programación.

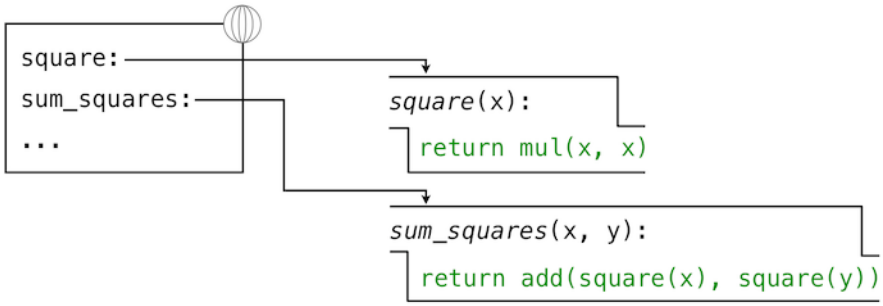
Nueva característica del entorno: Aplicación de función.

1.3.3 Ejemplo: Llamar a una función definida por el usuario

Consideremos nuevamente nuestras dos definiciones simples:

```
from operator import add, mul
def square(x):
    return mul(x, x)

def sum_squares(x, y):
    return add(square(x), square(y))
```



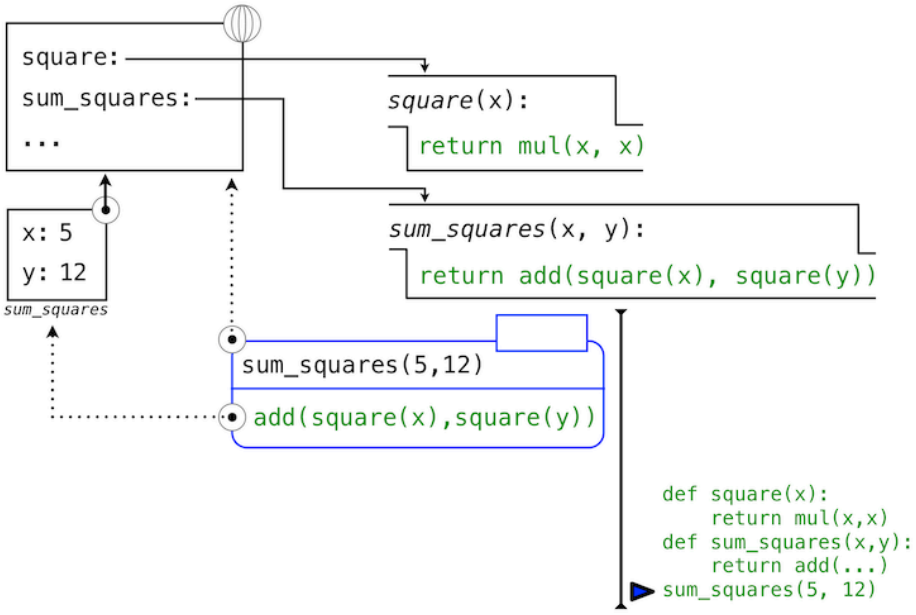
Y el proceso que evalúa la siguiente expresión de llamada:

```
square(21)
```

```
169
```

Python primero evalúa el nombre`sum_squares`, que está vinculada a una función definida por el usuario en el marco global. Las expresiones numéricas primitivas 5 y 12 se evalúan como los números que representan.

A continuación, Python aplica`sum_squares`, que introduce un marco local que vincula x a 5 e y a 12.



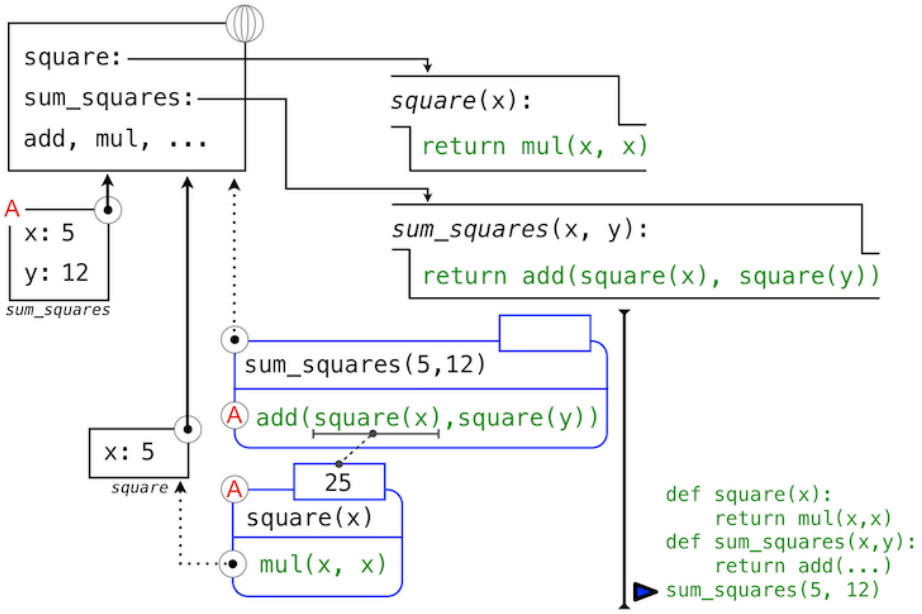
En este diagrama, el marco local apunta a su sucesor, el marco global. Todos los marcos locales deben apuntar a un predecesor y estos vínculos definen la secuencia de marcos que constituye el entorno actual.

El cuerpo de`sum_squares`contiene esta expresión de llamada:

```
square(21)
```

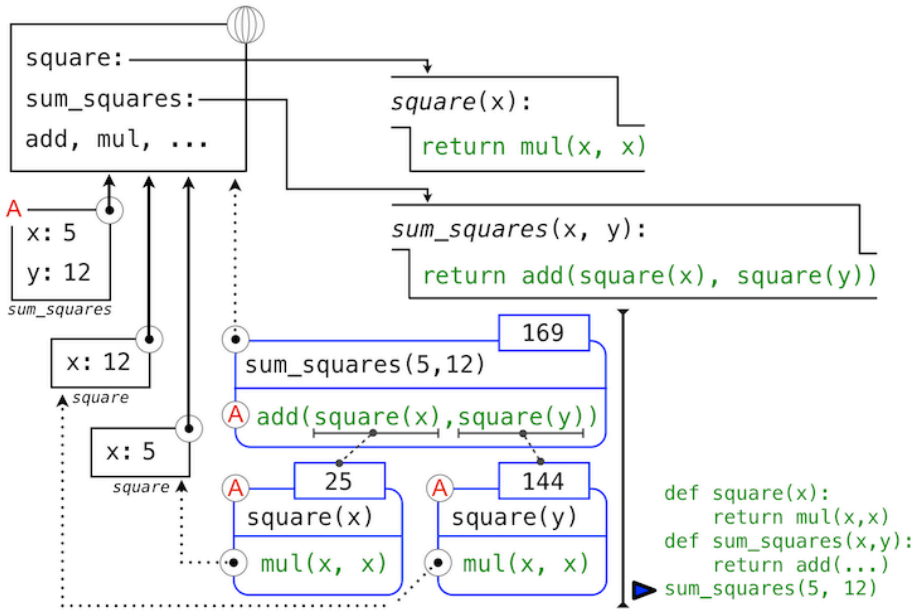
Las tres subexpresiones se evalúan en el entorno actual, que comienza con el marco etiquetado *suma_cuadrados*. La subexpresión del operador`add`es un nombre que se encuentra en el marco global, vinculado a la función incorporada para la suma. Las dos subexpresiones de operandos deben evaluarse a su vez, antes de aplicar la suma. Ambos operandos se evalúan en el entorno actual comenzando con el marco etiquetado`sum_squares`En los siguientes diagramas de entorno, llamaremos a este marco`A`y reemplace las flechas que apuntan a este marco con la etiqueta`A`también.

En`operand 0`, `square`nombra una función definida por el usuario en el marco global, mientras que`x`nombra el número 5 en el marco local. Python aplica`square`a 5 introduciendo otro marco local que une x a 5.



Utilizando este marco local, la expresión corporal`mul(x, x)`evalúa a 25.

Nuestro procedimiento de evaluación ahora se centra en`operand 1`, para lo cualynombra el número 12. Python evalúa el cuerpo de`square`De nuevo, esta vez introduciendo otro marco de entorno local que unexhasta 12. Por lo tanto,`operand 1`evalúa a 144.



Finalmente, al aplicar la adición a los argumentos 25 y 144 se obtiene un valor final para el cuerpo de`sum_squares`:169.

Esta figura, aunque compleja, sirve para ilustrar muchas de las ideas fundamentales que hemos desarrollado hasta ahora. Los nombres están ligados a valores, que se distribuyen en muchos marcos locales que preceden a un único marco global que contiene nombres compartidos. Las expresiones tienen una estructura de árbol y el entorno debe ampliarse cada vez que una subexpresión contiene una llamada a una función definida por el usuario.

Todo este mecanismo existe para garantizar que los nombres se resuelvan en los valores correctos en los puntos correctos del árbol de expresión. Este ejemplo ilustra por qué nuestro modelo requiere la complejidad que hemos introducido. Los tres marcos locales contienen un enlace para el nombre`x`, pero ese nombre está vinculado a distintos valores en distintos marcos. Los marcos locales mantienen separados estos nombres.

1.3.4 Nombres locales

Un detalle de la implementación de una función que no debería afectar el comportamiento de la misma es la elección de los nombres que hace el implementador para los parámetros formales de la función. Por lo tanto, las siguientes funciones deberían proporcionar el mismo comportamiento:

```
square(21)
```

Este principio (que establece que el significado de una función debe ser independiente de los nombres de los parámetros elegidos por su autor) tiene consecuencias importantes para los lenguajes de programación. La consecuencia más simple es que los nombres de los parámetros de una función deben permanecer locales en el cuerpo de la función.

Si los parámetros no fueran locales a los cuerpos de sus respectivas funciones, entonces el parámetro`x`en`square`Podría confundirse con el parámetro`x`en`sum_squares`. Críticamente, este no es el caso: la vinculación para`x`en diferentes marcos locales no están relacionados. Nuestro modelo de cálculo está cuidadosamente diseñado para garantizar esta independencia.

Decimos que el *alcance* El alcance de un nombre local se limita al cuerpo de la función definida por el usuario que lo define. Cuando un nombre ya no es accesible, queda fuera del alcance. Este comportamiento de alcance no es un hecho nuevo en nuestro modelo; es una consecuencia de la forma en que funcionan los entornos.

1.3.5 Orientación práctica: elección de nombres

La intercambiabilidad de nombres no implica que los nombres de parámetros formales no importen en absoluto. Por el contrario, los nombres de funciones

y parámetros bien elegidos son esenciales para la interpretación humana de las definiciones de funciones.

Las siguientes pautas son adaptaciones de lasGuía de estilo para código Python, que sirve como guía para todos los programadores de Python (no rebeldes). Un conjunto compartido de convenciones facilita la comunicación entre los miembros de una comunidad de programación. Como efecto secundario de seguir estas convenciones, descubrirá que su código se vuelve más coherente internamente.

1. Los nombres de las funciones deben escribirse en minúsculas y las palabras deben separarse con guiones bajos. Se recomienda utilizar nombres descriptivos.
2. Los nombres de funciones generalmente evocan operaciones aplicadas a los argumentos por el intérprete (por ejemplo,`print`, `add`, `square`) o el nombre de la cantidad resultante (por ejemplo,`max`, `abs`, `sum`).
3. Los nombres de los parámetros deben estar en minúsculas y las palabras deben estar separadas por guiones bajos. Se prefieren nombres de una sola palabra.
4. Los nombres de los parámetros deben evocar el rol del parámetro en la función, no solo el tipo de valor permitido.
5. Los nombres de parámetros de una sola letra son aceptables cuando su función es obvia, pero nunca use "l" (letra minúscula), "O" (letra mayúscula) o "I" (letra i mayúscula) para evitar confusiones con números.

Revise estas pautas periódicamente mientras escribe programas y pronto sus nombres serán deliciosamente pitónicos.

1.3.6 Funciones como abstracciones

Aunque es muy simple,`sum_squares`ejemplifica la propiedad más poderosa de las funciones definidas por el usuario. La función`sum_squares`se define en términos de la función`square`, pero se basa únicamente en la relación que`square`define entre sus argumentos de entrada y sus valores de salida.

Podemos escribir`sum_squares`Sin preocuparnos por *cómo* elevar al cuadrado un número. Los detalles de cómo se calcula el cuadrado se pueden suprimir, para ser considerados en un momento posterior. De hecho, en lo que respecta a`sum_squares`se preocupa,`square`no es un cuerpo de función particular, sino más bien una abstracción de una función, una llamada abstracción funcional. En este nivel de abstracción, cualquier función que calcule el cuadrado es igualmente buena.

Por lo tanto, considerando únicamente los valores que devuelven, las dos funciones siguientes para elevar al cuadrado un número deberían ser indistinguibles. Cada una toma un argumento numérico y produce el cuadrado de ese número como valor.

```
square(21)
```

En otras palabras, la definición de una función debería ser capaz de suprimir detalles. Es posible que los usuarios de la función no hayan escrito la función ellos mismos, sino que la hayan obtenido de otro programador como una "caja negra". Un usuario no debería necesitar saber cómo se implementa la función para poder usarla. La biblioteca Python tiene esta propiedad. Muchos desarrolladores usan las funciones definidas allí, pero pocos inspeccionan su implementación. De hecho, muchas implementaciones de funciones de la biblioteca Python no están escritas en Python en absoluto, sino en el lenguaje C.

1.3.7 Operadores

Los operadores matemáticos (como + y -) proporcionaron nuestro primer ejemplo de un método de combinación, pero aún tenemos que definir un procedimiento de evaluación para expresiones que contienen estos operadores.

Las expresiones de Python con operadores infijos tienen sus propios procedimientos de evaluación, pero a menudo se puede pensar en ellas como una forma abreviada de expresiones de llamada. Cuando vea

```
square(21)
```

```
5
```

simplemente considérelo como una forma abreviada de

```
square(21)
```

```
5
```

La notación infija se puede anidar, al igual que las expresiones de llamada. Python aplica las reglas matemáticas normales de precedencia de operadores, que dictan cómo interpretar una expresión compuesta con múltiples operadores.

Número de serie 16

```
19
```

evalúa al mismo resultado que

```
square(21)
```

```
19
```

La anidación en la expresión de llamada es más explícita que en la versión del operador. Python también permite la agrupación de subexpresiones con paréntesis para anular las reglas de precedencia normales o hacer que la estructura anidada de una expresión sea más explícita.

Número de serie 18

```
45
```

evalúa al mismo resultado que

```
square(21)
```

```
45
```

Puede utilizar libremente estos operadores y paréntesis en sus programas. Python idiomático prefiere los operadores a las expresiones de llamada para operaciones matemáticas simples.

Contents

1.4 Orientación práctica: El arte de la función **1**
1.4.1 Cadenas de documentación 1
1.4.2 Valores de argumentos predeterminados 2

\tableofcontents

1.4 Orientación práctica: El arte de la función

Las funciones son un ingrediente esencial de todos los programas, grandes y pequeños, y sirven como nuestro medio principal para expresar procesos computacionales en un lenguaje de programación. Hasta ahora, hemos analizado las propiedades formales de las funciones y cómo se aplican. Ahora abordaremos el tema de qué hace que una función sea buena. Fundamentalmente, las cualidades de las buenas funciones refuerzan la idea de que las funciones son abstracciones.

- Cada función debe tener exactamente un trabajo. Ese trabajo debe ser identificable con un nombre corto y caracterizable en una sola línea de texto. Las funciones que realizan varios trabajos en secuencia deben dividirse en varias funciones.
- *No te repitas* es un principio central de la ingeniería de software. El llamado principio DRY establece que varios fragmentos de código no deben describir una lógica redundante. En cambio, esa lógica debe implementarse una vez, dársele un nombre y aplicarse varias veces. Si se encuentra copiando y pegando un bloque de código, probablemente haya encontrado una oportunidad para la abstracción funcional.
- Las funciones deben definirse de manera general. El cuadrado no está en la biblioteca de Python precisamente porque es un caso especial de la`pow`función que eleva números a potencias arbitrarias.

Estas pautas mejoran la legibilidad del código, reducen la cantidad de errores y, a menudo, minimizan la cantidad total de código escrito. Descomponer una tarea compleja en funciones concisas es una habilidad que requiere experiencia para dominarla. Afortunadamente, Python ofrece varias funciones que respaldan sus esfuerzos.

1.4.1 Cadenas de documentación

Una definición de función a menudo incluirá documentación que describe la función, denominada *cadena de documentación* , que debe sangrarse junto con el cuerpo de la función. Las cadenas de documentación se escriben convencionalmente entre comillas triples. La primera línea describe el trabajo de la función en una sola línea. Las siguientes líneas pueden describir argumentos y aclarar el comportamiento de la función:

```
>>> def pressure(v, t, n):
```

```
        """Compute the pressure in pascals of an ideal gas.

        Applies the ideal gas law: http://en.wikipedia.org/wiki/Ideal_gas_law

        v -- volume of gas, in cubic meters
        t -- absolute temperature in degrees kelvin
        n -- particles of gas
        """
        k = 1.38e-23  # Boltzmann's constant
        return n * k * t / v
```

Cuando llamas`help`con el nombre de una función como argumento, verá su cadena de documentación (tipo`q`para salir de la ayuda de Python).

```
help(pressure)
```

Al escribir programas Python, incluya cadenas de documentación para todas las funciones, excepto las más simples. Recuerde que el código se escribe solo una vez, pero a menudo se lee muchas veces. La documentación de Python incluyePautas de la cadena de documentaciónque mantienen la coherencia entre diferentes proyectos de Python.

1.4.2 Valores de argumentos predeterminados

Una consecuencia de definir funciones generales es la introducción de argumentos adicionales. Las funciones con muchos argumentos pueden resultar difíciles de llamar y de leer.

En Python, podemos proporcionar valores predeterminados para los argumentos de una función. Al llamar a esa función, los argumentos con valores predeterminados son opcionales. Si no se proporcionan, el valor predeterminado se vincula al nombre del parámetro formal. Por ejemplo, si una aplicación calcula comúnmente la presión para un mol de partículas, este valor se puede proporcionar como predeterminado:

```
>>> k_b=1.38e-23  # Boltzmann's constant
>>> def pressure(v, t, n=6.022e23):
        """Compute the pressure in pascals of an ideal gas.

        v -- volume of gas, in cubic meters
        t -- absolute temperature in degrees kelvin
        n -- particles of gas (default: one mole)
        """
        return n * k_b * t / v

pressure(1, 273.15)
```

Aquí,`pressure`está definido para tomar tres argumentos, pero solo se proporcionan dos en la expresión de llamada que sigue. En este caso, el valor den se toma

de la`def`valores predeterminados de la declaración (que parecen una asignación a`n`, aunque como sugiere esta discusión, se trata más bien de una asignación condicional).

Como guía, la mayoría de los valores de datos utilizados en el cuerpo de una función deben expresarse como valores predeterminados para argumentos nombrados, de modo que sean fáciles de inspeccionar y puedan ser modificados por el llamador de la función. Algunos valores que nunca cambian, como la constante fundamental`k_b`, se puede definir en el marco global.

Contents

1.5 Control **1**
1.5.1 Declaraciones 1
1.5.2 Declaraciones compuestas 2
1.5.3 Definición de funciones II: Asignación local 3
1.5.4 Declaraciones condicionales 4
1.5.5 Iteración 7
1.5.6 Orientación práctica: Pruebas 8

\tableofcontents

1.5 Control

El poder expresivo de las funciones que podemos definir en este punto es muy limitado, porque no hemos introducido una forma de hacer pruebas y realizar diferentes operaciones en función del resultado de una prueba._Declaraciones de control_ nos dará esta capacidad. Las sentencias de control difieren fundamentalmente de las expresiones que hemos estudiado hasta ahora. Se desvían de la evaluación estricta de subexpresiones de izquierda a derecha, y reciben su nombre del hecho de que controlan lo que el intérprete debe hacer a continuación, posiblemente en función de los valores de las expresiones.

1.5.1 Declaraciones

Hasta ahora, hemos considerado principalmente cómo evaluar expresiones. Sin embargo, hemos visto tres tipos de declaraciones: asignación,`def`, y`return`declaraciones. Estas líneas de código Python no son en sí mismas expresiones, aunque todas contienen expresiones como componentes.

Para enfatizar que el valor de una declaración es irrelevante (o inexistente), describimos las declaraciones como *ejecutado* en lugar de evaluarse. Cada declaración describe algún cambio en el estado del intérprete y la ejecución de una declaración aplica ese cambio. Como hemos visto para`return`y declaraciones de asignación, la ejecución de declaraciones puede implicar la evaluación de subexpresiones contenidas en ellas.

Las expresiones también se pueden ejecutar como sentencias, en cuyo caso se evalúan, pero su valor se descarta. La ejecución de una función pura no tiene ningún efecto, pero la ejecución de una función no pura puede causar efectos como consecuencia de la aplicación de la función.

Consideremos, por ejemplo,

```
def square(x):
    mul(x, x) # Watch out! This call doesn't return a value.
```

Esto es válido en Python, pero probablemente no sea lo que se pretendía. El cuerpo de la función consiste en una expresión. Una expresión por sí misma es una declaración válida, pero el efecto de la declaración es que la`mul`Se llama a la función y se descarta el resultado. Si desea hacer algo con el resultado de una expresión, debe indicarlo: puede almacenarlo con una declaración de asignación o devolverlo con una declaración de retorno:

```
def square(x):
    return mul(x, x)
```

A veces tiene sentido tener una función cuyo cuerpo sea una expresión, cuando una función no pura como`print`se llama.

```
def print_square(x):
    print(square(x))
```

En su nivel más alto, el trabajo del intérprete de Python es ejecutar programas compuestos de sentencias. Sin embargo, gran parte del trabajo interesante de computación proviene de la evaluación de expresiones. Las sentencias gobiernan la relación entre las diferentes expresiones en un programa y lo que sucede con sus resultados.

1.5.2 Declaraciones compuestas

En general, el código Python es una secuencia de sentencias. Una sentencia simple es una sola línea que no termina en dos puntos. Una sentencia compuesta se llama así porque está compuesta de otras sentencias (simples y compuestas). Las sentencias compuestas suelen abarcar varias líneas y comienzan con un encabezado de una línea que termina en dos puntos, que identifica el tipo de sentencia. Juntos, un encabezado y un conjunto de sentencias con sangría se denominan cláusula. Una sentencia compuesta consta de una o más cláusulas:

```
<header>:
    <statement>
    <statement>
    ...
<separating header>:
    <statement>
    <statement>
    ...
...
```

Podemos entender las afirmaciones que ya hemos introducido en estos términos.

- Las expresiones, declaraciones de retorno y declaraciones de asignación son declaraciones simples.
- `A`La declaración `def` es una declaración compuesta. La suite que sigue a la`def`El encabezado define el cuerpo de la función.

Las reglas de evaluación especializadas para cada tipo de encabezado dictan cuándo y si se ejecutan las instrucciones de su conjunto. Decimos que el encabezado controla su conjunto. Por ejemplo, en el caso de`def`En las declaraciones, vimos que la expresión de retorno no se evalúa inmediatamente, sino que se almacena para su uso posterior cuando finalmente se aplica la función definida.

Ahora también podemos entender programas multilínea.

- Para ejecutar una secuencia de instrucciones, ejecute la primera instrucción. Si esa instrucción no redirige el control, proceda a ejecutar el resto de la secuencia de instrucciones, si queda alguna.

Esta definición expone la estructura esencial de una definición recursiva._secuencia_ : una secuencia puede descomponerse en su primer elemento y el resto de sus elementos. ¡El "resto" de una secuencia de sentencias es en sí mismo una secuencia de sentencias! Por lo tanto, podemos aplicar recursivamente esta regla de ejecución. Esta visión de las secuencias como estructuras de datos recursivas aparecerá nuevamente en capítulos posteriores.

La consecuencia importante de esta regla es que las instrucciones se ejecutan en orden, pero es posible que nunca se llegue a las instrucciones posteriores debido al control redirigido.

Orientación práctica. Al sangrar un conjunto, todas las líneas deben sangrarse con la misma cantidad y de la misma manera (espacios, no tabulaciones). Cualquier variación en la sangría provocará un error.

1.5.3 Definición de funciones II: Asignación local

Originalmente, dijimos que el cuerpo de una función definida por el usuario consistía únicamente en una`return`Declaración con una única expresión de retorno. De hecho, las funciones pueden definir una secuencia de operaciones que se extiende más allá de una única expresión. La estructura de las declaraciones compuestas de Python nos permite naturalmente extender nuestro concepto de cuerpo de función a múltiples declaraciones.

Siempre que se aplica una función definida por el usuario, la secuencia de cláusulas en el conjunto de su definición se ejecuta en un entorno local.`return`La declaración redirige el control: el proceso de aplicación de la función termina cuando la primera`return`Se ejecuta la declaración y el valor de la`return`expresión es el valor devuelto de la función que se está aplicando.

De esta manera, las instrucciones de asignación ahora pueden aparecer dentro del cuerpo de una función. Por ejemplo, esta función devuelve la diferencia absoluta entre dos cantidades como porcentaje de la primera, mediante un cálculo de dos pasos:

```
def percent_difference(x, y):
    difference = abs(x-y)
```

```
    return 100 * difference / x
percent_difference(40, 50)
```

El efecto de una declaración de asignación es vincular un nombre a un valor en la *primero* marco del entorno actual. Como consecuencia, las instrucciones de asignación dentro del cuerpo de una función no pueden afectar el marco global. El hecho de que las funciones solo puedan manipular su entorno local es fundamental para crear *modular* programas en los que las funciones puras interactúan únicamente a través de los valores que toman y devuelven.

Por supuesto, el`percent_difference`La función podría escribirse como una única expresión, como se muestra a continuación, pero la expresión de retorno es más compleja.

```
def percent_difference(x, y):
    return 100 * abs(x-y) / x
```

Hasta ahora, la asignación local no ha aumentado el poder expresivo de nuestras definiciones de funciones. Lo hará cuando se combine con las instrucciones de control que se indican a continuación. Además, la asignación local también desempeña un papel fundamental a la hora de aclarar el significado de expresiones complejas al asignar nombres a cantidades intermedias.

Nueva característica del entorno: Asignación local.

1.5.4 Declaraciones condicionales

Python tiene una función incorporada para calcular valores absolutos.

```
abs(-2)
```

Nos gustaría poder implementar una función de este tipo nosotros mismos, pero actualmente no podemos definir una función que tenga una prueba y una opción. Nos gustaría expresar que sixes positivo,`abs(x)`devolucionesxAdemás, sixes 0,`abs(x)`devuelve 0. De lo contrario,`abs(x)`devoluciones-xEn Python, podemos expresar esta elección con una declaración condicional.

```
def absolute_value(x):
    """Compute abs(x)."""
    if x > 0:
        return x
    elif x == 0:
        return 0
    else:
        return -x

absolute_value(-2) == abs(-2)
```

```
True
```

Esta implementación de`absolute_value`plantea varias cuestiones importantes.

Declaraciones condicionales Una declaración condicional en Python consta de una serie de encabezados y conjuntos: un requerido`if`cláusula, una secuencia opcional de`elif`cláusulas y, finalmente, una opcional`else`cláusula:

```
if <expression>:
    <suite>
elif <expression>:
    <suite>
else:
    <suite>
```

Al ejecutar una declaración condicional, cada cláusula se considera en orden.

1. Evalúa la expresión del encabezado.
2. Si es un valor verdadero, ejecute la suite. Luego, omita todas las cláusulas subsiguientes en la declaración condicional.

Si el`else`Se alcanza la cláusula (lo que sólo ocurre si todos`if`y`elif`Las expresiones evalúan valores falsos), se ejecuta su suite.

Contextos booleanos. Arriba, los procedimientos de ejecución mencionan "un valor falso" y "un valor verdadero". Se dice que las expresiones dentro de las declaraciones de encabezado de los bloques condicionales están en *contextos booleanos* : sus valores de verdad son importantes para controlar el flujo, pero de lo contrario sus valores nunca se pueden asignar ni devolver. Python incluye varios valores falsos, incluidos 0,`None`, y el *booleano* valor`False`Todos los demás números son valores verdaderos. En el Capítulo 2, veremos que cada tipo de datos nativo en Python tiene valores verdaderos y falsos.

Valores booleanos Python tiene dos valores booleanos, llamados`True`y`False`Los valores booleanos representan valores de verdad en expresiones lógicas. Las operaciones de comparación integradas,`>, <, >=, <=, ==, !=`, devuelve estos valores.

```
def square(x):
    return mul(x, x)
```

```
False
```

```
def square(x):
    return mul(x, x)
```

```
True
```

Este segundo ejemplo dice "5 es mayor o igual a 5", y corresponde a la función`ge`en el`operator`módulo.

```
def square(x):
    return mul(x, x)
```

```
True
```

Este último ejemplo dice "0 es igual a -0" y corresponde a`eq`en el`operator`módulo. Tenga en cuenta que Python distingue entre asignación (`=`) de la prueba de igualdad (`==`), una convención compartida entre muchos lenguajes de programación.

Operadores booleanos Python también incorpora tres operadores lógicos básicos:

```
def square(x):
    return mul(x, x)
```

```
False
```

```
def square(x):
    return mul(x, x)
```

```
True
```

```
def square(x):
    return mul(x, x)
```

```
True
```

Las expresiones lógicas tienen procedimientos de evaluación correspondientes. Estos procedimientos aprovechan el hecho de que el valor de verdad de una expresión lógica a veces se puede determinar sin evaluar todas sus subexpresiones, una característica llamada *cortocircuito*.

Para evaluar la expresión`<left> and <right>`:

1. Evaluar la subexpresión`<left>`.
2. Si el resultado es un valor falso`v`, entonces la expresión se evalúa como`v`.
3. De lo contrario, la expresión se evalúa como el valor de la subexpresión.`<right>`.

Para evaluar la expresión`<left> or <right>`:

1. Evaluar la subexpresión`<left>`.
2. Si el resultado es un valor verdadero`v`, entonces la expresión se evalúa como`v`.
3. De lo contrario, la expresión se evalúa como el valor de la subexpresión.`<right>`.

Para evaluar la expresión`not <exp>`:

1. Evaluar`<exp>`; El valor es`True`Si el resultado es un valor falso, y`False`de lo contrario.

Estos valores, reglas y operadores nos proporcionan una forma de combinar los resultados de las pruebas. Las funciones que realizan pruebas y devuelven valores booleanos suelen comenzar con`is`, no seguido de un guión bajo (p. ej.,`isfinite`, `isdigit`, `isinstance`, etc.).

1.5.5 Iteración

Además de seleccionar qué sentencias ejecutar, las sentencias de control se utilizan para expresar la repetición. Si cada línea de código que escribimos se ejecutara sólo una vez, la programación sería un ejercicio muy improductivo. Sólo mediante la ejecución repetida de sentencias liberamos el potencial de las computadoras para hacernos poderosos. Ya hemos visto una forma de repetición: una función puede aplicarse muchas veces, aunque sólo se defina una vez. Las estructuras de control iterativas son otro mecanismo para ejecutar las mismas sentencias muchas veces.

Consideremos la secuencia de números de Fibonacci, en la que cada número es la suma de los dos anteriores:

Número de serie 16

Cada valor se construye aplicando repetidamente la regla de la suma de los dos valores anteriores. Para construir el valor n, necesitamos hacer un seguimiento de cuántos valores hemos creado (`k`), junto con el valor k (`curr`) y su predecesor (`pred`), así:

```
def square(x):
    return mul(x, x)
```

Número de serie 18

```
13
```

Recuerde que las comas separan varios nombres y valores en una declaración de asignación. La línea:

```
def square(x):
    return mul(x, x)
```

tiene el efecto de volver a vincular el nombre`pred`al valor de`curr`, y al mismo tiempo volver a enlazar`curr`al valor de`pred + curr`. Todas las expresiones a la derecha de=se evalúan antes de que se realice cualquier revinculación.

A`while`La cláusula contiene una expresión de encabezado seguida de un conjunto:

```
def print_square(x):
    print(square(x))
```

Para ejecutar una`while`cláusula:

1. Evalúa la expresión del encabezado.
2. Si es un valor verdadero, ejecute la suite y luego regrese al paso 1.

En el paso 2, todo el conjunto de`while`La cláusula se ejecuta antes de que se evalúe nuevamente la expresión del encabezado.

Para evitar la suite de un`while`Para evitar que la cláusula se ejecute indefinidamente, la suite siempre debe cambiar el estado del entorno en cada pasada.

A`while`La declaración que no termina se llama bucle infinito. Presione`<Control>-C`para forzar a Python a detener el bucle.

1.5.6 Orientación práctica: Pruebas

Probar una función es el acto de verificar que el comportamiento de la función coincide con las expectativas. Nuestro lenguaje de funciones es ahora lo suficientemente complejo como para que necesitemos comenzar a probar nuestras implementaciones.

A *prueba* es un mecanismo para realizar esta verificación sistemáticamente. Las pruebas suelen adoptar la forma de otra función que contiene una o más llamadas de muestra a la función que se está probando. El valor devuelto se verifica luego con un resultado esperado. A diferencia de la mayoría de las funciones, que están pensadas para ser generales, las pruebas implican la selección y validación de llamadas con valores de argumentos específicos. Las pruebas también sirven como documentación: demuestran cómo llamar a una función y qué valores de argumentos son apropiados.

Tenga en cuenta que también hemos utilizado la palabra "prueba" como término técnico para la expresión en el encabezado de un`if`o`while`Declaración. El contexto debería dejar claro cuándo usamos la palabra "prueba" para denotar una expresión y cuándo la usamos para denotar un mecanismo de verificación.

Afirmaciones. Los programadores utilizan`assert`declaraciones para verificar expectativas, como la salida de una función que se está probando.`assert`La declaración tiene una expresión en un contexto booleano, seguida de una línea de texto entre comillas (tanto las comillas simples como las dobles están bien, pero deben ser consistentes) que se mostrará si la expresión se evalúa como un valor falso.

```
def print_square(x):
    print(square(x))
```

Cuando la expresión que se está afirmando se evalúa como un valor verdadero, la ejecución de una declaración de afirmación no tiene ningún efecto. Cuando es un valor falso,`assert`Provoca un error que detiene la ejecución.

Una función de prueba para`fib`Debería probar varios argumentos, incluidos valores extremos de`n`.

```
def print_square(x):
    print(square(x))
```

Al escribir Python en archivos, en lugar de hacerlo directamente en el intérprete, las pruebas deben escribirse en el mismo archivo o en un archivo vecino con el sufijo`_test.py`.

Pruebas de doctorado. Python proporciona un método conveniente para colocar pruebas simples directamente en la cadena de documentación de una función. La primera línea de una cadena de documentación debe contener una descripción de una sola línea de la función, seguida de una línea en blanco. A continuación puede incluirse una descripción detallada de los argumentos y el comportamiento. Además, la cadena de documentación puede incluir una sesión interactiva de muestra que llama a la función:

```
def print_square(x):
    print(square(x))
```

Luego, la interacción se puede verificar a través de lamódulo doctestAbajo, el`globals`La función devuelve una representación del entorno global, que el intérprete necesita para evaluar expresiones.

```
def print_square(x):
    print(square(x))
```

Al escribir Python en archivos, todas las pruebas de documentación en un archivo se pueden ejecutar iniciando Python con la opción de línea de comando doctest:

```
def print_square(x):
    print(square(x))
```

La clave para realizar pruebas efectivas es escribir (y ejecutar) pruebas inmediatamente después (o incluso antes) de implementar nuevas funciones. Una prueba que aplica una sola función se denomina *prueba unitaria* Las pruebas unitarias exhaustivas son un sello distintivo de un buen diseño de programa.

Contents

2.1 Introducción **1**
2.1.1 La metáfora del objeto 1
2.1.2 Tipos de datos nativos 3

\tableofcontents

2.1 Introducción

En el capítulo 1 nos concentramos en los procesos computacionales y en el papel de las funciones en el diseño de programas. Vimos cómo utilizar datos primitivos (números) y operaciones primitivas (operaciones aritméticas), cómo formar funciones compuestas mediante composición y control, y cómo crear abstracciones funcionales dando nombres a los procesos. También vimos que las funciones de orden superior mejoran el poder de nuestro lenguaje al permitirnos manipular, y por lo tanto razonar, en términos de métodos generales de computación. Esto es gran parte de la esencia de la programación.

Este capítulo se centra en los datos. Los datos nos permiten representar y manipular información sobre el mundo utilizando las herramientas computacionales que hemos adquirido hasta ahora. Los programas sin estructuras de datos pueden ser suficientes para explorar las propiedades matemáticas. Pero los fenómenos del mundo real, como los documentos, las relaciones, las ciudades y los patrones climáticos, tienen una estructura compleja que se representa mejor utilizando *tipos de datos compuestos* Con datos estructurados, los programas pueden simular y razonar sobre prácticamente cualquier ámbito del conocimiento y la experiencia humanos. Gracias al crecimiento explosivo de Internet, una enorme cantidad de información estructurada sobre el mundo está disponible para todos nosotros en línea.

2.1.1 La metáfora del objeto

Al principio de este curso, hicimos una distinción entre funciones y datos: las funciones realizaban operaciones y se operaba sobre los datos. Cuando incluimos valores de funciones entre nuestros datos, reconocimos que los datos también pueden tener comportamiento. Se podía operar sobre las funciones como si fueran datos, pero también se las podía llamar para realizar cálculos.

En este curso,*objetos* servirá como nuestra metáfora central de programación para valores de datos que también tienen comportamiento. Los objetos representan información, pero también *comportarse* Al igual que los conceptos abstractos que representan, la lógica de cómo un objeto interactúa con otros objetos se combina con la información que codifica el valor del objeto. Cuando se imprime un objeto, sabe cómo deletrearse con letras y números. Si un objeto está compuesto de partes, sabe cómo revelar esas partes cuando se lo solicitan.

Los objetos son a la vez información y procesos, agrupados para representar las propiedades, interacciones y comportamientos de cosas complejas.

La metáfora de objeto se implementa en Python mediante una sintaxis de objeto especializada y una terminología asociada, que podemos presentar con un ejemplo. Una fecha es un tipo de objeto simple.

```
from datetime import date
```

El nombre`date`Está ligado a un *clase* Una clase representa un tipo de objeto. Las fechas individuales se denominan *instancias* de esa clase, y pueden ser *construido* llamando a la clase como una función sobre argumentos que caracterizan la instancia.

```
today = date(2011, 9, 12)
```

Mientras`today`se construyó a partir de números primitivos, se comporta como una fecha. Por ejemplo, al restarlo de otra fecha, se obtendrá una diferencia horaria, que podemos mostrar como una línea de texto llamando`str`.

```
str(date(2011, 12, 2) - today)
```

```
'81 days, 0:00:00'
```

Los objetos tienen *atributos* , que son valores nombrados que forman parte del objeto. En Python, utilizamos la notación de puntos para designar un atributo de un objeto.

.

Arriba, el`<expression>`se evalúa como un objeto, y`<name>`es el nombre de un atributo para ese objeto.

A diferencia de los nombres que hemos considerado hasta ahora, estos nombres de atributos no están disponibles en el entorno general. En cambio, los nombres de atributos son específicos de la instancia del objeto que precede al punto.

```
today.year
```

Los objetos también tienen *métodos* , que son atributos con valores de función. Metafóricamente, el objeto "sabe" cómo llevar a cabo esos métodos. Los métodos calculan sus resultados a partir de sus argumentos y de su objeto. Por ejemplo, The`strftime`método de`today`toma un único argumento que especifica cómo mostrar una fecha (por ejemplo,`%A`significa que el día de la semana debe escribirse con todas las letras).

```
today.strftime('%A, %B %d')
```

```
'Monday, September 12'
```

Calcular el valor de retorno de`strftime`requiere dos entradas: la cadena que describe el formato de la salida y la información de fecha incluida en`today`En este método se aplica una lógica específica de fecha para obtener este resultado. Nunca dijimos que el 12 de septiembre de 2011 fuera lunes, pero conocer el día

de la semana en el que uno vive forma parte de lo que significa ser una fecha. Al agrupar el comportamiento y la información, este objeto de Python nos ofrece una abstracción convincente y autónoma de una fecha.

La notación de puntos proporciona otra forma de expresión combinada en Python. La notación de puntos también tiene un procedimiento de evaluación bien definido. Sin embargo, el desarrollo de una explicación precisa de cómo se evalúa la notación de puntos tendrá que esperar hasta que presentemos el paradigma completo de la programación orientada a objetos en las próximas secciones.

Aunque todavía no hemos descrito con precisión cómo funcionan los objetos, es hora de comenzar a pensar en los datos como objetos ahora, porque en Python cada valor es un objeto.

2.1.2 Tipos de datos nativos

Cada objeto en Python tiene un *tipo*. El`type`La función nos permite inspeccionar el tipo de un objeto.

```
type(today)
```

Hasta ahora, los únicos tipos de objetos que hemos estudiado son números, funciones, booleanos y ahora fechas. También nos encontramos brevemente con conjuntos y cadenas, pero necesitaremos estudiarlos con más profundidad. Hay muchos otros tipos de objetos (sonidos, imágenes, ubicaciones, conexiones de datos, etc.), la mayoría de los cuales se pueden definir mediante los medios de combinación y abstracción que desarrollamos en este capítulo. Python tiene solo un puñado de objetos primitivos o *nativo* tipos de datos integrados en el lenguaje.

Los tipos de datos nativos tienen las siguientes propiedades:

1. Hay expresiones primitivas que evalúan objetos de estos tipos, llamados *literales*.
2. Hay funciones, operadores y métodos integrados para manipular estos objetos.

Como hemos visto, los números son nativos; los literales numéricos evalúan números y los operadores matemáticos manipulan objetos numéricos.

```
12 + 3000000000000000000000000
```

De hecho, Python incluye tres tipos numéricos nativos: enteros (`int`), números reales (`float`) y números complejos (`complex`).

```
type(2)
```

```
type(1.5)
```

```
type(1+1j)
```

El nombre`float`proviene de la forma en que se representan los números reales en Python: una representación de "punto flotante". Si bien los detalles de cómo se representan los números no son un tema de este curso, algunas diferencias de alto nivel entre`int`y`float`Es importante conocer los objetos. En particular,`int`Los objetos sólo pueden representar números enteros, pero los representan exactamente, sin ninguna aproximación. Por otra parte,`float`Los objetos pueden representar una amplia gama de números fraccionarios, pero no todos los números racionales son representables. No obstante, los objetos float se utilizan a menudo para representar números reales y racionales de forma aproximada, hasta un número determinado de cifras significativas.

Lectura adicional. Las siguientes secciones presentan más tipos de datos nativos de Python, centrándose en el papel que desempeñan en la creación de abstracciones de datos útiles. Un capítulo sobretipos de datos nativosEn Dive Into Python 3 se ofrece una descripción general pragmática de todos los tipos de datos nativos de Python y cómo utilizarlos de manera eficaz, incluidos numerosos ejemplos de uso y consejos prácticos. No es necesario que lea ese capítulo ahora, pero considérelo una referencia valiosa.

Contents

2.2 Abstracción de datos **1**
2.2.1 Ejemplo: Aritmética sobre números racionales 2
2.2.2 Tuplas 3
2.2.3 Barreras de abstracción 5
2.2.4 Las propiedades de los datos 7

\tableofcontents

2.2 Abstracción de datos

Si consideramos el amplio conjunto de cosas del mundo que nos gustaría representar en nuestros programas, descubrimos que la mayoría de ellas tienen una estructura compuesta. Una fecha tiene un año, un mes y un día; una posición geográfica tiene una latitud y una longitud. Para representar posiciones, nos gustaría que nuestro lenguaje de programación tuviera la capacidad de "pegar" una latitud y una longitud para formar un par: un *datos compuestos* valor — que nuestros programas podrían manipular de una manera que sería consistente con el hecho de que consideramos una posición como una unidad conceptual única, que tiene dos partes.

El uso de datos compuestos también nos permite aumentar la modularidad de nuestros programas. Si podemos manipular posiciones geográficas directamente como objetos por derecho propio, entonces podemos separar la parte de nuestro programa que trata con valores per se de los detalles de cómo se pueden representar esos valores. La técnica general de aislar las partes de un programa que tratan de cómo se representan los datos de las partes de un programa que tratan de cómo se manipulan esos datos es una poderosa metodología de diseño llamada *abstracción de datos* La abstracción de datos hace que los programas sean mucho más fáciles de diseñar, mantener y modificar.

La abstracción de datos es similar en carácter a la abstracción funcional. Cuando creamos una abstracción funcional, se pueden suprimir los detalles de cómo se implementa una función, y la función en sí puede reemplazarse por cualquier otra función con el mismo comportamiento general. En otras palabras, podemos hacer una abstracción que separe la forma en que se utiliza la función de los detalles de cómo se implementa. Análogamente, la abstracción de datos es una metodología que nos permite aislar cómo se utiliza un objeto de datos compuesto de los detalles de cómo se construye.

La idea básica de la abstracción de datos es estructurar los programas de forma que operen sobre datos abstractos. Es decir, nuestros programas deben utilizar los datos de forma que se hagan la menor cantidad posible de suposiciones sobre ellos. Al mismo tiempo, se define una representación concreta de los datos, independientemente de los programas que utilicen los datos. La interfaz entre estas dos partes de nuestro sistema será un conjunto de funciones, llamadas

selectores y constructores, que implementan los datos abstractos en términos de la representación concreta. Para ilustrar esta técnica, consideraremos cómo diseñar un conjunto de funciones para manipular números racionales.

A medida que lea las siguientes secciones, tenga en cuenta que la mayoría del código Python escrito hoy en día utiliza tipos de datos abstractos de muy alto nivel que están integrados en el lenguaje, como clases, diccionarios y listas. Dado que estamos desarrollando una comprensión de cómo funcionan estas abstracciones, aún no podemos usarlas nosotros mismos. En consecuencia, escribiremos un código que no es Pythonic; no es necesariamente la forma típica de implementar nuestras ideas en el lenguaje. Sin embargo, lo que escribimos es instructivo, porque demuestra cómo se pueden construir estas abstracciones. Recuerde que la informática no se trata solo de aprender a usar lenguajes de programación, sino también de aprender cómo funcionan.

2.2.1 Ejemplo: Aritmética sobre números racionales

Recordemos que un número racional es una proporción de números enteros y que los números racionales constituyen una subclase importante de los números reales. Un número racional como`1/3`o`17/29`Normalmente se escribe así:

```
<numerator>/<denominator>
```

donde ambos`<numerator>`y`<denominator>`son marcadores de posición para valores enteros. Ambas partes son necesarias para caracterizar exactamente el valor del número racional.

Los números racionales son importantes en informática porque, al igual que los números enteros, se pueden representar con exactitud. Los números irracionales (como`pi`o`e`o`sqrt(2)`) se aproximan, en cambio, utilizando una expansión binaria finita. Por lo tanto, trabajar con números racionales debería, en principio, permitirnos evitar errores de aproximación en nuestra aritmética.

Sin embargo, tan pronto como dividimos el numerador por el denominador, podemos quedarnos con una aproximación decimal truncada (una`float`).

```
1/3
```

Y los problemas con esta aproximación aparecen cuando empezamos a realizar pruebas:

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

```
True
```

La forma en que las computadoras aproximan números reales con expansiones decimales de longitud finita es un tema para otra clase. La idea importante aquí es que al representar números racionales como cocientes de números enteros, evitamos por completo el problema de aproximación. Por lo tanto, nos gustaría mantener el numerador y el denominador separados por razones de precisión, pero tratarlos como una sola unidad.

Sabemos, gracias al uso de abstracciones funcionales, que podemos empezar a programar de forma productiva antes de tener una implementación de algunas partes de nuestro programa. Comencemos suponiendo que ya tenemos una forma de construir un número racional a partir de un numerador y un denominador. Supongamos también que, dado un número racional, tenemos una forma de extraer (o seleccionar) su numerador y su denominador. Supongamos además que el constructor y los selectores están disponibles como las tres funciones siguientes:

- `make_rat(n, d)`devuelve el número racional con numerador`n`y denominador`d`.
- `numer(x)`devuelve el numerador del número racional`x`.
- `denom(x)`devuelve el denominador del número racional`x`.

Utilizamos aquí una potente estrategia de síntesis:*Pensamiento ilusorio* Aún no hemos dicho cómo se representa un número racional, ni cómo se representan las funciones`numer`, `denom`, y`make_rat`Debería implementarse. Aun así, si tuviéramos estas tres funciones, podríamos sumar, multiplicar y probar la igualdad de números racionales llamándolas:

```
def add_rat(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return make_rat(nx * dy + ny * dx, dx * dy)

def mul_rat(x, y):
    return make_rat(numer(x) * numer(y), denom(x) * denom(y))

def eq_rat(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Ahora tenemos las operaciones sobre números racionales definidas en términos de las funciones selectoras.`numer`y`denom`, y la función constructora`make_rat`, pero aún no hemos definido estas funciones. Lo que necesitamos es alguna forma de unir un numerador y un denominador para formar una unidad.

2.2.2 Tuplas

Para permitirnos implementar el nivel concreto de nuestra abstracción de datos, Python proporciona una estructura compuesta llamada`tuple`, que se puede construir separando los valores con comas. Aunque no es estrictamente obligatorio, los paréntesis casi siempre rodean las tuplas.

```
(1, 2)
```

```
(1, 2)
```

Los elementos de una tupla se pueden descomprimir de dos maneras. La primera es mediante nuestro conocido método de asignación múltiple.

```
pair = (1, 2)
pair
```

```
(1, 2)
```

```
x, y = pair
x
```

```
y
```

De hecho, la asignación múltiple ha estado creando y descomprimiendo tuplas todo el tiempo.

Un segundo método para acceder a los elementos de una tupla es mediante el operador de indexación, escrito como corchetes.

```
1/3
```

```
1
```

```
1/3
```

```
2
```

Las tuplas en Python (y las secuencias en la mayoría de los otros lenguajes de programación) tienen un índice 0, lo que significa que el índice`0`Selecciona el primer elemento, índice`1`selecciona el segundo, y así sucesivamente. Una intuición que subyace a esta convención de indexación es que el índice representa la distancia a la que se encuentra un elemento desde el comienzo de la tupla.

La función equivalente para el operador de selección de elementos se llama`getitem`, y también utiliza posiciones indexadas en 0 para seleccionar elementos de una tupla.

```
1/3
```

```
1
```

Las tuplas son tipos nativos, lo que significa que existen operadores integrados de Python para manipularlas. Volveremos a las propiedades completas de las tuplas en breve. Por el momento, solo nos interesa cómo las tuplas pueden servir como el pegamento que implementa los tipos de datos abstractos.

Representando números racionales. Las tuplas ofrecen una forma natural de implementar números racionales como un par de dos enteros: un numerador y un denominador. Podemos implementar nuestras funciones de constructor y selector para números racionales manipulando tuplas de 2 elementos.

```
1/3
```

```
1/3
```

```
1/3
```

Una función para imprimir números racionales completa nuestra implementación de este tipo de datos abstracto.

Número de serie 16

Junto con las operaciones aritméticas que definimos anteriormente, podemos manipular números racionales con las funciones que hemos definido.

```
1/3
```

```
'1/2'
```

Número de serie 18

```
'1/6'
```

```
1/3
```

```
'6/9'
```

Como muestra el ejemplo final, nuestra implementación de números racionales no reduce los números racionales a su mínima expresión. Podemos solucionar esto cambiando`make_rat`Si tenemos una función para calcular el máximo común denominador de dos números enteros, podemos usarla para reducir el numerador y el denominador a sus términos más bajos antes de construir el par. Como sucede con muchas herramientas útiles, esta función ya existe en la biblioteca de Python.

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

El operador de doble barra,`//`, expresa la división de números enteros, que redondea hacia abajo la parte fraccionaria del resultado de la división. Como sabemos que`g`divide a ambos`n`y`d`De manera uniforme, la división entera es exacta en este caso. Ahora tenemos

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

```
'2/3'
```

como se desea. Esta modificación se logró cambiando el constructor sin cambiar ninguna de las funciones que implementan las operaciones aritméticas reales.

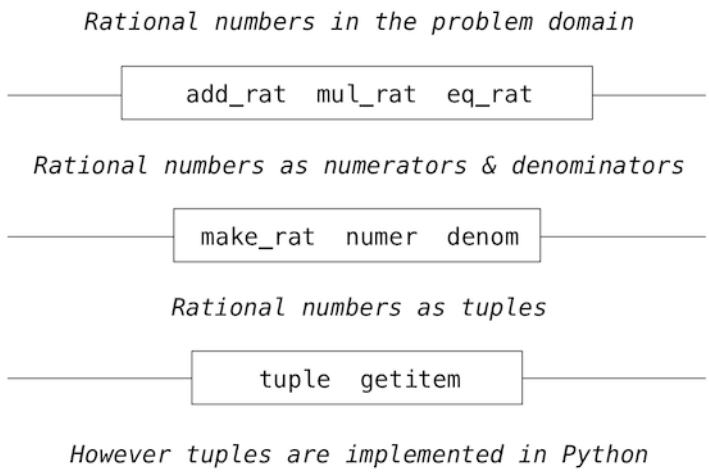
Lectura adicional. El`str_rat`La implementación anterior utiliza *cadenas de formato* , que contienen marcadores de posición para valores. Los detalles sobre cómo utilizar cadenas de formato y`format`El método aparece en elcadenas de formatoSección de Dive Into Python 3.

2.2.3 Barreras de abstracción

Antes de continuar con más ejemplos de datos compuestos y abstracción de datos, consideremos algunas de las cuestiones planteadas por el ejemplo del número racional. Definimos las operaciones en términos de un constructor.`make_rat`y selectores`numer`y`denom`En general, la idea subyacente de la

abstracción de datos es identificar para cada tipo de valor un conjunto básico de operaciones en términos de las cuales se expresarán todas las manipulaciones de valores de ese tipo y luego utilizar solo esas operaciones para manipular los datos.

Podemos imaginar la estructura del sistema de números racionales como una serie de capas.



Las líneas horizontales representan barreras de abstracción que aíslan diferentes niveles del sistema. En cada nivel, la barrera separa las funciones (arriba) que utilizan la abstracción de datos de las funciones (abajo) que implementan la abstracción de datos. Los programas que utilizan números racionales los manipulan únicamente en términos de sus funciones aritméticas:`add_rat`, `mul_rat`, y`eq_rat`Estos, a su vez, se implementan únicamente en términos del constructor y los selectores.`make_rat`, `numer`, y`denom`, que a su vez se implementan en términos de tuplas. Los detalles de cómo se implementan las tuplas son irrelevantes para el resto de las capas, siempre que las tuplas permitan la implementación de los selectores y el constructor.

En cada capa, las funciones dentro de la caja imponen el límite de abstracción porque son las únicas funciones que dependen tanto de la representación que se encuentra por encima de ellas (por su uso) como de la implementación que se encuentra por debajo de ellas (por sus definiciones). De esta manera, las barreras de abstracción se expresan como conjuntos de funciones.

Las barreras de abstracción ofrecen muchas ventajas. Una de ellas es que hacen que los programas sean mucho más fáciles de mantener y modificar. Cuantas menos funciones dependan de una representación concreta, menos cambios se necesitarán cuando se desee cambiar esa representación.

2.2.4 Las propiedades de los datos

Comenzamos la implementación de números racionales implementando operaciones aritméticas en términos de tres funciones no especificadas:`make_rat`, `numer`, y`denom`En ese punto, podríamos pensar en las operaciones como definidas en términos de objetos de datos — numeradores, denominadores y números racionales — cuyo comportamiento estaba especificado por las últimas tres funciones.

Pero, ¿qué se entiende exactamente por datos? No basta con decir "lo que sea implementado por los selectores y constructores dados". Necesitamos garantizar que estas funciones juntas especifiquen el comportamiento correcto. Es decir, si construimos un número racional`x`de números enteros`n`y`d`, entonces debería ser el caso que`numer(x)/denom(x)`es igual a`n/d`.

En general, podemos pensar en un tipo de datos abstracto como definido por una colección de selectores y constructores, junto con algunas condiciones de comportamiento. Mientras se cumplan las condiciones de comportamiento (como la propiedad de división anterior), estas funciones constituyen una representación válida del tipo de datos.

Este punto de vista se puede aplicar también a otros tipos de datos, como la tupla de dos elementos que utilizamos para implementar números racionales. En realidad, nunca dijimos mucho sobre qué era una tupla, solo que el lenguaje proporcionaba operadores para crear y manipular tuplas. Ahora podemos describir las condiciones de comportamiento de las tuplas de dos elementos, también llamadas pares, que son relevantes para el problema de la representación de números racionales.

Para implementar números racionales, necesitábamos una forma de pegamento para dos números enteros, que tuviera el siguiente comportamiento:

- Si un par`p`se construyó a partir de valores`x`y`y`, entonces`getitem_pair(p, 0)`devuelve`x`, y`getitem_pair(p, 1)`devuelve`y`.

Podemos implementar funciones`make_pair`y`getitem_pair`que cumplen esta descripción tan bien como una tupla.

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

Con esta implementación, podemos crear y manipular pares.

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

```
1
```

```
1/3 == 0.33333333333333300000  # Beware of approximations
```

```
2
```

Este uso de funciones no se corresponde en nada con nuestra noción intuitiva de lo que deberían ser los datos. Sin embargo, estas funciones son suficientes para representar datos compuestos en nuestros programas.

El punto sutil a tener en cuenta es que el valor devuelto por`make_pair`es una función llamada`dispatch`, que toma un argumento`m`y devuelve cualquiera`x`o`y`. Entonces,`getitem_pair`llama a esta función para recuperar el valor apropiado. Volveremos al tema de las funciones de envío varias veces a lo largo de este capítulo.

El objetivo de mostrar la representación funcional de un par no es que Python realmente funcione de esta manera (las tuplas se implementan de manera más directa, por razones de eficiencia), sino que podría funcionar de esta manera. La representación funcional, aunque oscura, es una forma perfectamente adecuada de representar pares, ya que cumple las únicas condiciones que los pares deben cumplir. Este ejemplo también demuestra que la capacidad de manipular funciones como valores nos proporciona automáticamente la capacidad de representar datos compuestos.

Contents

2.3 Secuencias **1**
2.3.1 Pares anidados 2
2.3.2 Listas recursivas 3
2.3.3 Tuplas II 6
2.3.4 Iteración de secuencia 7
2.3.5 Abstracción de secuencias 10
2.3.6 Cadenas 12
2.3.7 Interfaces convencionales 15

\tableofcontents

2.3 Secuencias

Una secuencia es una colección ordenada de valores de datos. A diferencia de un par, que tiene exactamente dos elementos, una secuencia puede tener una cantidad arbitraria (pero finita) de elementos ordenados.

La secuencia es una abstracción fundamental y poderosa en la informática. Por ejemplo, si tenemos secuencias, podemos enumerar a todos los estudiantes de Berkeley, o de todas las universidades del mundo, o a todos los estudiantes de todas las universidades. Podemos enumerar todas las clases que hemos tomado, todas las tareas que hemos completado, todas las calificaciones que hemos recibido. La abstracción de secuencias permite la ejecución de miles de programas basados en datos que afectan a nuestras vidas todos los días.

Una secuencia no es un tipo de datos abstracto en particular, sino una colección de comportamientos que comparten distintos tipos. Es decir, hay muchos tipos de secuencias, pero todas comparten ciertas propiedades. En particular,

Longitud. Una secuencia tiene una longitud finita.

Selección de elementos. Una secuencia tiene un elemento correspondiente a cualquier índice entero no negativo menor que su longitud, comenzando en 0 para el primer elemento.

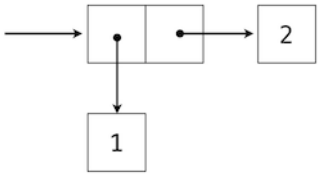
A diferencia de un tipo de datos abstracto, no hemos explicado cómo construir una secuencia. La abstracción de secuencia es una colección de comportamientos que no especifica completamente un tipo (es decir, con constructores y selectores), pero que puede ser compartida entre varios tipos. Las secuencias proporcionan una capa de abstracción que puede ocultar los detalles de exactamente qué tipo de secuencia está siendo manipulado por un programa en particular.

En esta sección, desarrollamos un tipo de datos abstracto particular que puede implementar la abstracción de secuencia. Luego, presentamos tipos integrados de Python que también implementan la misma abstracción.

2.3.1 Pares anidados

Para los números racionales, emparejamos dos objetos enteros utilizando una tupla de dos elementos y luego demostramos que también podíamos implementar pares utilizando funciones. En ese caso, los elementos de cada par que construimos eran números enteros. Sin embargo, al igual que las expresiones, las tuplas pueden anidarse. Cualquier elemento de un par puede ser en sí mismo un par, una propiedad que es válida para cualquiera de los métodos de implementación de un par que hemos visto: como una tupla o como una función de envío.

Una forma estándar de visualizar un par: en este caso, el par`(1,2)`-– se llama *caja y puntero* Notación. Cada valor, compuesto o primitivo, se representa como un puntero a un cuadro. El cuadro de un valor primitivo contiene una representación de ese valor. Por ejemplo, el cuadro de un número contiene un numeral. El cuadro de un par es en realidad un cuadro doble: la parte izquierda contiene (una flecha hacia) el primer elemento del par y la parte derecha contiene el segundo.

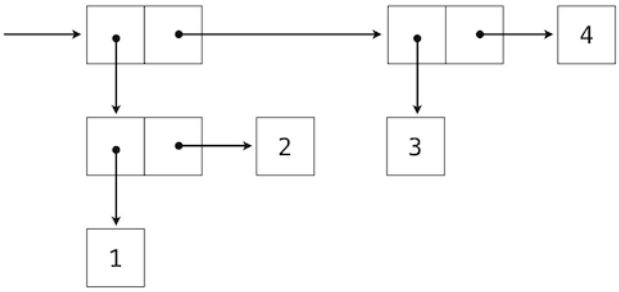


Esta expresión de Python para una tupla anidada,

```
((1, 2), (3, 4))
```

```
((1, 2), (3, 4))
```

Tendría la siguiente estructura.

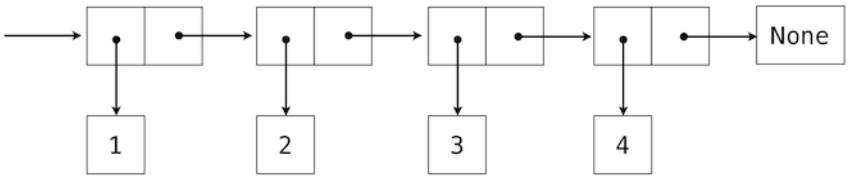


Nuestra capacidad de utilizar tuplas como elementos de otras tuplas proporciona un nuevo medio de combinación en nuestro lenguaje de programación. Llamamos a la capacidad de las tuplas de anidarse de esta manera una *propiedad de cierre* del tipo de datos tupla. En general, un método para combinar valores de datos satisface la propiedad de clausura si el resultado de la combinación puede combinarse mediante el mismo método. La clausura es la clave

del poder en cualquier método de combinación porque nos permite crear estructuras jerárquicas: estructuras formadas por partes, que a su vez están formadas por partes, y así sucesivamente. Exploraremos una variedad de estructuras jerárquicas en el Capítulo 3. Por ahora, consideraremos una estructura particularmente importante.

2.3.2 Listas recursivas

Podemos utilizar pares anidados para formar listas de elementos de longitud arbitraria, lo que nos permitirá implementar la abstracción de secuencia. La figura siguiente ilustra la estructura de la representación recursiva de una lista de cuatro elementos:`1`, `2`, `3`, `4`.



La lista se representa mediante una cadena de pares. El primer elemento de cada par es un elemento de la lista, mientras que el segundo es un par que representa el resto de la lista. El segundo elemento del par final es`None`, lo que indica que la lista ha terminado. Podemos construir esta estructura utilizando un literal de tupla anidada:

```
(1, (2, (3, (4, None))))
```

```
(1, (2, (3, (4, None))))
```

Esta estructura anidada corresponde a una forma muy útil de pensar en las secuencias en general, que hemos visto antes en las reglas de ejecución del intérprete de Python. Una secuencia no vacía se puede descomponer en:

- su primer elemento, y
- El resto de la secuencia.

El resto de una secuencia es en sí mismo una secuencia (posiblemente vacía). Llamamos recursiva a esta visión de las secuencias, porque las secuencias contienen otras secuencias como segundo componente.

Dado que nuestra representación de lista es recursiva, la llamaremos`rlist`en nuestra implementación, para no confundirlo con el incorporado`list`tipo en Python que presentaremos más adelante en este capítulo. Una lista recursiva se puede construir a partir de un primer elemento y el resto de la lista. El valor`None`representa una lista recursiva vacía.

```
empty_rlist = None
def make_rlist(first, rest):
```

```
    """Make a recursive list from its first element and the rest."""
    return (first, rest)

def first(s):
    """Return the first element of a recursive list s."""
    return s[0]

def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

Estos dos selectores, un constructor y una constante, implementan juntos el tipo de datos abstracto de lista recursiva. La única condición de comportamiento para una lista recursiva es que, como un par, su constructor y sus selectores sean funciones inversas.

- Si una lista recursiva`s`Fue construido a partir de elementos`f`y lista`r`, entonces`first(s)`devoluciones`f`, y`rest(s)`devoluciones`r`.

Podemos utilizar el constructor y los selectores para manipular listas recursivas.

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
rest(counts)
```

```
(2, (3, (4, None)))
```

Recordemos que pudimos representar pares usando funciones y, por lo tanto, también podemos representar listas recursivas usando funciones.

La lista recursiva puede almacenar una secuencia de valores en orden, pero aún no implementa la abstracción de secuencia. Utilizando el tipo de datos abstracto que hemos definido, podemos implementar los dos comportamientos que caracterizan a una secuencia: longitud y selección de elementos.

```
def len_rlist(s):
    """Return the length of recursive list s."""
    length = 0
    while s != empty_rlist:
        s, length = rest(s), length + 1
    return length

def getitem_rlist(s, i):
    """Return the element at index i of recursive list s."""
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

Ahora, podemos manipular una lista recursiva como una secuencia:

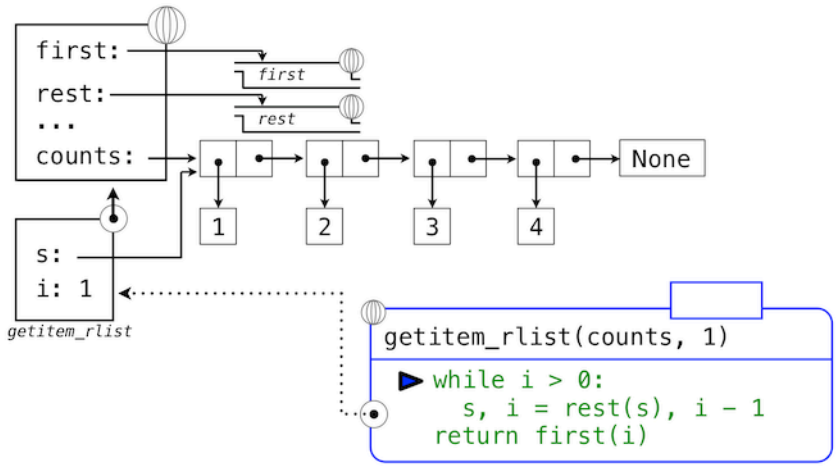
```
len_rlist(counts)
```

```
(1, (2, (3, (4, None))))
```

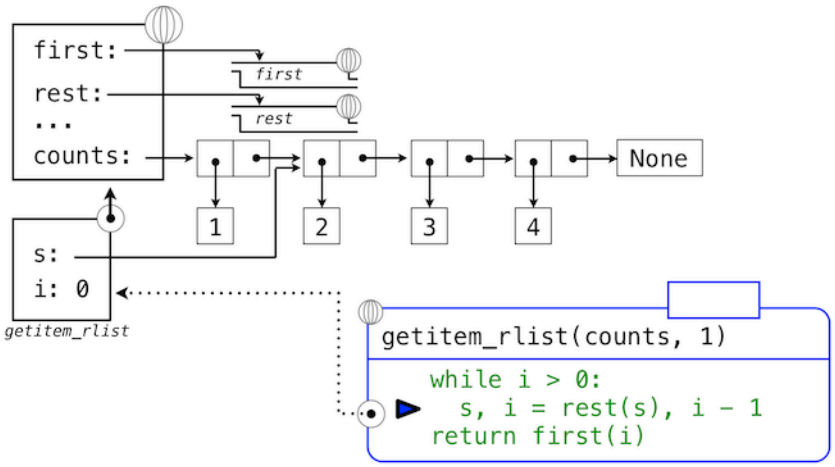
```
2
```

Ambas implementaciones son iterativas. Van quitando cada capa del par anidado hasta el final de la lista (en`len_rlist`) o el elemento deseado (en`getitem_rlist`) se alcanza.

La serie de diagramas de entorno a continuación ilustra el proceso iterativo mediante el cual`getitem_rlist`encuentra el elemento`2`en el índice`1`En la lista recursiva. Primero, la función`getitem_rlist`se llama, creando un marco local.

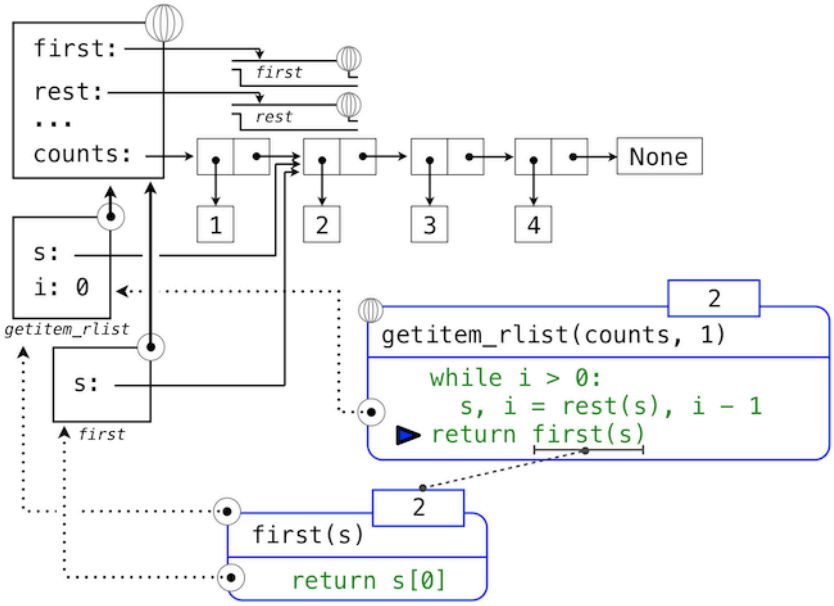


La expresión en el`while`el encabezado se evalúa como verdadero, lo que hace que la declaración de asignación en el`while`Suite a ejecutar.



En este caso, el nombre local`s`ahora se refiere a la sublista que comienza con el segundo elemento de la lista original. Evaluar la`while`La expresión de encabezado

ahora produce un valor falso y, por lo tanto, Python evalúa la expresión en la declaración de retorno en la línea final de`getitem_rlist`.



Este diagrama de entorno final muestra el marco local para la llamada a`first`, que contiene el nombres`s`vinculado a esa misma sublista.`first`La función selecciona el valor`2`y lo devuelve, completando la llamada a`getitem_rlist`.

Este ejemplo demuestra un patrón común de cálculo con listas recursivas, donde cada paso de una iteración opera sobre un sufijo cada vez más corto de la lista original. Este procesamiento incremental para encontrar la longitud y los elementos de una lista recursiva lleva algún tiempo de cálculo. (En el Capítulo 3, aprenderemos a caracterizar el tiempo de cálculo de funciones iterativas como estas). Los tipos de secuencia integrados de Python se implementan de una manera diferente que no tiene un gran costo computacional para calcular la longitud de una secuencia o recuperar sus elementos.

La forma en que construimos listas recursivas es bastante prolija. Afortunadamente, Python proporciona una variedad de tipos de secuencias integrados que brindan tanto la versatilidad de la abstracción de secuencias como una notación conveniente.

2.3.3 Tuplas II

De hecho, el`tuple`El tipo que introdujimos para formar pares primitivos es en sí mismo un tipo de secuencia completa. Las tuplas proporcionan una funcionalidad sustancialmente mayor que el tipo de datos abstractos de pares que implementamos funcionalmente.

Las tuplas pueden tener una longitud arbitraria y presentan los dos comportamientos principales de la abstracción de secuencia: longitud y selección de elementos. A continuación,`digits`es una tupla con cuatro elementos.

```
(1, (2, (3, (4, None))))
```

```
4
```

```
(1, (2, (3, (4, None))))
```

```
8
```

Además, las tuplas se pueden sumar y multiplicar por números enteros. En el caso de las tuplas, la suma y la multiplicación no suman ni multiplican elementos, sino que combinan y replican las tuplas en sí. Es decir, la`add`función en el`operator`módulo (y el`+`operador) devuelve una nueva tupla que es la conjunción de los argumentos añadidos.`mul`función en`operator`(y el`*`operador) puede tomar un número entero`k`y una tupla y devuelve una nueva tupla que consta de`k`copias del argumento de la tupla.

```
(1, (2, (3, (4, None))))
```

```
(2, 7, 1, 8, 2, 8, 1, 8, 2, 8)
```

Cartografía. Un método eficaz para transformar una tupla en otra es aplicar una función a cada elemento y recopilar los resultados. Esta forma general de cálculo se denomina *cartografía* una función sobre una secuencia, y corresponde a la función incorporada`map`. El resultado de`map`es un objeto que no es en sí mismo una secuencia, pero puede convertirse en una secuencia llamando`tuple`, la función constructora para tuplas.

```
(1, (2, (3, (4, None))))
```

```
(1, 2, 3, 4, 5)
```

El`map`La función es importante porque se basa en la abstracción de secuencia: no necesitamos preocuparnos por la estructura de la tupla subyacente; solo que podemos acceder a cada uno de sus elementos individualmente para pasarlo como argumento a la función mapeada (`abs`, en este caso).

2.3.4 Iteración de secuencia

El mapeo es en sí mismo una instancia de un patrón general de computación: iterar sobre todos los elementos de una secuencia. Para mapear una función sobre una secuencia, no solo seleccionamos un elemento en particular, sino cada elemento a su vez. Este patrón es tan común que Python tiene una declaración de control adicional para procesar datos secuenciales: la`for`declaración.

Considere el problema de contar cuántas veces aparece un valor en una secuencia. Podemos implementar una función para calcular este recuento utilizando una`while`bucle.

```
(1, (2, (3, (4, None))))
```

Número de serie 16

2

La pitónforLa declaración puede simplificar el cuerpo de esta función iterando sobre los valores de los elementos directamente, sin introducir el nombre.indexen absoluto.ForPor ejemplo (juego de palabras intencionado), podemos escribir:

```
(1, (2, (3, (4, None))))
```

Número de serie 18

2

AforLa declaración consta de una sola cláusula con la forma:

```
(1, (2, (3, (4, None))))
```

AforLa instrucción se ejecuta mediante el siguiente procedimiento:

1. Evaluar el encabezado<expression>, que debe producir un valor iterable.
2. Para cada valor de elemento en esa secuencia, en orden: 1. Unir<name>a ese valor en el entorno local. 2. Ejecutar el<suite>.

El paso 1 se refiere a un valor iterable. Las secuencias son iterables y sus elementos se consideran en su orden secuencial. Python incluye otros tipos iterables, pero por ahora nos centraremos en las secuencias; la definición general del término "iterable" aparece en la sección sobre iteradores del Capítulo 4.

Una consecuencia importante de este procedimiento de evaluación es que<name>se vinculará al último elemento de la secuencia después de laforSe ejecuta la sentencia.forloop introduce otra forma en la que el entorno local puede actualizarse mediante una declaración.

Desempaquetado de secuencia. Un patrón común en los programas es tener una secuencia de elementos que son en sí mismos secuencias, pero todos de una longitud fija.ForLas instrucciones pueden incluir varios nombres en su encabezado para "descomprimir" cada secuencia de elementos en sus respectivos elementos. Por ejemplo, podemos tener una secuencia de pares (es decir, tuplas de dos elementos),

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

y deseo encontrar el número de pares que tienen el mismo primer y segundo elemento.

```
empty_rlist = None
def make_rlist(first, rest):
```

```
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

La siguiente`for`Una declaración con dos nombres en su encabezado vinculará cada nombre`x`y`y`al primer y segundo elemento de cada par, respectivamente.

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
2
```

Este patrón de vincular múltiples nombres a múltiples valores en una secuencia de longitud fija se llama *desempaquetado de secuencia* ; Es el mismo patrón que vemos en las declaraciones de asignación que vinculan múltiples nombres a múltiples valores.

Rangos. A`range`es otro tipo de secuencia incorporado en Python, que representa un rango de números enteros. Los rangos se crean con el`range`función, que toma dos argumentos enteros: el primer número y uno más allá del último número en el rango deseado.

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
range(1, 10)
```

Llamando al`tuple`El constructor de un rango creará una tupla con los mismos elementos que el rango, de modo que los elementos se puedan inspeccionar fácilmente.

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
(5, 6, 7)
```

Si solo se da un argumento, se interpreta como uno más allá del último valor para un rango que comienza en 0.

```
empty_rlist = None
def make_rlist(first, rest):
```

```
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
(0, 1, 2, 3)
```

Los rangos suelen aparecer como expresión en una`for`encabezado para especificar el número de veces que se debe ejecutar la suite:

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

```
18
```

Una convención común es utilizar un solo carácter de subrayado para el nombre en el`for`encabezado si el nombre no se utiliza en la suite:

```
empty_rlist = None
def make_rlist(first, rest):
    """Make a recursive list from its first element and the rest."""
    return (first, rest)
```

Tenga en cuenta que un guión bajo es simplemente otro nombre en el entorno en lo que respecta al intérprete, pero tiene un significado convencional entre los programadores que indica que el nombre no aparecerá en ninguna expresión.

2.3.5 Abstracción de secuencias

Hemos introducido dos tipos de datos nativos que implementan la abstracción de secuencia: tuplas y rangos. Ambos satisfacen las condiciones con las que comenzamos esta sección: longitud y selección de elementos. Python incluye dos comportamientos más de tipos de secuencia que extienden la abstracción de secuencia.

Afiliación. Se puede comprobar la pertenencia de un valor a una secuencia. Python tiene dos operadores`in`y`not in`que evalúan a`True`o`False`dependiendo de si un elemento aparece en una secuencia.

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
(1, 8, 2, 8)
```

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
True
```

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
True
```

Todas las secuencias también tienen métodos llamados`index`y`count`, que devuelven el índice (o el recuento) de un valor en una secuencia.

Rebanar. Las secuencias contienen secuencias más pequeñas dentro de ellas. Observamos esta propiedad al desarrollar nuestra implementación de pares anidados, que descomponía una secuencia en su primer elemento y el resto._rebanada_ de una secuencia es cualquier tramo de la secuencia original, designado por un par de números enteros. Al igual que con el`range`constructor, el primer entero indica el índice inicial de la porción y el segundo indica uno más allá del índice final.

En Python, la segmentación de secuencias se expresa de forma similar a la selección de elementos, mediante corchetes. Los dos puntos separan los índices inicial y final. Se supone que cualquier límite que se omite es un valor extremo: 0 para el índice inicial y la longitud de la secuencia para el índice final.

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
(1, 8)
```

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
(8, 2, 8)
```

Enumerar estos comportamientos adicionales de la abstracción de secuencia de Python nos da la oportunidad de reflexionar sobre lo que constituye una abstracción de datos útil en general. La riqueza de una abstracción (es decir, cuántos comportamientos incluye) tiene consecuencias. Para los usuarios de una abstracción, los comportamientos adicionales pueden ser útiles. Por otro lado, satisfacer los requisitos de una abstracción rica con un nuevo tipo de datos puede ser un desafío. Asegurarnos de que nuestra implementación de listas recursivas admitiera estos comportamientos adicionales requeriría algo de trabajo. Otra consecuencia negativa de las abstracciones ricas es que los usuarios tardan más en aprenderlas.

Las secuencias tienen una gran capacidad de abstracción porque son tan comunes en la informática que se justifica el aprendizaje de algunos comportamientos complejos. En general, la mayoría de las abstracciones definidas por el usuario deben mantenerse lo más simples posible.

Lectura adicional. La notación de sectores admite una variedad de casos especiales, como valores iniciales negativos, valores finales y tamaños de paso. Una descripción completa aparece en la subsección llamadacortar una listaen Dive Into Python 3. En este capítulo, solo utilizaremos las características básicas descritas anteriormente.

2.3.6 Cadenas

Los valores de texto son quizás más fundamentales para la informática que los números pares. Por ejemplo, los programas de Python se escriben y almacenan como texto. El tipo de datos nativo para el texto en Python se denomina cadena y corresponde al constructor`str`.

Existen muchos detalles sobre cómo se representan, expresan y manipulan las cadenas en Python. Las cadenas son otro ejemplo de una abstracción rica, que requiere un compromiso sustancial por parte del programador para dominarla. Esta sección sirve como una introducción condensada a los comportamientos esenciales de las cadenas.

Los literales de cadena pueden expresar texto arbitrario, rodeado de comillas simples o dobles.

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
'I am string!'
```

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
"I've got an apostrophe"
```

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
' '
```

Ya hemos visto cadenas en nuestro código, como docstrings, en llamadas a`print`, y como mensajes de error en`assert`declaraciones.

Las cadenas satisfacen las dos condiciones básicas de una secuencia que presentamos al comienzo de esta sección: tienen una longitud y admiten la selección de elementos.

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
8
```

```
def first(s):
    """Return the first element of a recursive list s."""
    return s[0]
```

```
'k'
```

Los elementos de una cadena son en sí mismos cadenas que tienen un solo carácter. Un carácter es cualquier letra del alfabeto, signo de puntuación u otro símbolo. A diferencia de muchos otros lenguajes de programación, Python no tiene un tipo de carácter independiente; cualquier texto es una cadena y las cadenas que representan caracteres individuales tienen una longitud de 1.

Al igual que las tuplas, las cadenas también se pueden combinar mediante la suma y la multiplicación.

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
'Berkeley, CA'
```

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
'Shabu Shabu '
```

Afiliación. El comportamiento de las cadenas difiere de otros tipos de secuencias en Python. La abstracción de cadenas no se ajusta a la abstracción de secuencia completa que describimos para tuplas y rangos. En particular, el operador de pertenencia`in`Se aplica a cadenas, pero tiene un comportamiento completamente diferente que cuando se aplica a secuencias. Coincide con subcadenas en lugar de elementos.

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
True
```

De la misma manera, la`county`y`index`Los métodos en cadenas toman subcadenas como argumentos, en lugar de elementos de un solo carácter. El comportamiento de`count`es particularmente matizado; cuenta el número de ocurrencias no superpuestas de una subcadena en una cadena.

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
4
```

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
1
```

Literales multilineales. Las cadenas no se limitan a una sola línea. Las comillas triples delimitan literales de cadena que abarcan varias líneas. Ya hemos utilizado estas comillas triples de forma extensiva para cadenas de documentación.

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, "Readability counts."\nRead more: import this.'
```

En el resultado impreso arriba, el`\n`(pronunciado "*barra invertida en*") es un elemento único que representa una nueva línea. Aunque aparece como dos caracteres (barra invertida y "n"), se considera un único carácter a los efectos de selección de longitud y elementos.

Coerción de cuerdas. Se puede crear una cadena a partir de cualquier objeto en Python llamando a la función`str`Función constructora con un valor de objeto como argumento. Esta característica de las cadenas es útil para construir cadenas descriptivas a partir de objetos de varios tipos.

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
'2 is an element of (1, 8, 2, 8)'
```

El mecanismo por el cual un solo`str`La función puede aplicarse a cualquier tipo de argumento y devolver un valor apropiado es el tema de la sección posterior sobre funciones genéricas.

Métodos. El comportamiento de las cadenas en Python es extremadamente productivo debido a un amplio conjunto de métodos para devolver variantes de cadenas y buscar contenidos. A continuación, se presentan algunos de estos métodos mediante un ejemplo.

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
```

```
    return s[1]
```

```
True
```

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
'Robert De Niro'
```

```
def rest(s):
    """Return the rest of the elements of a recursive list s."""
    return s[1]
```

```
True
```

Lectura adicional. La codificación de texto en computadoras es un tema complejo. En este capítulo, nos abstraeremos de los detalles de cómo se representan las cadenas. Sin embargo, para muchas aplicaciones, los detalles particulares de cómo las computadoras codifican las cadenas son un conocimiento esencial.Secciones 4.1-4.3 de Dive Into Python 3Proporciona una descripción de las codificaciones de caracteres y Unicode.

2.3.7 Interfaces convencionales

Al trabajar con datos compuestos, hemos destacado cómo la abstracción de datos nos permite diseñar programas sin enredarnos en los detalles de las representaciones de datos, y cómo la abstracción nos preserva la flexibilidad para experimentar con representaciones alternativas. En esta sección, presentamos otro principio de diseño poderoso para trabajar con estructuras de datos: el uso de *interfaces convencionales*.

Una interfaz convencional es un formato de datos que se comparte entre muchos componentes modulares y que se pueden combinar para realizar el procesamiento de datos. Por ejemplo, si tenemos varias funciones que toman una secuencia como argumento y devuelven una secuencia como valor, podemos aplicar cada una de ellas a la salida de la siguiente en el orden que elijamos. De esta manera, podemos crear un proceso complejo encadenando una secuencia de funciones, cada una de las cuales es simple y específica.

Esta sección tiene un doble propósito: presentar la idea de organizar un programa alrededor de una interfaz convencional y demostrar ejemplos de procesamiento de secuencia modular.

Consideremos estos dos problemas, que a primera vista parecen estar relacionados sólo en su uso de secuencias:

1. Suma los miembros pares del primer$n$Números de Fibonacci.
2. Enumere las letras del acrónimo de un nombre, que incluye la primera letra de cada palabra en mayúscula.

Estos problemas están relacionados porque se pueden descomponer en operaciones simples que toman secuencias como entrada y generan secuencias como salida. Además, esas operaciones son ejemplos de métodos generales de cálculo sobre secuencias. Consideremos el primer problema. Se puede descomponer en los siguientes pasos:

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

El`fib`La función a continuación calcula los números de Fibonacci (ahora actualizados a partir de la definición del Capítulo 1 con un`for`declaración),

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

y un predicado`iseven`se puede definir utilizando el operador de resto entero,`%`.

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

Las funciones`map`y`filter`son operaciones sobre secuencias. Ya nos hemos encontrado con`map`, que aplica una función a cada elemento de una secuencia y recopila los resultados.`filter`La función toma una secuencia y devuelve aquellos elementos de una secuencia para los cuales un predicado es verdadero. Ambas funciones devuelven objetos intermedios.`map`y`filter`objetos, que son objetos iterables que pueden convertirse en tuplas o sumarse.

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
(6, -8)
```

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
35
```

Ahora podemos implementar`even_fib`, la solución a nuestro primer problema, en términos de`map`, `filter`, y`sum`.

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
3382
```

Ahora, consideremos el segundo problema. También se puede descomponer como una secuencia de operaciones de secuencia que incluyen`map`y`filter`:

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

Las palabras de una cadena se pueden enumerar mediante el`split`método de un objeto de cadena, que por defecto se divide en espacios.

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
('Spaces', 'between', 'words')
```

La primera letra de una palabra se puede recuperar utilizando el operador de selección, y se puede definir un predicado que determina si una palabra está en mayúscula utilizando el predicado incorporado.`isupper`.

```
counts = make_rlist(1, make_rlist(2, make_rlist(3, make_rlist(4, empty_rlist))))
first(counts)
```

```
rest(counts)
```

En este punto, nuestra función acrónimo se puede definir mediante`mapy``filter`.

```
rest(counts)
```

```
rest(counts)
```

```
('U', 'C', 'B', 'U', 'G', 'G')
```

Estas soluciones similares a problemas bastante diferentes muestran cómo combinar componentes generales que operan en la interfaz convencional de una secuencia utilizando los patrones computacionales generales de mapeo, filtrado y acumulación. La abstracción de secuencias nos permite especificar estas soluciones de manera concisa.

Expresar los programas como operaciones de secuencia nos ayuda a diseñar programas modulares. Es decir, nuestros diseños se construyen combinando piezas relativamente independientes, cada una de las cuales transforma una secuencia. En general, podemos fomentar el diseño modular proporcionando una biblioteca de componentes estándar junto con una interfaz convencional para conectar los componentes de formas flexibles.

Expresiones generadoras. El lenguaje Python incluye un segundo enfoque para procesar secuencias, llamado *expresiones generadoras.* que proporcionan una funcionalidad similar a`mapy``filter`, pero puede requerir menos definiciones de funciones.

Las expresiones generadoras combinan las ideas de filtrado y mapeo en un único tipo de expresión con la siguiente forma:

```
rest(counts)
```

Para evaluar una expresión de generador, Python evalúa la`<sequence expression>`, que debe devolver un valor iterable. Luego, para cada elemento en orden, el valor del elemento está vinculado a`<name>`, se evalúa la expresión de filtro y, si produce un valor verdadero, se evalúa la expresión del mapa.

El valor resultante de evaluar una expresión de generador es en sí mismo un valor iterable. Funciones de acumulación como`tuple`, `sum`, `max`, y`min`Puede tomar este objeto devuelto como argumento.

```
rest(counts)
```

```
rest(counts)
```

Las expresiones generadoras son una sintaxis especializada que utiliza la interfaz convencional de valores iterables, como las secuencias. Estas expresiones abarcan la mayor parte de la funcionalidad de`map`y`filter`, pero evitan crear realmente los valores de función que se aplican (o, incidentalmente, crear los marcos de entorno necesarios para aplicar esas funciones).

Reducir. En nuestros ejemplos usamos funciones específicas para acumular resultados, ya sea`tuple`o`sum`Los lenguajes de programación funcional (incluido Python) incluyen acumuladores generales de orden superior que tienen varios nombres. Python incluye`reduce`en el`functools`módulo que aplica una función de dos argumentos de forma acumulativa a los elementos de una secuencia de izquierda a derecha para reducir la secuencia a un valor. La siguiente expresión calcula el factorial 5.

```
rest(counts)
```

```
120
```

Utilizando esta forma más general de acumulación, también podemos calcular el producto de números de Fibonacci pares, además de la suma, utilizando secuencias como interfaz convencional.

```
rest(counts)
```

```
rest(counts)
```

```
123476336640
```

La combinación de procedimientos de orden superior correspondientes a`map`, `filter`, y`reduce`Aparecerá nuevamente en el Capítulo 4, cuando consideremos métodos para distribuir el cómputo entre múltiples computadoras.

Contents

2.4 Datos mutables **1**
2.4.1 Estado local 1
2.4.2 Los beneficios de la asignación no local 5
2.4.3 El costo de la asignación no local 7
2.4.4 Listas 9
2.4.5 Diccionarios 13
2.4.6 Ejemplo: Propagación de restricciones 15

\tableofcontents

2.4 Datos mutables

Hemos visto cómo la abstracción es vital para ayudarnos a lidiar con la complejidad de los sistemas grandes. La síntesis eficaz de programas también requiere principios organizativos que puedan guiarnos en la formulación del diseño general de un programa. En particular, necesitamos estrategias que nos ayuden a estructurar sistemas grandes de modo que sean *modular* , es decir, que puedan dividirse "naturalmente" en partes coherentes que puedan desarrollarse y mantenerse por separado.

Una técnica poderosa para crear programas modulares es introducir nuevos tipos de datos que puedan cambiar de estado con el tiempo. De esta manera, un único objeto de datos puede representar algo que evoluciona independientemente del resto del programa. El comportamiento de un objeto cambiante puede verse influenciado por su historia, al igual que una entidad en el mundo. Añadir estado a los datos es un ingrediente esencial de nuestro destino final en este capítulo: la programación orientada a objetos.

Los tipos de datos nativos que hemos introducido hasta ahora (números, booleanos, tuplas, rangos y cadenas) son todos tipos de *inmutable* objetos. Si bien los nombres pueden cambiar las vinculaciones a diferentes valores en el entorno durante el transcurso de la ejecución, los valores en sí no cambian. En esta sección, presentaremos una colección de *mudable* tipos de datos. Los objetos mutables pueden cambiar durante la ejecución de un programa.

2.4.1 Estado local

Nuestro primer ejemplo de un objeto mutable será una función que tiene un estado local. Ese estado cambiará durante la ejecución de un programa.

Para ilustrar lo que queremos decir con tener una función con estado local, modelemos la situación de retirar dinero de una cuenta bancaria. Lo haremos creando una función llamada`withdraw`, que toma como argumento una cantidad a retirar. Si hay suficiente dinero en la cuenta para realizar el retiro, entonces`withdraw`deberá devolver el saldo restante después del retiro. De

lo contrario,`withdraw`Debería devolver el mensaje`'Insufficient funds'`Por ejemplo, si comenzamos con $100 en la cuenta, nos gustaría obtener la siguiente secuencia de valores de retorno al llamar a retirar:

```
withdraw(25)
```

```
withdraw(25)
```

```
withdraw(60)
```

```
'Insufficient funds'
```

```
withdraw(15)
```

Observe que la expresión`withdraw(25)`, evaluada dos veces, produce valores diferentes. Este es un nuevo tipo de comportamiento para una función definida por el usuario: no es pura. Llamar a la función no solo devuelve un valor, sino que también tiene el efecto secundario de cambiar la función de alguna manera, de modo que la próxima llamada con el mismo argumento devuelva un resultado diferente. Todas nuestras funciones definidas por el usuario hasta ahora han sido funciones puras, a menos que llamaran a una función incorporada no pura. ¡Han permanecido puras porque no se les ha permitido realizar ningún cambio fuera de su marco de entorno local!

Para`withdraw`Para que tenga sentido, debe crearse con un saldo de cuenta inicial. La función`make_withdraw`es una función de orden superior que toma un saldo inicial como argumento. La función`withdraw`es su valor de retorno.

```
withdraw = make_withdraw(100)
```

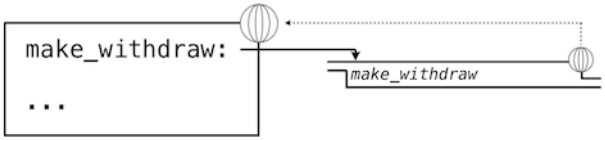
Una implementación de`make_withdraw`requiere un nuevo tipo de declaración: una`nonlocal`Declaración. Cuando llamamos`make_withdraw`, vinculamos el nombre`balance`al importe inicial. Luego definimos y devolvemos una función local,`withdraw`, que actualiza y devuelve el valor de`balance`Cuando se le llama.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

La parte novedosa de esta implementación es la`nonlocal`declaración, que exige que siempre que cambiemos la vinculación del nombre`balance`, el enlace se cambia en el primer cuadro en el que`balance`ya está atado. Recuerde que sin el`nonlocal`declaración, una declaración de asignación siempre vincularía un nombre en el primer marco del entorno.`nonlocal`La declaración indica que el

nombre aparece en algún lugar del entorno distinto del primer marco (local) o el último marco (global).

Podemos visualizar estos cambios con diagramas de entorno. Los siguientes diagramas de entorno ilustran los efectos de cada llamada, comenzando con la definición anterior. Abreviamos el código de ausencia en los valores de función y los árboles de expresión que no son centrales para nuestro análisis.

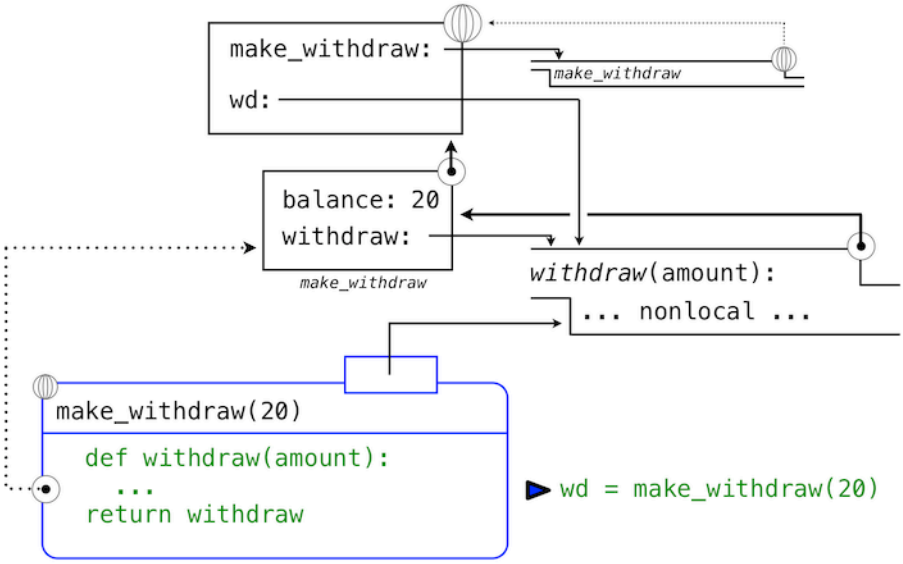


Nuestra declaración de definición tiene el efecto habitual: crea una nueva función definida por el usuario y vincula el nombre`make_withdraw`a esa función en el marco global.

A continuación, llamamos`make_withdraw`con un argumento de equilibrio inicial de`20`.

```
wd = make_withdraw(20)
```

Esta declaración de asignación vincula el nombre`wd`a la función devuelta en el marco global.

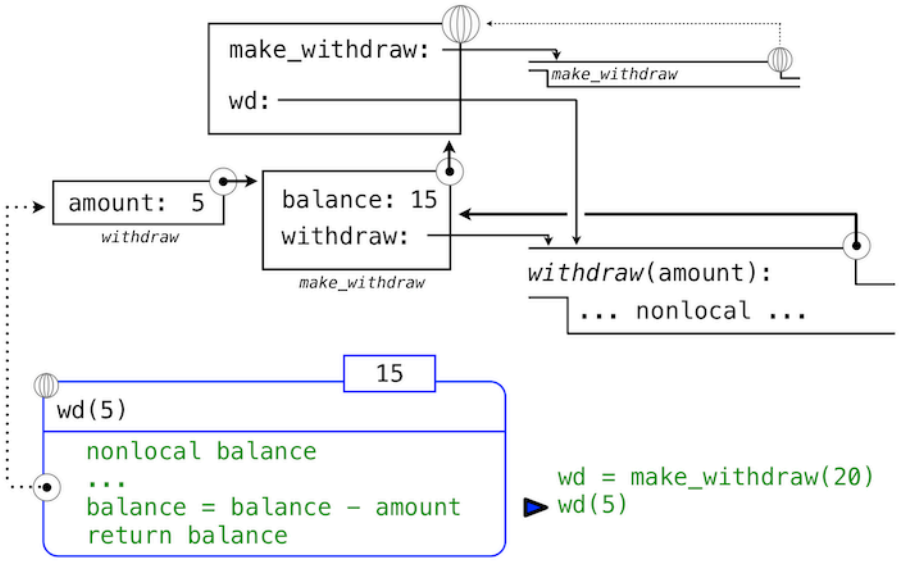


La función devuelta, (intrínsecamente) llamada *retirar* , está asociado con el entorno local para el *hacer_retirar* invocación en la que se definió. El nombre`balance`está vinculado a este entorno local. Fundamentalmente, solo habrá este vínculo único para el nombre`balance`a lo largo del resto de este ejemplo.

A continuación, evaluamos una expresión que llama *retirar* en una cantidad`5`.

```
wd(5)
```

El nombre`wd`está ligado a la *retirar* función, por lo que el cuerpo de *retirar* se evalúa en un nuevo entorno que amplía el entorno en el que *retirar* Se definió. Seguimiento del efecto de la evaluación *retirar* ilustra el efecto de una`nonlocal`Declaración en Python.



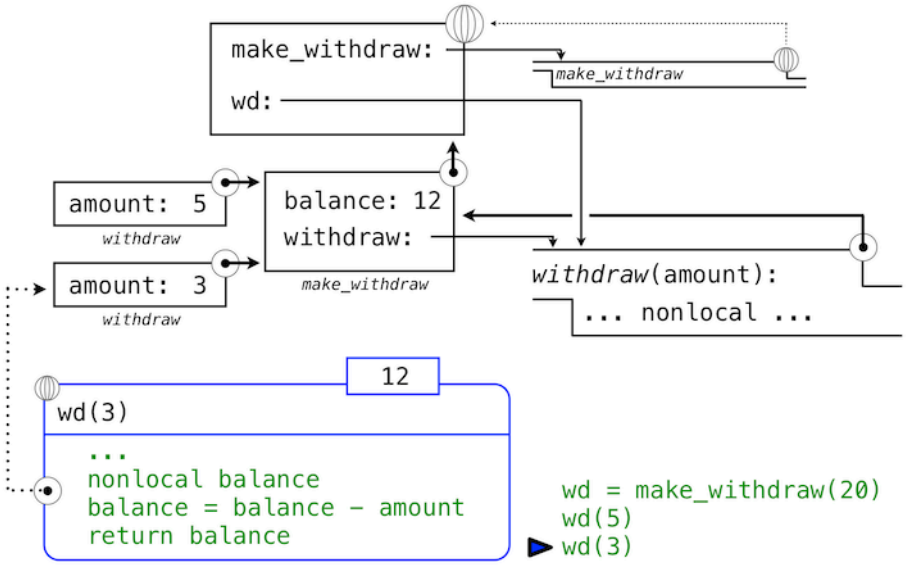
La declaración de asignación en *retirar* Normalmente crearía un nuevo enlace para`balance`en *retirar* marco local. En cambio, debido a la`nonlocal`declaración, la asignación encuentra el primer marco en el que`balance`ya estaba definido y vuelve a vincular el nombre en ese marco. Si`balance`no había sido previamente ligado a un valor, entonces el`nonlocal`Esta afirmación habría dado un error.

En virtud de cambiar la vinculación para`balance`, hemos cambiado el *retirar* función también. La próxima vez *retirar* se llama, el nombre`balance`se evaluará a`15`en lugar de`20`.

Cuando llamamos`wd`una segunda vez,

```
wd(3)
```

Vemos que los cambios en el valor ligado al nombre`balance`son acumulativos en las dos llamadas.



Aquí, el segundo llamado a *retirar* Sí, como siempre, se creó un segundo marco local. Sin embargo, ambos *retirar* Los marcos amplían el entorno para *hacer_retirar* , que contiene el enlace para`balance`. Por lo tanto, comparten ese vínculo de nombre particular. Llamado *retirar* tiene el efecto secundario de alterar el entorno que se extenderá por futuras llamadas a *retirar*.

Orientación práctica. Mediante la introducción`nonlocal`En el caso de las sentencias de asignación, hemos creado una función dual para las sentencias de asignación. O bien modifican los enlaces locales o bien modifican los enlaces no locales. De hecho, las sentencias de asignación ya tenían una función dual: o bien creaban nuevos enlaces o bien volvían a vincular nombres existentes. Las numerosas funciones de la asignación de Python pueden ocultar los efectos de la ejecución de una sentencia de asignación. Depende de usted, como programador, documentar su código con claridad para que los efectos de la asignación puedan ser comprendidos por otros.

2.4.2 Los beneficios de la asignación no local

La asignación no local es un paso importante en nuestro camino hacia ver un programa como una colección de tareas independientes y autónomas._objetos_ , que interactúan entre sí pero cada uno gestiona su propio estado interno.

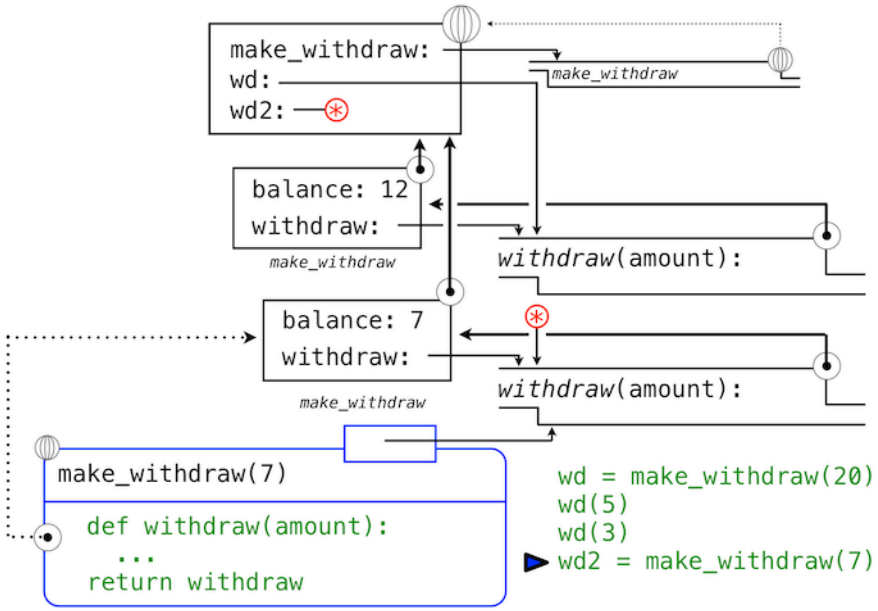
En particular, la asignación no local nos ha dado la capacidad de mantener un estado que es local para una función, pero que evoluciona a lo largo de llamadas sucesivas a esa función.`balance`asociado con una función de retiro en particular se comparte entre todas las llamadas a esa función. Sin embargo, el enlace para el saldo asociado con una instancia de retiro es inaccesible para el resto del programa. Solo *retirar* está asociado con el marco para *hacer_retirar* en el

que se definió. Si *hacer_retirar* se llama nuevamente, luego creará un marco separado con un enlace separado para`balance`.

Podemos continuar con nuestro ejemplo para ilustrar este punto. Un segundo llamado a *hacer_retirar* devuelve un segundo *retirar* función que está asociada con otro entorno.

```
wd2 = make_withdraw(7)
```

Este segundo *retirar* La función está ligada al nombre`wd2`en el marco global. Hemos abreviado la línea que representa este enlace con un asterisco. Ahora, vemos que, de hecho, hay dos enlaces para el nombre`balance`. El nombre`wd`todavía está ligado a un *retirar* función con un equilibrio de`12`, mientras`wd2`Está ligado a un nuevo *retirar* función con un equilibrio de`7`.

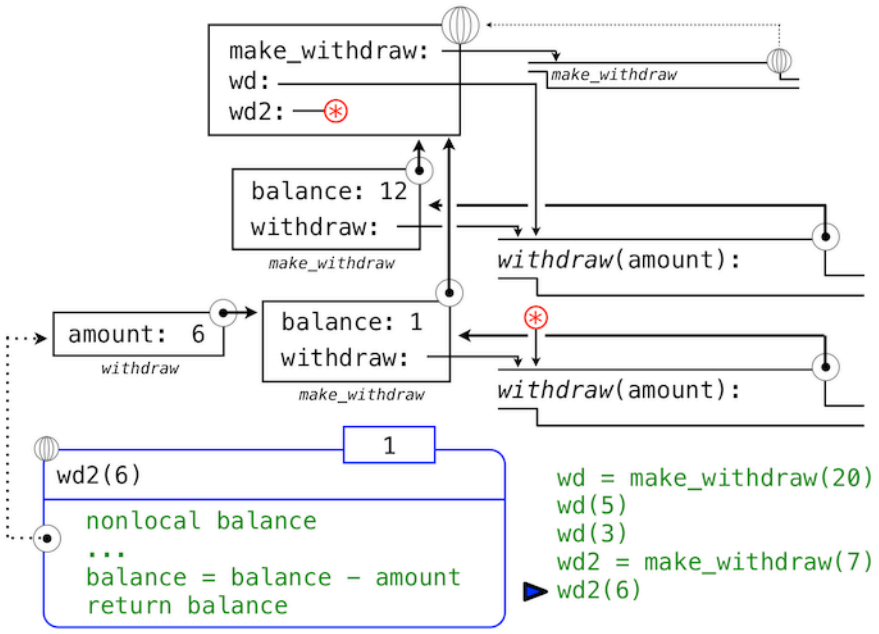


Finalmente, llamamos al segundo *retirar* obligado a`wd2`:

```
withdraw(25)
```

```
1
```

Esta llamada cambia el enlace de su no local`balance`nombre, pero no afecta al primero *retirar* ligado al nombre`wd`en el marco global.



De esta manera, cada instancia de *retirar* mantiene su propio estado de saldo, pero ese estado es inaccesible para cualquier otra función del programa. Al considerar esta situación a un nivel superior, hemos creado una abstracción de una cuenta bancaria que administra sus propios elementos internos, pero se comporta de una manera que modela las cuentas en el mundo: cambia con el tiempo en función de su propio historial de solicitudes de retiro.

2.4.3 El costo de la asignación no local

Nuestro modelo de entorno computacional se extiende claramente para explicar los efectos de la asignación no local. Sin embargo, la asignación no local introduce algunos matices importantes en la forma en que pensamos sobre los nombres y los valores.

Antes, nuestros valores no cambiaban; sólo cambiaban nuestros nombres y nuestros vínculos. Cuando dos nombres`a`y`b`Ambos estaban ligados al valor`4`, no importaba si estaban atados al mismo`4`o diferente`4`'s. Hasta donde pudimos ver, solo había uno`4`objeto que nunca cambió.

Sin embargo, las funciones con estado no se comportan de esta manera. Cuando dos nombres`wd`y`wd2`Ambos están ligados a un *retirar* función, es *hace* No importa si están ligados a la misma función o a instancias diferentes de esa función. Consideremos el siguiente ejemplo, que contrasta con el que acabamos de analizar.

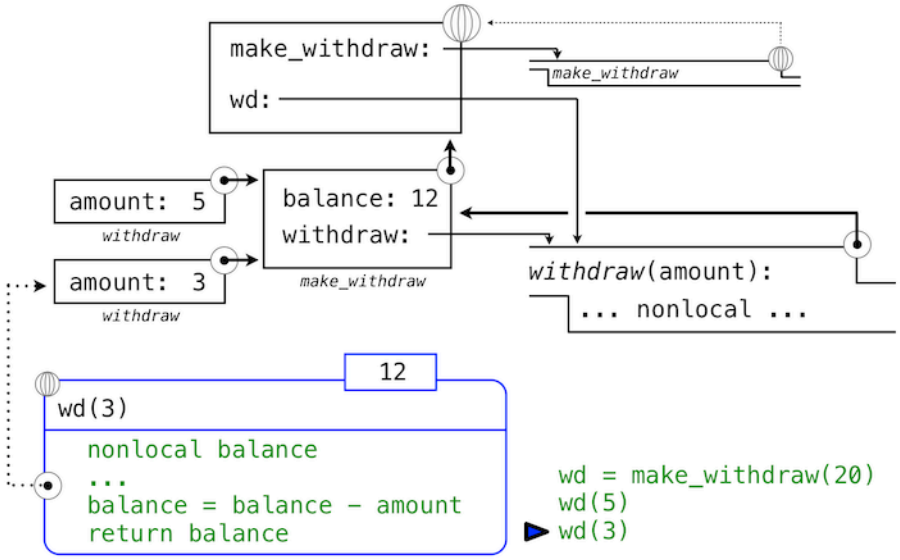
```
withdraw(25)
```

```
11
```

```
withdraw(25)
```

```
10
```

En este caso, llamar a la función nombrada por`wd2`¿Cambió el valor de la función nombrada por?`wd`, porque ambos nombres hacen referencia a la misma función. El diagrama del entorno después de ejecutar estas instrucciones muestra este hecho.



No es inusual que dos nombres hagan referencia al mismo valor en el mundo, y lo mismo ocurre en nuestros programas. Pero, como los valores cambian con el tiempo, debemos tener mucho cuidado de entender el efecto de un cambio en otros nombres que puedan hacer referencia a esos valores.

La clave para analizar correctamente el código con asignación no local es recordar que solo las llamadas a funciones pueden introducir nuevos marcos. Las instrucciones de asignación siempre cambian los enlaces en los marcos existentes. En este caso, a menos que *hacer_retirar* se llama dos veces, solo puede haber un enlace para`balance`.

Misma identidad y cambio. Estas sutilezas surgen porque, al introducir funciones no puras que cambian el entorno no local, hemos cambiado la naturaleza de las expresiones. Una expresión que contiene solo llamadas a funciones puras es *referencialmente transparente* ; su valor no cambia si sustituimos una de sus subexpresiones con el valor de esa subexpresión.

Las operaciones de revinculación violan las condiciones de transparencia referencial porque hacen más que devolver un valor: cambian el entorno. Cuando introducimos una revinculación arbitraria, nos topamos con un espinoso problema epistemológico: qué significa que dos valores sean iguales. En nuestro

modelo de entorno de computación, dos funciones definidas por separado no son iguales, porque los cambios en una pueden no reflejarse en la otra.

En general, siempre que no modifiquemos nunca los objetos de datos, podemos considerar que un objeto de datos compuesto es precisamente la totalidad de sus partes. Por ejemplo, un número racional se determina dando su numerador y su denominador. Pero esta visión ya no es válida en presencia de un cambio, donde un objeto de datos compuesto tiene una "identidad" que es algo diferente de las partes que lo componen. Una cuenta bancaria sigue siendo "la misma" cuenta bancaria incluso si cambiamos el saldo haciendo un retiro; a la inversa, podríamos tener dos cuentas bancarias que tengan el mismo saldo, pero sean objetos diferentes.

A pesar de las complicaciones que introduce, la asignación no local es una herramienta poderosa para crear programas modulares. Diferentes partes de un programa, que corresponden a diferentes marcos de entorno, pueden evolucionar por separado a lo largo de la ejecución del programa. Además, al usar funciones con estado local, podemos implementar tipos de datos mutables. En el resto de esta sección, presentamos algunos de los tipos de datos integrados más útiles en Python, junto con métodos para implementar esos tipos de datos usando funciones con asignación no local.

2.4.4 Listas

El`list`es el tipo de secuencia más útil y flexible de Python. Una lista es similar a una tupla, pero es mutable. Las llamadas a métodos y las instrucciones de asignación pueden cambiar el contenido de una lista.

Podemos presentar muchas operaciones de edición de listas mediante un ejemplo que ilustra la historia de los naipes (drásticamente simplificado). Los comentarios en los ejemplos describen el efecto de cada invocación de método.

Las cartas de juego se inventaron en China, quizás alrededor del siglo IX. Una de las primeras barajas tenía tres palos, que correspondían a las denominaciones de dinero.

```
withdraw(25)
```

A medida que las cartas migraron a Europa (quizás a través de Egipto), solo el palo de monedas permaneció en las barajas españolas (*oro*).

```
withdraw(25)
```

```
'myriad'
```

```
withdraw(25)
```

Se agregaron tres trajes más (evolucionaron en nombre y diseño con el tiempo),

Número de serie 16

y los italianos las llamaban espadas *espadas*.

```
withdraw(25)
```

dando los palos de una baraja de cartas tradicional italiana.

Número de serie 18

```
['coin', 'cup', 'spade', 'club']
```

La variante francesa que utilizamos hoy en Estados Unidos cambia los dos primeros:

```
withdraw(25)
```

```
['heart', 'diamond', 'spade', 'club']
```

También existen métodos para insertar, ordenar e invertir listas. Todos estos *operaciones de mutación* cambian el valor de la lista; no crean nuevos objetos de lista.

Compartir e identidad. Debido a que hemos estado cambiando una sola lista en lugar de crear nuevas listas, el objeto vinculado al nombre`chinese_suits`También ha cambiado, porque es el mismo objeto de lista que estaba vinculado a`suits`.

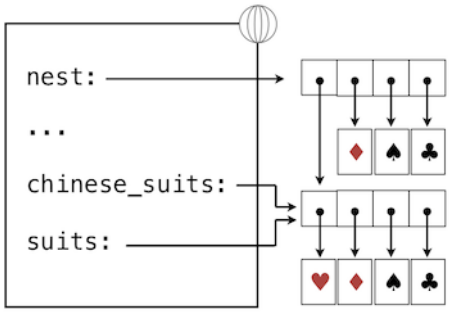
```
withdraw(60)
```

```
['heart', 'diamond', 'spade', 'club']
```

Las listas se pueden copiar utilizando el`list`Función constructora. Los cambios en una lista no afectan a otra, a menos que compartan estructura.

```
withdraw(60)
```

Después de esta tarea final, nos quedamos con el siguiente entorno, donde las listas se representan utilizando notación de caja y puntero.



De acuerdo a este entorno, cambiar la lista a la que hace referencia`suits`afectará a la lista anidada que es el primer elemento de`nest`, pero no los demás elementos.

```
withdraw(60)
```

```
[['heart', 'diamond', 'Joker', 'spade', 'club'], 'diamond', 'spade', 'club']
```

Y de la misma manera, deshaciendo este cambio en el primer elemento de`nest`cambiará`suit`también.

```
withdraw(60)
```

```
'Joker'
```

```
withdraw(60)
```

```
['heart', 'diamond', 'spade', 'club']
```

Como resultado de esta última invocación de la`pop`Método, volvemos al entorno representado anteriormente.

Dado que dos listas pueden tener el mismo contenido pero, de hecho, ser listas diferentes, necesitamos un medio para comprobar si dos objetos son iguales. Python incluye dos operadores de comparación, llamados`is`y`is not`, que prueban si dos expresiones de hecho evalúan el objeto idéntico. Dos objetos son idénticos si son iguales en su valor actual, y cualquier cambio en uno siempre se reflejará en el otro. La identidad es una condición más fuerte que la igualdad.

```
withdraw(60)
```

```
True
```

```
withdraw(60)
```

```
False
```

```
withdraw(60)
```

```
True
```

Las dos últimas comparaciones ilustran la diferencia entre`is`y`==`El primero verifica la identidad, mientras que el segundo verifica la igualdad de contenidos.

Listas por comprensión. Una comprensión de listas utiliza una sintaxis extendida para crear listas, análoga a la sintaxis de las expresiones generadoras.

Por ejemplo, el`unicodedata`El módulo rastrea los nombres oficiales de cada carácter del alfabeto Unicode. Podemos buscar los caracteres correspondientes a los nombres, incluidos los de los palos de las cartas.

```
withdraw(60)
```

```
[' ', ' ', ' ', ' ']
```

Las listas por comprensión refuerzan el paradigma del procesamiento de datos utilizando la interfaz convencional de secuencias, como`list`es un tipo de datos de secuencia.

Lectura adicional. Dive Into Python 3 tiene un capítulo sobrecomprensionesque incluye ejemplos de cómo navegar por el sistema de archivos de una computadora usando Python. El capítulo presenta el`os`módulo que, por ejemplo, puede enumerar el contenido de los directorios. Este material no forma parte del

curso, pero se recomienda para cualquier persona que desee aumentar su conocimiento de Python.

Implementación. Las listas son secuencias, como las tuplas. El lenguaje Python no nos da acceso a la implementación de listas, solo a la abstracción de secuencias y a los métodos de mutación que hemos presentado en esta sección. Para superar esta barrera de abstracción impuesta por el lenguaje, podemos desarrollar una implementación funcional de listas, nuevamente utilizando una representación recursiva. Esta sección también tiene un segundo propósito: profundizar nuestra comprensión de las funciones de envío.

Implementaremos una lista como una función que tiene una lista recursiva como su estado local. Las listas deben tener una identidad, como cualquier valor mutable. En particular, no podemos usar`None`para representar una lista mutable vacía, porque dos listas vacías no son valores idénticos (por ejemplo, agregar a una no agrega a la otra), pero`None is None`. Por otro lado, dos funciones diferentes que cada uno tiene`empty_rlist`ya que su estado local bastará para distinguir dos listas vacías.

Nuestra lista mutable es una función de envío, al igual que nuestra implementación funcional de un par era una función de envío. Verifica el "mensaje" de entrada con mensajes conocidos y toma una acción apropiada para cada entrada diferente. Nuestra lista mutable responde a cinco mensajes diferentes. Los primeros dos implementan los comportamientos de la abstracción de secuencia. Los dos siguientes agregan o eliminan el primer elemento de la lista. El mensaje final devuelve una representación de cadena de todo el contenido de la lista.

```
withdraw(60)
```

También podemos agregar una función de conveniencia para construir una lista recursiva implementada funcionalmente a partir de cualquier secuencia incorporada, simplemente agregando cada elemento en orden inverso.

```
withdraw(15)
```

En la definición anterior, la función`reversed`toma y devuelve un valor iterable; es otro ejemplo de una función que utiliza la interfaz convencional de secuencias.

En este punto, podemos construir listas implementadas funcionalmente. Tenga en cuenta que la lista en sí es una función.

```
withdraw(15)
```

```
withdraw(15)
```

```
"('heart', ('diamond', ('spade', ('club', None))))"
```

Además, podemos pasar mensajes a la lista.`s`que cambian su contenido, por ejemplo eliminando el primer elemento.

```
withdraw(15)
```

```
'heart'
```

```
withdraw(15)
```

```
"('diamond', ('spade', ('club', None)))"
```

En principio, las operaciones`push_first`y`pop_first`Basta con hacer cambios arbitrarios en una lista. Siempre podemos vaciar la lista por completo y luego reemplazar su contenido anterior con el resultado deseado.

Paso de mensajes. Con algo de tiempo, podríamos implementar muchas operaciones de mutación útiles de las listas de Python, como`extend`y`insert`Tendríamos una opción: podríamos implementarlos todos como funciones, que utilizan los mensajes existentes.`pop_first`y`push_first`para realizar todos los cambios. Alternativamente, podríamos agregar más`elif`cláusulas al cuerpo de`dispatch`, cada uno comprobando si hay un mensaje (por ejemplo,`'extend'`) y aplicar el cambio apropiado a`contents`directamente.

Este segundo enfoque, que encapsula la lógica de todas las operaciones sobre un valor de datos dentro de una función que responde a diferentes mensajes, se denomina paso de mensajes. Un programa que utiliza el paso de mensajes define funciones de despacho, cada una de las cuales puede tener un estado local, y organiza el cálculo pasando "mensajes" como primer argumento a esas funciones. Los mensajes son cadenas que corresponden a comportamientos particulares.

Uno podría imaginar que enumerar todos estos mensajes por nombre en el cuerpo del mensaje`dispatch`Se volvería tedioso y propenso a errores. Los diccionarios de Python, que se presentan en la siguiente sección, proporcionan un tipo de datos que nos ayudará a gestionar la asignación entre mensajes y operaciones.

2.4.5 Diccionarios

Los diccionarios son el tipo de datos integrado de Python para almacenar y manipular relaciones de correspondencia. Un diccionario contiene pares clave-valor, donde tanto las claves como los valores son objetos. El propósito de un diccionario es proporcionar una abstracción para almacenar y recuperar valores que no están indexados por números enteros consecutivos, sino por claves descriptivas.

Las cadenas suelen servir como claves, porque son nuestra representación convencional de los nombres de las cosas. Este literal de diccionario proporciona los valores de varios números romanos.

```
withdraw(15)
```

La búsqueda de valores por sus claves utiliza el operador de selección de elementos que aplicamos anteriormente a las secuencias.

```
withdraw(15)
```

```
10
```

Un diccionario puede tener como máximo un valor para cada clave. La adición de nuevos pares clave-valor y la modificación del valor existente de una clave se pueden realizar mediante instrucciones de asignación.

```
withdraw(15)
```

```
{'I': 1, 'X': 10, 'L': 50, 'V': 5}
```

Tenga en cuenta que`'L'`No se agregó al final de la salida anterior. Los diccionarios son colecciones desordenadas de pares clave-valor. Cuando imprimimos un diccionario, las claves y los valores se representan en algún orden, pero como usuarios del lenguaje no podemos predecir cuál será ese orden.

La abstracción del diccionario también admite varios métodos de iteración del contenido del diccionario en su conjunto. Los métodos`keys`, `values`, y`items`todos devuelven valores iterables.

```
withdraw(15)
```

```
66
```

Una lista de pares clave-valor se puede convertir en un diccionario llamando al método`dict`función constructora.

```
withdraw(15)
```

```
{3: 9, 4: 16, 5: 25}
```

Los diccionarios tienen algunas restricciones:

- Una clave de un diccionario no puede ser un objeto de un tipo incorporado mutable.
- Puede haber como máximo un valor para una clave dada.

Esta primera restricción está relacionada con la implementación subyacente de los diccionarios en Python. Los detalles de esta implementación no son tema de este curso. Intuitivamente, considere que la clave le indica a Python dónde encontrar ese par clave-valor en la memoria; si la clave cambia, la ubicación del par puede perderse.

La segunda restricción es una consecuencia de la abstracción del diccionario, que está diseñada para almacenar y recuperar valores de claves. Solo podemos recuperar *el* valor de una clave si existe como máximo un valor de este tipo en el diccionario.

Un método útil implementado por los diccionarios es`get`, que devuelve el valor de una clave, si la clave está presente, o un valor predeterminado. Los argumentos de`get`son la clave y el valor predeterminado.

```
withdraw = make_withdraw(100)
```

```
0
```

```
withdraw = make_withdraw(100)
```

```
5
```

Los diccionarios también tienen una sintaxis de comprensión análoga a la de las listas y las expresiones generadoras. La evaluación de una comprensión de diccionario genera un nuevo objeto de diccionario.

```
withdraw = make_withdraw(100)
```

```
{3: 9, 4: 16, 5: 25}
```

Implementación. Podemos implementar un tipo de datos abstracto que se ajuste a la abstracción del diccionario como una lista de registros, cada uno de los cuales es una lista de dos elementos que consta de una clave y el valor asociado.

```
withdraw = make_withdraw(100)
```

Nuevamente, utilizamos el método de paso de mensajes para organizar nuestra implementación. Hemos admitido cuatro mensajes:`getitem`, `setitem`, `keys`, y`values`Para buscar un valor para una clave, iteramos a través de los registros para encontrar una clave coincidente. Para insertar un valor para una clave, iteramos a través de los registros para ver si ya hay un registro con esa clave. Si no, formamos un nuevo registro. Si ya hay un registro con esta clave, establecemos el valor del registro en el nuevo valor designado.

Ahora podemos usar nuestra implementación para almacenar y recuperar valores.

```
withdraw = make_withdraw(100)
```

```
9
```

```
withdraw = make_withdraw(100)
```

```
16
```

```
withdraw = make_withdraw(100)
```

```
(3, 4)
```

```
withdraw = make_withdraw(100)
```

```
(9, 16)
```

Esta implementación de un diccionario es *no* Optimizado para una búsqueda rápida de registros, porque cada respuesta al mensaje`'getitem'`debe iterar a través de toda la lista de`records`El tipo de diccionario incorporado es considerablemente más eficiente.

2.4.6 Ejemplo: Propagación de restricciones

Los datos mutables nos permiten simular sistemas con cambios, pero también nos permiten crear nuevos tipos de abstracciones. En este ejemplo ampliado, combinamos asignaciones no locales, listas y diccionarios para crear un *sistema*

basado en restricciones que admite el cálculo en múltiples direcciones. Expresar programas como restricciones es un tipo de *programación declarativa* , en el que un programador declara la estructura de un problema a resolver, pero abstrae los detalles de exactamente cómo se calcula la solución del problema.

Los programas informáticos se organizan tradicionalmente como cálculos unidireccionales, que realizan operaciones sobre argumentos preestablecidos para producir los resultados deseados. Por otro lado, a menudo queremos modelar sistemas en términos de relaciones entre cantidades. Por ejemplo, anteriormente consideramos la ley de los gases ideales, que relaciona la presión (`p`), volumen (`v`), cantidad (`n`) y la temperatura (`t`) de un gas ideal a través de la constante de Boltzmann (`k`):

```
withdraw = make_withdraw(100)
```

Esta ecuación no es unidireccional. Dadas cuatro de las cantidades, podemos utilizar esta ecuación para calcular la quinta. Sin embargo, traducir la ecuación a un lenguaje informático tradicional nos obligaría a elegir una de las cantidades para calcular en función de las otras cuatro. Por lo tanto, una función para calcular la presión no podría utilizarse para calcular la temperatura, aunque los cálculos de ambas cantidades surjan de la misma ecuación.

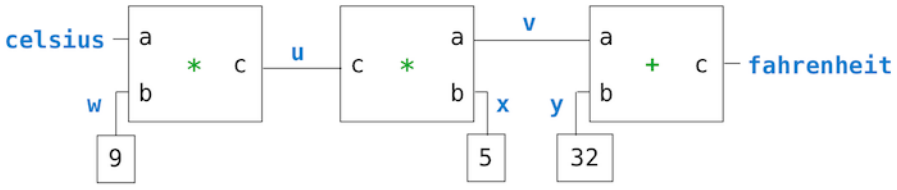
En esta sección, esbozamos el diseño de un modelo general de relaciones lineales. Definimos restricciones primitivas que se cumplen entre cantidades, como una`adder(a, b, c)`restricción que refuerza la relación matemática`a + b = c`.

También definimos un medio de combinación, de modo que las restricciones primitivas se puedan combinar para expresar relaciones más complejas. De esta manera, nuestro programa se asemeja a un lenguaje de programación. Combinamos restricciones construyendo una red en la que las restricciones se unen mediante conectores. Un conector es un objeto que "contiene" un valor y puede participar en una o más restricciones.

Por ejemplo, sabemos que la relación entre las temperaturas Fahrenheit y Celsius es:

```
withdraw = make_withdraw(100)
```

Esta ecuación es una restricción compleja entre`c`y`f`Se puede pensar en una restricción de este tipo como una red que consta de primitivas`adder`, `multiplier`, y`constant`Restricciones.



En esta figura, vemos a la izquierda una caja multiplicadora con tres terminales,

etiquetada`a`, `b`, y`c`Estos conectan el multiplicador al resto de la red de la siguiente manera:`a`El terminal está conectado a un conector`celsius`, que mantendrá la temperatura Celsius.`b`El terminal está conectado a un conector`w`, que también está vinculado a un cuadro constante que contiene`9`. El`c`terminal, que el cuadro multiplicador restringe a ser el producto de`a`y`b`, está vinculado a la`c`terminal de otra caja multiplicadora, cuya`b`está conectado a una constante`5`y cuyo`a`está conectado a uno de los términos en la restricción de suma.

El cálculo de una red de este tipo se realiza de la siguiente manera: cuando se le asigna un valor a un conector (por parte del usuario o por parte de una caja de restricción a la que está vinculado), este activa todas sus restricciones asociadas (excepto la restricción que lo acaba de activar) para informarles que tiene un valor. Cada caja de restricción activada sondea a sus conectores para ver si hay suficiente información para determinar un valor para un conector. Si es así, la caja activa ese conector, que luego activa todas sus restricciones asociadas, y así sucesivamente. Por ejemplo, en la conversión entre Celsius y Fahrenheit,`w`, `x`, y`y`se establecen inmediatamente mediante los cuadros constantes a`9`, `5`, y`32`, respectivamente. Los conectores activan los multiplicadores y el sumador, que determinan que no hay suficiente información para continuar. Si el usuario (o alguna otra parte de la red) configura el`celsius`conector a un valor (por ejemplo`25`), el multiplicador más a la izquierda se activará y se estableceráu`a`25 `* 9 = 225`. Entoncesudespierta el segundo multiplicador, que estableceva`45`, yvDespierta a la víbora, que establece el`fahrenheit`conector a`77`.

Utilizando el sistema de restricciones. Para utilizar el sistema de restricciones para realizar el cálculo de temperatura descrito anteriormente, primero creamos dos conectores con nombre,`celsius`y`fahrenheit`, llamando al`make_connector`constructor.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

Luego, conectamos estos conectores en una red que refleja la figura anterior. La función`make_converter`reúne los distintos conectores y restricciones de la red.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
```

```
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw

def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

Utilizaremos un sistema de paso de mensajes para coordinar las restricciones y los conectores. En lugar de utilizar funciones para responder a los mensajes, utilizaremos diccionarios. Un diccionario de envío tendrá claves con valores de cadena que denotan los mensajes que acepta. Los valores asociados a esas claves serán las respuestas a esos mensajes.

Las restricciones son diccionarios que no contienen estados locales. Sus respuestas a los mensajes son funciones no puras que modifican los conectores que restringen.

Los conectores son diccionarios que contienen un valor actual y responden a mensajes que manipulan ese valor. Las restricciones no cambiarán el valor de los conectores directamente, sino que lo harán enviando mensajes, de modo que el conector pueda notificar a otras restricciones en respuesta al cambio. De esta manera, un conector representa un número, pero también encapsula el comportamiento del conector.

Un mensaje que podemos enviar a un conector es establecer su valor. Aquí, nosotros (el`'user'`) establece el valor de`celsius`a`25`.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

```
Celsius = 25
Fahrenheit = 77.0
```

No sólo el valor de`celsius`cambiar a`25`, pero su valor se propaga a través de la red, por lo que el valor de`fahrenheit`También se modificó. Estos cambios se imprimen porque nombramos estos dos conectores cuando los construimos.

Ahora podemos intentar configurarlo`fahrenheit`a un nuevo valor, digamos`212`.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

```
Contradiction detected: 77.0 vs 212
```

El conector se queja de que ha detectado una contradicción: su valor es`77.0`, y alguien está intentando configurarlo en`212`Si realmente queremos reutilizar la red con nuevos valores, podemos decirle`celsius`para olvidar su antiguo valor:

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

```
Celsius is forgotten
Fahrenheit is forgotten
```

El conector`celsius`encuentra que el`user`, que fijó su valor originalmente, ahora está retractándose de ese valor, por lo que`celsius`acepta perder su valor e informa al resto de la red de este hecho. Esta información finalmente se propaga a`fahrenheit`, que ahora descubre que no tiene motivos para seguir creyendo que su propio valor es`77`. Por lo tanto, también renuncia a su valor.

Ahora que`fahrenheit`No tiene ningún valor, somos libres de configurarlo.`212`:

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

```
Fahrenheit = 212
```

```
Celsius = 100.0
```

Este nuevo valor, al propagarse a través de la red, obliga a`celsius`tener un valor de`100`Hemos utilizado la misma red para calcular`celsius`dado`fahrenheit`y para calcular`fahrenheit`dado`celsius`Esta no direccionalidad del cálculo es la característica distintiva de los sistemas basados en restricciones.

Implementación del sistema de restricciones. Como hemos visto, los conectores son diccionarios que asignan nombres de mensajes a valores de funciones y datos. Implementaremos conectores que respondan a los siguientes mensajes:

- `connector['set_val'](source, value)`indica que el`source`está solicitando al conector que establezca su valor actual en`value`.
- `connector['has_val']()`devuelve si el conector ya tiene un valor.
- `connector['val']`es el valor actual del conector.
- `connector['forget'](source)`le dice al conector que el`source`está pidiendo que olvide su valor.
- `connector['connect'](source)`le dice al conector que participe en una nueva restricción, el`source`.

Las restricciones también son diccionarios, que reciben información de los conectores mediante dos mensajes:

- `constraint['new_val']()`indica que algún conector que está conectado a la restricción tiene un nuevo valor.
- `constraint['forget']()`indica que algún conector que está conectado a la restricción ha olvidado su valor.

Cuando las restricciones reciben estos mensajes, los propagan adecuadamente a otros conectores.

El`adder`La función construye una restricción sumadora sobre tres conectores, donde los dos primeros deben sumarse al tercero:`a + b = c`Para admitir la propagación de restricciones multidireccionales, el sumador también debe especificar que resta`a`de`c`Llegar`b`y asimismo resta`b`de`c`Llega`a`.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

Nos gustaría implementar una restricción ternaria genérica (de tres vías), que utiliza los tres conectores y las tres funciones de`adder`para crear una restricción

que acepte`new_val`y`forget`mensajes. La respuesta a los mensajes son funciones locales, que se colocan en un diccionario llamado`constraint`.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

El diccionario llamado`constraint`es un diccionario de envío, pero también el objeto de restricción en sí. Responde a los dos mensajes que reciben las restricciones, pero también se pasa como`source`argumento en llamadas a sus conectores.

La función local de la restricción`new_value`se llama siempre que se informa a la restricción de que uno de sus conectores tiene un valor. Esta función primero verifica si ambos`a`y`b`tener valores. Si es así, te lo dice`c`para establecer su valor en el valor de retorno de la función`ab`, que es`add`en el caso de una`adder`La restricción pasa *sí mismo* (`constraint`) como el`source`argumento del conector, que es el objeto sumador. Si`a`y`b`no tienen ambos valores, entonces la restricción verifica`a`y`c`, etcétera.

Si se informa a la restricción que uno de sus conectores ha olvidado su valor, solicita que todos sus conectores olviden ahora sus valores. (En realidad, solo se pierden aquellos valores que se establecieron mediante esta restricción).

A`multiplier`es muy similar a un`adder`.

```
def make_withdraw(balance):
    """Return a withdraw function that draws down balance with each call."""
    def withdraw(amount):
        nonlocal balance                 # Declare the name "balance" nonlocal
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount       # Re-bind the existing balance name
        return balance
    return withdraw
```

Una constante también es una restricción, pero una a la que nunca se le envían mensajes, porque implica solo un único conector que establece en la construcción.

```
wd = make_withdraw(20)
```

Estas tres restricciones son suficientes para implementar nuestra red de conversión de temperatura.

Representando conectores. Un conector se representa como un diccionario que contiene un valor, pero también tiene funciones de respuesta con estado local. El conector debe realizar un seguimiento de la`informant`que le dio su valor actual, y una lista de`constraints`en el que participa.

El constructor`make_connector`tiene funciones locales para establecer y olvidar valores, que son las respuestas a los mensajes de restricciones.

```
wd = make_withdraw(20)
```

Un conector es nuevamente un diccionario de envío para los cinco mensajes que utilizan las restricciones para comunicarse con los conectores. Cuatro respuestas son funciones y la respuesta final es el valor en sí.

La función local`set_value`Se llama cuando hay una solicitud para establecer el valor del conector. Si el conector no tiene un valor actualmente, establecerá su valor y lo recordará como`informant`la restricción de origen que solicitó que se estableciera el valor. Luego, el conector notificará a todas las restricciones participantes, excepto a la restricción que solicitó que se estableciera el valor. Esto se logra utilizando la siguiente función iterativa.

```
wd = make_withdraw(20)
```

Si se le pide a un conector que olvide su valor, llama a la función local`forget-value`, que primero verifica que la solicitud provenga de la misma restricción que estableció el valor originalmente. Si es así, el conector informa a sus restricciones asociadas sobre la pérdida del valor.

La respuesta al mensaje`has_val`indica si el conector tiene un valor. La respuesta al mensaje`connect`agrega la restricción de origen a la lista de restricciones.

El programa de restricciones que hemos diseñado introduce muchas ideas que aparecerán nuevamente en la programación orientada a objetos. Las restricciones y los conectores son abstracciones que se manipulan a través de mensajes. Cuando se cambia el valor de un conector, se cambia a través de un mensaje que no solo cambia el valor, sino que lo valida (verificando la fuente) y propaga sus efectos (informando a otras restricciones). De hecho, utilizaremos una arquitectura similar de diccionarios con claves con valores de cadena y valores funcionales para implementar un sistema orientado a objetos más adelante en este capítulo.

Contents

2.5 Programación orientada a objetos **1**
2.5.1 Objetos y clases 2
2.5.2 Definición de clases 3
2.5.3 Paso de mensajes y expresiones de puntos 6
2.5.4 Atributos de clase 8
2.5.5 Herencia 10
2.5.6 Uso de la herencia 11
2.5.7 Herencia múltiple 12
2.5.8 El papel de los objetos 14

\tableofcontents

2.5 Programación orientada a objetos

La programación orientada a objetos (POO) es un método para organizar programas que reúne muchas de las ideas presentadas en este capítulo. Al igual que los tipos de datos abstractos, los objetos crean una barrera de abstracción entre el uso y la implementación de los datos. Al igual que los diccionarios de envío en el paso de mensajes, los objetos responden a solicitudes de comportamiento. Al igual que las estructuras de datos mutables, los objetos tienen un estado local al que no se puede acceder directamente desde el entorno global. El sistema de objetos de Python proporciona una nueva sintaxis para facilitar la tarea de implementar todas estas técnicas útiles para organizar programas.

Pero el sistema de objetos ofrece más que sólo conveniencia; permite una nueva metáfora para diseñar programas en los que varios agentes independientes interactúan dentro de la computadora. Cada objeto agrupa el estado local y el comportamiento de una manera que oculta la complejidad de ambos detrás de una abstracción de datos. Nuestro ejemplo de un programa de restricciones comenzó a desarrollar esta metáfora al pasar mensajes entre restricciones y conectores. El sistema de objetos de Python extiende esta metáfora con nuevas formas de expresar cómo las diferentes partes de un programa se relacionan y se comunican entre sí. Los objetos no sólo pasan mensajes, sino que también comparten comportamiento entre otros objetos del mismo tipo y heredan características de tipos relacionados.

El paradigma de la programación orientada a objetos tiene su propio vocabulario que refuerza la metáfora del objeto. Hemos visto que un objeto es un valor de datos que tiene métodos y atributos, accesibles mediante la notación de puntos. Cada objeto también tiene un tipo, llamado *clase* Se pueden definir nuevas clases en Python, al igual que se pueden definir nuevas funciones.

2.5.1 Objetos y clases

Una clase sirve como plantilla para todos los objetos cuyo tipo es esa clase. Cada objeto es una instancia de una clase en particular. Todos los objetos que hemos utilizado hasta ahora tienen clases integradas, pero se pueden definir nuevas clases de manera similar a como se definen nuevas funciones. Una definición de clase especifica los atributos y métodos compartidos entre los objetos de esa clase. Presentaremos la declaración de clase revisando el ejemplo de una cuenta bancaria.

Al introducir el estado local, vimos que las cuentas bancarias se modelan naturalmente como valores mutables que tienen una`balance`Un objeto de cuenta bancaria debe tener un`withdraw`Método que actualiza el saldo de la cuenta y devuelve el importe solicitado, si está disponible. Nos gustaría que hubiera un comportamiento adicional para completar la abstracción de la cuenta: una cuenta bancaria debería poder devolver su saldo actual, devolver el nombre del titular de la cuenta y aceptar depósitos.

Un`Account`La clase nos permite crear múltiples instancias de cuentas bancarias. El acto de crear una nueva instancia de objeto se conoce como *instanciando* La sintaxis en Python para crear una instancia de una clase es idéntica a la sintaxis para llamar a una función. En este caso, llamamos`Account`Con el argumento`'Jim'`, el nombre del titular de la cuenta.

```
a = Account('Jim')
```

Un *atributo* de un objeto es un par nombre-valor asociado con el objeto, al que se puede acceder mediante la notación de puntos. Los atributos específicos de un objeto en particular, a diferencia de todos los objetos de una clase, se denominan *atributos de instancia.* Cada`Account`tiene su propio saldo y nombre del titular de la cuenta, que son ejemplos de atributos de instancia. En la comunidad de programación más amplia, los atributos de instancia también pueden llamarse *campos* , *propiedades* , o *variables de instancia.*

```
a.holder
```

```
'Jim'
```

```
a.balance
```

Las funciones que operan sobre el objeto o realizan cálculos específicos del objeto se denominan métodos. Los efectos secundarios y el valor de retorno de un método pueden depender de otros atributos del objeto y cambiarlos. Por ejemplo,`deposit`es un método nuestro`Account`objeto`a`. Toma un argumento, la cantidad a depositar, cambia el`balance`atributo del objeto y devuelve el saldo resultante.

```
a.deposit(15)
```

En programación orientada a objetos, decimos que los métodos son *invocado* sobre un objeto en particular. Como resultado de invocar el`withdraw`método, o

bien se aprueba el retiro y se deduce y devuelve el monto, o bien se rechaza la solicitud y la cuenta imprime un mensaje de error.

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
a.balance       # The balance attribute has changed
```

```
a.withdraw(10)
```

```
'Insufficient funds'
```

Como se ilustra arriba, el comportamiento de un método puede depender de los atributos cambiantes del objeto. Dos llamadas a`withdraw`con el mismo argumento devuelven resultados diferentes.

2.5.2 Definición de clases

Las clases definidas por el usuario son creadas por`class`declaraciones, que constan de una sola cláusula. Una declaración de clase define el nombre de la clase y una clase base (que se analiza en la sección sobre herencia) y luego incluye un conjunto de declaraciones para definir los atributos de la clase:

```
class <name>(<base class>):
    <suite>
```

Cuando se ejecuta una declaración de clase, se crea una nueva clase y se vincula a`<name>`en el primer marco del entorno actual. Luego se ejecuta la suite. Todos los nombres enlazados dentro del`<suite>`de un`class`declaración, a través de`def`o declaraciones de asignación, crean o modifican atributos de la clase.

Las clases se organizan típicamente en torno a la manipulación de atributos de instancia, que son los pares nombre-valor asociados no con la clase en sí, sino con cada objeto de esa clase. La clase especifica los atributos de instancia de sus objetos definiendo un método para inicializar nuevos objetos. Por ejemplo, parte de la inicialización de un objeto de la`Account`La clase consiste en asignarle un saldo inicial de 0.

El`<suite>`de un`class`La declaración contiene`def`declaraciones que definen nuevos métodos para objetos de esa clase. El método que inicializa objetos tiene un nombre especial en Python,`__init__`(dos guiones bajos a cada lado de "init"), y se llama *constructor* Para la clase.

```
class Account(object):
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

El`__init__`método para`Account`tiene dos parámetros formales. El primero,`self`, está ligado al recién creado`Account`objeto. El segundo parámetro,`account_holder`, está ligado al argumento que se pasa a la clase cuando se la llama para ser instanciada.

El constructor vincula el nombre del atributo de instancia`balance`a`0`También vincula el nombre del atributo.`holder`al valor del nombre`account_holder`. El parámetro formal`account_holder`es un nombre local para el`__init__`método. Por otra parte, el nombre`holder`que está vinculado a través de la declaración de asignación final persiste, porque se almacena como un atributo de`self`utilizando notación de puntos.

Una vez definido el`Account`clase, podemos instanciarla.

```
a = Account('Jim')
```

Este "llamado" a la`Account`La clase crea un nuevo objeto que es una instancia de`Account`, luego llama a la función constructora`__init__`con dos argumentos: el objeto recién creado y la cadena`'Jim'`Por convención, utilizamos el nombre del parámetro`self`para el primer argumento de un constructor, porque está vinculado al objeto que se instancia. Esta convención se adopta en prácticamente todo el código Python.

Ahora, podemos acceder al objeto.`balance`y`holder`utilizando notación de puntos.

```
a.holder
```

```
0
```

```
a.holder
```

```
'Jim'
```

Identidad. Cada nueva instancia de cuenta tiene su propio atributo de saldo, cuyo valor es independiente de otros objetos de la misma clase.

```
a.holder
```

```
[0, 200]
```

Para hacer cumplir esta separación, cada objeto que sea una instancia de una clase definida por el usuario tiene una identidad única. La identidad del objeto se compara utilizando`is`y`is not`operadores.

```
a.holder
```

```
True
```

```
a.holder
```

```
True
```

A pesar de estar construidos a partir de llamadas idénticas, los objetos vinculados a`a`y`b`no son lo mismo. Como es habitual, vincular un objeto a un nuevo nombre mediante una asignación no crea un nuevo objeto.

```
a.holder
```

```
True
```

Los objetos nuevos que tienen clases definidas por el usuario solo se crean cuando una clase (como`Account`) se instancia con la sintaxis de expresión de llamada.

Métodos. Los métodos de objeto también se definen mediante un`def`Declaración en la suite de un`class`Declaración. A continuación,`deposit`y`withdraw`Ambos se definen como métodos sobre objetos del`Account`clase.

Número de serie 16

Si bien las definiciones de métodos no difieren de las definiciones de funciones en la forma en que se declaran, las definiciones de métodos tienen un efecto diferente. El valor de función que se crea mediante un`def`declaración dentro de una`class`La declaración está vinculada al nombre declarado, pero está vinculada localmente dentro de la clase como un atributo. Ese valor se invoca como un método que utiliza la notación de punto desde una instancia de la clase.

Cada definición de método incluye nuevamente un primer parámetro especial`self`, que está vinculado al objeto en el que se invoca el método. Por ejemplo, digamos que`deposit`se invoca en un caso particular`Account`objeto y se le pasó un único valor de argumento: la cantidad depositada. El objeto en sí está vinculado a`self`, mientras que el argumento está destinado a`amount`Todos los métodos invocados tienen acceso al objeto a través de`self`parámetro, por lo que todos pueden acceder y manipular el estado del objeto.

Para invocar estos métodos, utilizamos nuevamente la notación de puntos, como se ilustra a continuación.

```
a.holder
```

```
100
```

Número de serie 18

```
10
```

```
a.holder
```

```
'Insufficient funds'
```

```
a.balance
```

```
'Tom'
```

Cuando se invoca un método mediante notación de puntos, el objeto en sí (vinculado a`tom_account`, en este caso) cumple una doble función. En primer lugar, determina el nombre`withdraw`medio;`withdraw`no es un nombre en el entorno, sino un nombre que es local para el`Account`Clase. En segundo lugar, está ligado al primer parámetro.`self`Cuando el`withdraw`Se invoca el método. Los detalles del procedimiento para evaluar la notación de puntos se describen en la siguiente sección.

2.5.3 Paso de mensajes y expresiones de puntos

Los métodos, que se definen en clases, y los atributos de instancia, que normalmente se asignan en constructores, son los elementos fundamentales de la programación orientada a objetos. Estos dos conceptos replican gran parte del comportamiento de un diccionario de envío en una implementación de paso de mensajes de un valor de datos. Los objetos reciben mensajes mediante notación de puntos, pero en lugar de que esos mensajes sean claves arbitrarias con valores de cadena, son nombres locales de una clase. Los objetos también tienen valores de estado locales nombrados (los atributos de instancia), pero se puede acceder a ese estado y manipularlo mediante notación de puntos, sin tener que emplear`nonlocal`Declaraciones en la implementación.

La idea central en el paso de mensajes era que los valores de los datos deberían tener un comportamiento que respondiera a los mensajes que son relevantes para el tipo abstracto que representan. La notación de puntos es una característica sintáctica de Python que formaliza la metáfora del paso de mensajes. La ventaja de usar un lenguaje con un sistema de objetos integrado es que el paso de mensajes puede interactuar sin problemas con otras características del lenguaje, como las declaraciones de asignación. No necesitamos diferentes mensajes para "obtener" o "establecer" el valor asociado con un nombre de atributo local; la sintaxis del lenguaje nos permite usar el nombre del mensaje directamente.

Expresiones de puntos. El fragmento de código`tom_account.deposit`se llama un *expresión de punto* Una expresión de punto consta de una expresión, un punto y un nombre:

```
a.balance
```

El`<expression>`puede ser cualquier expresión válida de Python, pero la`<name>`debe ser un nombre simple (no una expresión que evalúe un nombre). Una expresión de punto evalúa el valor del atributo con el valor dado.`<name>`, para el objeto que es el valor de la`<expression>`.

La función incorporada`getattr`También devuelve un atributo para un objeto por nombre. Es la función equivalente a la notación de puntos.`getattr`Podemos buscar un atributo usando una cadena, tal como lo hicimos con un diccionario de despacho.

```
a.balance
```

```
10
```

También podemos probar si un objeto tiene un atributo nombrado con`hasattr`.

```
a.balance
```

```
True
```

Los atributos de un objeto incluyen todos sus atributos de instancia, junto con todos los atributos (incluidos los métodos) definidos en su clase. Los métodos

son atributos de la clase que requieren un manejo especial.

Método y funciones. Cuando se invoca un método en un objeto, ese objeto se pasa implícitamente como el primer argumento del método. Es decir, el objeto que es el valor del`<expression>`a la izquierda del punto se pasa automáticamente como primer argumento al método nombrado en el lado derecho de la expresión del punto. Como resultado, el objeto está vinculado al parámetro`self`.

Para lograr la automatización`self`enlace, Python distingue entre *funciones* , que venimos creando desde el inicio del curso, y *métodos enlazados* , que acoplan una función y el objeto en el que se invocará ese método. Un valor de método enlazado ya está asociado con su primer argumento, la instancia en la que se invocó, que se llamará`self`cuando se llama al método.

Podemos ver la diferencia en el intérprete interactivo llamando`type`sobre los valores devueltos de expresiones de punto. Como atributo de una clase, un método es simplemente una función, pero como atributo de una instancia, es un método vinculado:

```
a.balance
```

```
a.balance
```

Estos dos resultados difieren solo en el hecho de que el primero es una función estándar de dos argumentos con parámetros.`self`y`amount`El segundo es un método de un solo argumento, donde el nombre`self`estará vinculado al objeto nombrado`tom_account`automáticamente cuando se llama al método, mientras que el parámetro`amount`se vinculará al argumento pasado al método. Ambos valores, ya sean valores de función o valores de método vinculados, están asociados con el mismo`deposit`cuerpo de la función.

Podemos llamar`deposit`de dos maneras: como función y como método ligado. En el primer caso, debemos proporcionar un argumento para la`self`parámetro explícitamente. En el último caso, el`self`El parámetro se vincula automáticamente.

```
a.balance
```

```
1011
```

```
a.balance
```

```
2011
```

La función`getattr`se comporta exactamente como la notación de puntos: si su primer argumento es un objeto pero el nombre es un método definido en la clase, entonces`getattr`devuelve un valor de método enlazado. Por otro lado, si el primer argumento es una clase, entonces`getattr`devuelve directamente el valor del atributo, que es una función simple.

Orientación práctica: convenciones de nomenclatura. Los nombres de clase se escriben convencionalmente utilizando la convención CapWords (tam-

bién llamada CamelCase porque las letras mayúsculas en el medio de un nombre son como jorobas). Los nombres de los métodos siguen la convención estándar de nombrar funciones utilizando palabras en minúscula separadas por guiones bajos.

En algunos casos, existen variables de instancia y métodos relacionados con el mantenimiento y la consistencia de un objeto que no queremos que los usuarios del objeto vean o utilicen. No son parte de la abstracción definida por una clase, sino de la implementación. La convención de Python dicta que si un nombre de atributo comienza con un guión bajo, solo se debe acceder a él dentro de los métodos de la clase en sí, en lugar de hacerlo por parte de los usuarios de la clase.

2.5.4 Atributos de clase

Algunos valores de atributos se comparten entre todos los objetos de una clase determinada. Dichos atributos están asociados con la clase en sí, en lugar de con cualquier instancia individual de la clase. Por ejemplo, supongamos que un banco paga intereses sobre el saldo de las cuentas a una tasa de interés fija. Esa tasa de interés puede cambiar, pero es un valor único compartido entre todas las cuentas.

Los atributos de clase se crean mediante declaraciones de asignación en el conjunto de una`class`declaración, fuera de cualquier definición de método. En la comunidad de desarrolladores más amplia, los atributos de clase también pueden denominarse variables de clase o variables estáticas. La siguiente declaración de clase crea un atributo de clase para`Account`con el nombre`interest`.

```
a.balance
```

Todavía se puede acceder a este atributo desde cualquier instancia de la clase.

```
a.balance
```

```
0.02
```

```
a.deposit(15)
```

```
0.02
```

Sin embargo, una sola declaración de asignación a un atributo de clase cambia el valor del atributo para todas las instancias de la clase.

```
a.deposit(15)
```

```
0.04
```

```
a.deposit(15)
```

```
0.04
```

Nombres de atributos. Hemos introducido suficiente complejidad en nuestro sistema de objetos como para que tengamos que especificar cómo se resuelven los

nombres en atributos particulares. Después de todo, podríamos tener fácilmente un atributo de clase y un atributo de instancia con el mismo nombre.

Como hemos visto, una expresión de punto consta de una expresión, un punto y un nombre:

```
a.deposit(15)
```

Para evaluar una expresión de puntos:

1. Evaluar la`<expression>`a la izquierda del punto, lo que da como resultado *objeto* de la expresión del punto.
2. `<name>`se compara con los atributos de instancia de ese objeto; si existe un atributo con ese nombre, se devuelve su valor.
3. Si`<name>`no aparece entre los atributos de instancia, entonces`<name>`se busca en la clase, lo que produce un valor de atributo de clase.
4. Ese valor se devuelve a menos que sea una función, en cuyo caso se devuelve un método enlazado.

En este procedimiento de evaluación, los atributos de instancia se encuentran antes que los atributos de clase, de la misma manera que los nombres locales tienen prioridad sobre los globales en un entorno. Los métodos definidos dentro de la clase se vinculan al objeto de la expresión de punto durante el tercer paso de este procedimiento de evaluación. El procedimiento para buscar un nombre en una clase tiene matices adicionales que surgirán en breve, una vez que introduzcamos la herencia de clases.

Asignación. Todas las instrucciones de asignación que contienen una expresión de punto en su lado izquierdo afectan a los atributos del objeto de esa expresión de punto. Si el objeto es una instancia, la asignación establece un atributo de instancia. Si el objeto es una clase, la asignación establece un atributo de clase. Como consecuencia de esta regla, la asignación a un atributo de un objeto no puede afectar a los atributos de su clase. Los ejemplos siguientes ilustran esta distinción.

Si asignamos al atributo nombrado`interest`de una instancia de cuenta, creamos un nuevo atributo de instancia que tiene el mismo nombre que el atributo de clase existente.

```
a.deposit(15)
```

y ese valor de atributo será devuelto desde una expresión de punto.

```
a.deposit(15)
```

```
0.08
```

Sin embargo, el atributo de clase`interest`aún conserva su valor original, que se devuelve para todas las demás cuentas.

```
a.deposit(15)
```

```
0.04
```

Cambios en el atributo de clase`interest`afectará`tom_account`, pero el atributo de instancia para`jim_account`no se verá afectado

```
a.deposit(15)
```

```
0.05
```

```
a.deposit(15)
```

```
0.08
```

2.5.5 Herencia

Cuando trabajamos en el paradigma de programación orientada a objetos, a menudo descubrimos que diferentes tipos de datos abstractos están relacionados. En particular, descubrimos que clases similares difieren en su grado de especialización. Dos clases pueden tener atributos similares, pero una representa un caso especial de la otra.

Por ejemplo, podríamos querer implementar una cuenta corriente, que es diferente de una cuenta estándar. Una cuenta corriente cobra $1 adicional por cada retiro y tiene una tasa de interés más baja. Aquí, demostramos el comportamiento deseado.

```
a.deposit(15)
```

```
0.01
```

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
20
```

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
14
```

A`CheckingAccount`es una especialización de un`Account`En la terminología de OOP, la cuenta genérica servirá como la clase base de`CheckingAccount`, mientras`CheckingAccount`será una subclase de`Account`. (Los términos *clase padre* y *superclase* También se utilizan para la clase base, mientras que *Clase infantil* También se utiliza para la subclase).

Una subclase *hereda* los atributos de su clase base, pero puede *anular* ciertos atributos, incluidos ciertos métodos. Con la herencia, solo especificamos qué es diferente entre la subclase y la clase base. Todo lo que no especificamos en la subclase se asume automáticamente que se comportará como lo haría en la clase base.

La herencia también tiene un papel en nuestra metáfora de objeto, además de ser una característica organizativa útil. La herencia está destinada a representar *es-un* relaciones entre clases, que contrastan con *tiene-un* relaciones. Una cuenta corriente *es-un* tipo específico de cuenta, por lo que tener una`CheckingAccount`heredar de`Account`es un uso adecuado de la herencia.

Por otro lado, un banco *tiene-un* lista de cuentas bancarias que administra, por lo que ninguna debe heredar de la otra. En cambio, una lista de objetos de cuenta se expresaría naturalmente como un atributo de instancia de un objeto bancario.

2.5.6 Uso de la herencia

Especificamos la herencia colocando la clase base entre paréntesis después del nombre de la clase. Primero, damos una implementación completa de la`Account`clase, que incluye cadenas de documentación para la clase y sus métodos.

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

Una implementación completa de`CheckingAccount`aparece a continuación.

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

Aquí, introducimos un atributo de clase`withdraw_charge`que es específico de la`CheckingAccount`clase. Asignamos un valor más bajo a la`interest`atributo. También definimos un nuevo`withdraw`método para anular el comportamiento definido en el`Account`clase. Sin más declaraciones en el conjunto de clases, todo el resto del comportamiento se hereda de la clase base.`Account`.

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
10
```

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
4
```

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
0.01
```

La expresión`checking.deposit`evalúa un método enlazado para realizar depósitos, que se definió en el`Account`clase. Cuando Python resuelve un nombre en una expresión de punto que no es un atributo de la instancia, busca el nombre en la clase. De hecho, el acto de "buscar" un nombre en una clase intenta encontrar ese nombre en cada clase base en la cadena de herencia para la clase del objeto original. Podemos definir este procedimiento de forma recursiva. Para buscar un nombre en una clase.

1. Si nombra un atributo en la clase, devuelve el valor del atributo.
2. De lo contrario, busque el nombre en la clase base, si hay una.

En el caso de`deposit`Python habría buscado el nombre primero en la instancia y luego en la`CheckingAccount`clase. Finalmente, se vería en el`Account`clase, donde`deposit`está definido. De acuerdo con nuestra regla de evaluación para expresiones de punto, dado que`deposit`es una función buscada en la clase para el`checking`Por ejemplo, la expresión de punto se evalúa como un valor de

método enlazado. Ese método se invoca con el argumento`10`, que llama al método de depósito con`self`Atado a la`checking`objeto y`amount`obligado a`10`.

La clase de un objeto permanece constante a lo largo de todo el tiempo. Aunque`deposit`El método se encontró en el`Account`clase,`deposit`se llama con`self`vinculado a una instancia de`CheckingAccount`, no de`Account`.

Llamando a los antepasados. Los atributos que se han anulado aún son accesibles a través de objetos de clase. Por ejemplo, implementamos el`withdraw`método de`CheckingAccount`llamando al`withdraw`método de`Account`con un argumento que incluía la`withdraw_charge`.

Tenga en cuenta que llamamos`self.withdraw_charge`en lugar del equivalente`CheckingAccount.withdraw_charge`El beneficio del primero sobre el segundo es que una clase que hereda de`CheckingAccount`podría anular el cargo por retiro. Si ese es el caso, nos gustaría que nuestra implementación`withdraw`para encontrar ese nuevo valor en lugar del antiguo.

2.5.7 Herencia múltiple

Python admite el concepto de una subclase que hereda atributos de múltiples clases base, una característica del lenguaje llamada *herencia múltiple*.

Supongamos que tenemos una`SavingsAccount`que hereda de`Account`, pero cobra a los clientes una pequeña tarifa cada vez que realizan un depósito.

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

Entonces, un ejecutivo inteligente concibe una`AsSeenOnTVAccount`Cuenta con las mejores características de ambos`CheckingAccount`y`SavingsAccount`: comisiones por retiro, comisiones por depósito y una tasa de interés baja. ¡Es una cuenta corriente y de ahorros al mismo tiempo! "Si la construimos", razona el ejecutivo, "alguien se registrará y pagará todas esas comisiones. Incluso les daremos un dólar".

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

De hecho, esta implementación está completa. Tanto los retiros como los depósitos generarán comisiones, utilizando las definiciones de funciones en`CheckingAccount`y`SavingsAccount`respectivamente.

```
a.withdraw(10)  # The withdraw method returns the balance after withdrawal
```

```
1
```

```
a.balance       # The balance attribute has changed
```

```
19
```

```
a.balance       # The balance attribute has changed
```

```
13
```

Las referencias no ambiguas se resuelven correctamente como se esperaba:

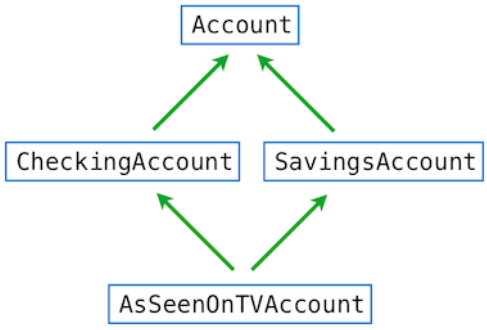
```
a.balance       # The balance attribute has changed
```

```
2
```

```
a.balance       # The balance attribute has changed
```

```
1
```

Pero ¿qué ocurre cuando la referencia es ambigua, como la referencia a la`withdraw`método que se define en ambos`Account`y`CheckingAccount`? La figura siguiente muestra una *gráfico de herencia* Para el`AsSeenOnTVAccount`clase. Cada flecha apunta desde una subclase a una clase base.



Para una forma de "diamante" simple como esta, Python resuelve los nombres de izquierda a derecha y luego hacia arriba. En este ejemplo, Python busca un nombre de atributo en las siguientes clases, en orden, hasta que encuentra un atributo con ese nombre:

```
a.balance       # The balance attribute has changed
```

No existe una solución correcta para el problema del ordenamiento de la herencia, ya que hay casos en los que podríamos preferir dar prioridad a ciertas clases heredadas sobre otras. Sin embargo, cualquier lenguaje de programación que admita la herencia múltiple debe seleccionar algún ordenamiento de manera consistente, de modo que los usuarios del lenguaje puedan predecir el comportamiento de sus programas.

Lectura adicional. Python resuelve este nombre mediante un algoritmo recursivo llamado Orden de resolución de métodos C3. El orden de resolución de métodos de cualquier clase se puede consultar mediante el algoritmo`mro`método en todas las clases.

```
a.balance       # The balance attribute has changed
```

```
['AsSeenOnTVAccount', 'CheckingAccount', 'SavingsAccount', 'Account', 'object']
```

El algoritmo preciso para encontrar ordenamientos de resolución de métodos no es un tema de este curso, pero esdescrito por el autor principal de Pythoncon

referencia al artículo original.

2.5.8 El papel de los objetos

El sistema de objetos de Python está diseñado para que la abstracción de datos y el paso de mensajes sean cómodos y flexibles. La sintaxis especializada de clases, métodos, herencia y expresiones de punto nos permiten formalizar la metáfora de objeto en nuestros programas, lo que mejora nuestra capacidad para organizar programas grandes.

En particular, nos gustaría que nuestro sistema de objetos promoviera una *separación de preocupaciones* entre los diferentes aspectos del programa. Cada objeto de un programa encapsula y administra alguna parte del estado del programa, y cada declaración de clase define las funciones que implementan alguna parte de la lógica general del programa. Las barreras de abstracción refuerzan los límites entre los diferentes aspectos de un programa grande.

La programación orientada a objetos es especialmente adecuada para programas que modelan sistemas que tienen partes separadas pero que interactúan entre sí. Por ejemplo, diferentes usuarios interactúan en una red social, diferentes personajes interactúan en un juego y diferentes formas interactúan en una simulación física. Al representar dichos sistemas, los objetos de un programa a menudo se asignan de manera natural a los objetos del sistema que se está modelando y las clases representan sus tipos y relaciones.

Por otra parte, las clases pueden no proporcionar el mejor mecanismo para implementar ciertas abstracciones. Las abstracciones funcionales proporcionan una metáfora más natural para representar relaciones entre entradas y salidas. Uno no debería sentirse obligado a encajar cada bit de lógica en un programa dentro de una clase, especialmente cuando definir funciones independientes para manipular datos es más natural. Las funciones también pueden imponer una separación de preocupaciones.

Los lenguajes multiparadigma como Python permiten a los programadores adaptar los paradigmas organizacionales a los problemas adecuados. Aprender a identificar cuándo introducir una nueva clase, en lugar de una nueva función, para simplificar o modularizar un programa es una habilidad de diseño importante en ingeniería de software que merece especial atención.

Contents

2.6 Implementación de clases y objetos **1**
2.6.1 Instancias 1
2.6.2 Clases 3
2.6.3 Uso de objetos implementados 4

\tableofcontents

2.6 Implementación de clases y objetos

Cuando trabajamos en el paradigma de programación orientada a objetos, utilizamos la metáfora del objeto para guiar la organización de nuestros programas. La mayor parte de la lógica sobre cómo representar y manipular datos se expresa en las declaraciones de clase. En esta sección, vemos que las clases y los objetos pueden representarse mediante funciones y diccionarios. El propósito de implementar un sistema de objetos de esta manera es ilustrar que el uso de la metáfora del objeto no requiere un lenguaje de programación especial. Los programas pueden estar orientados a objetos, incluso en lenguajes de programación que no tienen un sistema de objetos incorporado.

Para implementar objetos, abandonaremos la notación de puntos (que requiere soporte de lenguaje integrado), pero crearemos diccionarios de envío que se comporten de manera muy similar a los elementos del sistema de objetos integrado. Ya hemos visto cómo implementar el comportamiento de paso de mensajes a través de diccionarios de envío. Para implementar un sistema de objetos en su totalidad, enviamos mensajes entre instancias, clases y clases base, todas las cuales son diccionarios que contienen atributos.

No implementaremos todo el sistema de objetos de Python, que incluye características que no hemos cubierto en este texto (por ejemplo, metaclases y métodos estáticos). Nos centraremos en cambio en clases definidas por el usuario sin herencia múltiple y sin comportamiento introspectivo (como devolver la clase de una instancia). Nuestra implementación no está pensada para seguir la especificación precisa del sistema de tipos de Python. En cambio, está diseñada para implementar la funcionalidad principal que permite la metáfora del objeto.

2.6.1 Instancias

Comenzamos con las instancias. Una instancia tiene atributos con nombre, como el saldo de una cuenta, que se pueden configurar y recuperar. Implementamos una instancia utilizando un diccionario de envío que responde a los mensajes que "obtienen" y "configuran" valores de atributos. Los atributos en sí se almacenan en un diccionario local llamado`attributes`.

Como hemos visto anteriormente en este capítulo, los diccionarios en sí mismos son tipos de datos abstractos. Implementamos diccionarios con listas, implemen-

tamos listas con pares e implementamos pares con funciones. Al implementar un sistema de objetos en términos de diccionarios, tenga en cuenta que también podríamos implementar objetos utilizando únicamente funciones.

Para comenzar nuestra implementación, asumimos que tenemos una implementación de clase que puede buscar cualquier nombre que no sea parte de la instancia. Pasamos una clase a`make_instance`como parámetro`cls`.

```
def make_instance(cls):
    """Return a new object instance, which is a dispatch dictionary."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        else:
            value = cls['get'](name)
            return bind_method(value, instance)
    def set_value(name, value):
        attributes[name] = value
    attributes = {}
    instance = {'get': get_value, 'set': set_value}
    return instance
```

El`instance`es un diccionario de despacho que responde a los mensajes`get`y`set`. El`set`El mensaje corresponde a la asignación de atributos en el sistema de objetos de Python: todos los atributos asignados se almacenan directamente dentro del diccionario de atributos local del objeto.`get`, si`name`No aparece en el local`attributes`diccionario, luego se busca en la clase. Si el`value`devuelto por`cls`es una función, debe estar vinculada a la instancia.

Valores de método enlazados. El`get_value`función en`make_instance`encuentra un atributo nombrado en su clase con`get`, luego llama`bind_method`La vinculación de un método solo se aplica a valores de función y crea un valor de método vinculado a partir de un valor de función insertando la instancia como primer argumento:

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

Cuando se llama a un método, el primer parámetro`self`Estará ligado al valor de`instance`por esta definición.

2.6.2 Clases

Una clase también es un objeto, tanto en el sistema de objetos de Python como en el sistema que estamos implementando aquí. Para simplificar, decimos que las clases en sí mismas no tienen una clase. (En Python, las clases sí tienen clases; casi todas las clases comparten la misma clase, llamada`type`.) Una clase puede responder a`get`y`set`mensajes, así como la`new`mensaje:

```
def make_class(attributes, base_class=None):
    """Return a new class, which is a dispatch dictionary."""
    def get_value(name):
        if name in attributes:
            return attributes[name]
        elif base_class is not None:
            return base_class['get'](name)
    def set_value(name, value):
        attributes[name] = value
    def new(*args):
        return init_instance(cls, *args)
    cls = {'get': get_value, 'set': set_value, 'new': new}
    return cls
```

A diferencia de una instancia, la`get`La función para clases no consulta su clase cuando no se encuentra un atributo, sino que consulta su`base_class`No se requiere vinculación de métodos para las clases.

Inicialización. El`new`función en`make_class`llamadas`init_instance`, que primero crea una nueva instancia y luego invoca un método llamado`__init__`.

```
def init_instance(cls, *args):
    """Return a new object with type cls, initialized with args."""
    instance = make_instance(cls)
    init = cls['get']('__init__')
    if init:
        init(instance, *args)
    return instance
```

Esta función final completa nuestro sistema de objetos. Ahora tenemos instancias, que`set`localmente, pero vuelven a sus clases en`get`Después de que una instancia busca un nombre en su clase, se vincula a los valores de la función para crear métodos. Finalmente, las clases pueden crear`new`instancias, y aplican sus`__init__`función constructora inmediatamente después de la creación de la instancia.

En este sistema de objetos, la única función que debe ser llamada por el usuario es`create_class`Todas las demás funciones se habilitan mediante el paso de mensajes. De manera similar, el sistema de objetos de Python se invoca mediante`class`La declaración y todas sus demás funciones se habilitan a través de expresiones de puntos y llamadas a clases.

2.6.3 Uso de objetos implementados

Ahora volvemos a utilizar el ejemplo de la cuenta bancaria de la sección anterior. Utilizando nuestro sistema de objetos implementado, crearemos una`Account`clase, una`CheckingAccount`subclase y una instancia de cada una.

El`Account`La clase se crea a través de una`create_account_class`función, que tiene una estructura similar a una`class`declaración en Python, pero concluye con una llamada a`make_class`.

```
def make_account_class():
    """Return the Account class, which has deposit and withdraw methods."""
    def __init__(self, account_holder):
        self['set']('holder', account_holder)
        self['set']('balance', 0)
    def deposit(self, amount):
        """Increase the account balance by amount and return the new balance."""
        new_balance = self['get']('balance') + amount
        self['set']('balance', new_balance)
        return self['get']('balance')
    def withdraw(self, amount):
        """Decrease the account balance by amount and return the new balance."""
        balance = self['get']('balance')
        if amount > balance:
            return 'Insufficient funds'
        self['set']('balance', balance - amount)
        return self['get']('balance')
    return make_class({'__init__': __init__,
                       'deposit':  deposit,
                       'withdraw': withdraw,
                       'interest': 0.02})
```

En esta función, los nombres de los atributos se establecen al final. A diferencia de Python`class`declaraciones que imponen coherencia entre los nombres de funciones intrínsecas y los nombres de atributos, aquí debemos especificar manualmente la correspondencia entre los nombres de atributos y los valores.

El`Account`La clase finalmente se instancia mediante asignación.

```
Account = make_account_class()
```

Luego, se crea una instancia de cuenta a través de`new`mensaje, que requiere un nombre para acompañar la cuenta recién creada.

```
jim_acct = Account['new']('Jim')
```

Entonces,`get`mensajes pasados a`jim_acct`recuperar propiedades y métodos. Se pueden llamar métodos para actualizar el saldo de la cuenta.

```
jim_acct['get']('holder')
```

```
'Jim'
```

```
jim_acct['get']('interest')
```

```
jim_acct['get']('deposit')(20)
```

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

```
15
```

Al igual que con el sistema de objetos de Python, establecer un atributo de una instancia no cambia el atributo correspondiente de su clase.

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

```
0.02
```

Herencia. Podemos crear una subclase`CheckingAccount`sobrecargando un subconjunto de los atributos de la clase. En este caso, cambiamos el`withdraw`método para imponer una tarifa y reducimos la tasa de interés.

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

En esta implementación, llamamos al`withdraw`función de la clase base`Account`desde`withdraw`función de la subclase, como lo haríamos en el sistema de objetos integrado de Python.
Podemos crear la subclase en sí y una instancia, como antes.

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
```

```
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

Los depósitos se comportan de manera idéntica, al igual que la función constructora. Los retiros imponen una tarifa de $1 del especialista.`withdraw`método, y`interest`tiene el nuevo valor más bajo de`CheckingAccount`.

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

```
0.01
```

```
def bind_method(value, instance):
    """Return a bound method if value is callable, or value otherwise."""
    if callable(value):
        def method(*args):
            return value(instance, *args)
        return method
    else:
        return value
```

```
20
```

Número de serie 16

```
14
```

Nuestro sistema de objetos basado en diccionarios es bastante similar en su implementación al sistema de objetos integrado en Python. En Python, una instancia de cualquier clase definida por el usuario tiene un atributo especial`__dict__`que almacena los atributos de instancia local para ese objeto en un diccionario, de forma muy similar a nuestro`attributes`Python se diferencia porque distingue ciertos métodos especiales que interactúan con funciones integradas para garantizar que esas funciones se comporten correctamente para argumentos de muchos tipos diferentes. Las funciones que operan con diferentes tipos son el tema de la siguiente sección.

Contents

3.1 Introducción **1**
3.1.1 Lenguajes de programación 1

\tableofcontents

3.1 Introducción

Los capítulos 1 y 2 describen la estrecha relación que existe entre dos elementos fundamentales de la programación: las funciones y los datos. Vimos cómo se pueden manipular las funciones como datos mediante funciones de orden superior. También vimos cómo se puede dotar de comportamiento a los datos mediante el paso de mensajes y un sistema de objetos. También hemos estudiado técnicas para organizar programas grandes, como la abstracción funcional, la abstracción de datos, la herencia de clases y las funciones genéricas. Estos conceptos básicos constituyen una base sólida sobre la que construir programas modulares, mantenibles y extensibles.

Este capítulo se centra en el tercer elemento fundamental de la programación: los programas en sí. Un programa Python es simplemente una colección de texto. Solo a través del proceso de interpretación podemos realizar algún cálculo significativo basado en ese texto. Un lenguaje de programación como Python es útil porque podemos definir un *intérprete* , un programa que lleva a cabo los procedimientos de evaluación y ejecución de Python. No es exagerado considerar que esta es la idea más fundamental en programación: que un intérprete, que determina el significado de las expresiones en un lenguaje de programación, es simplemente otro programa.

Para entender este punto es necesario cambiar la imagen que tenemos de nosotros mismos como programadores. Empezamos a vernos como diseñadores de lenguajes, en lugar de ser meros usuarios de lenguajes diseñados por otros.

3.1.1 Lenguajes de programación

De hecho, podemos considerar muchos programas como intérpretes de algún lenguaje. Por ejemplo, el propagador de restricciones del capítulo anterior tiene sus propias primitivas y medios de combinación. El lenguaje de restricciones era bastante especializado: proporcionaba un método declarativo para describir una determinada clase de relaciones matemáticas, no un lenguaje totalmente general para describir cálculos. Si bien ya hemos estado diseñando lenguajes de algún tipo, el material de este capítulo ampliará en gran medida la gama de lenguajes que podemos interpretar.

Los lenguajes de programación varían ampliamente en sus estructuras sintácticas, características y dominio de aplicación. Entre los lenguajes de programación de propósito general, los constructos de definición de funciones y aplicación de

funciones son omnipresentes. Por otro lado, existen lenguajes potentes que no incluyen un sistema de objetos, funciones de orden superior o incluso constructos de control como`while`y`for`Para ilustrar cuán diferentes pueden ser los idiomas, presentaremosLogocomo ejemplo de un lenguaje de programación potente y expresivo que incluye muy pocas características avanzadas.

En este capítulo, estudiamos el diseño de intérpretes y los procesos computacionales que crean al ejecutar programas. La perspectiva de diseñar un intérprete para un lenguaje de programación general puede parecer desalentadora. Después de todo, los intérpretes son programas que pueden realizar cualquier cálculo posible, dependiendo de su entrada. Sin embargo, los intérpretes típicos tienen una elegante estructura común: dos funciones recursivas entre sí. La primera evalúa expresiones en entornos; la segunda aplica funciones a argumentos.

Estas funciones son *recursivo* En el sentido de que se definen entre sí: la aplicación de una función requiere evaluar las expresiones de su cuerpo, mientras que la evaluación de una expresión puede implicar la aplicación de una o más funciones. Las dos secciones siguientes de este capítulo se centran en las funciones recursivas y las estructuras de datos, que resultarán esenciales para comprender el diseño de un intérprete. El final del capítulo se centra en dos nuevos lenguajes y en la tarea de implementar intérpretes para ellos.

Contents

3.2 Funciones y procesos que generan **1**
3.2.1 Funciones recursivas 1
3.2.2 La anatomía de las funciones recursivas 3
3.2.3 Recursión de árboles 6
3.2.4 Ejemplo: Conteo de cambio 9
3.2.5 Órdenes de crecimiento 11

\tableofcontents

3.2 Funciones y procesos que generan

Una función es un patrón para la *evolución local* de un proceso computacional. Especifica cómo cada etapa del proceso se construye a partir de la etapa anterior. Nos gustaría poder hacer afirmaciones sobre el comportamiento general de un proceso cuya evolución local ha sido especificada por una o más funciones. Este análisis es muy difícil de hacer en general, pero al menos podemos intentar describir algunos patrones típicos de evolución de procesos.

En esta sección examinaremos algunas "formas" comunes de los procesos generados por funciones simples. También investigaremos las tasas a las que estos procesos consumen los importantes recursos computacionales del tiempo y el espacio.

3.2.1 Funciones recursivas

Una función se llama *recursivo* Si el cuerpo de esa función llama a la función misma, ya sea directa o indirectamente. Es decir, el proceso de ejecución del cuerpo de una función recursiva puede requerir a su vez aplicar esa función nuevamente. Las funciones recursivas no requieren ninguna sintaxis especial en Python, pero sí requieren cierto cuidado para definirlas correctamente.

Como introducción a las funciones recursivas, comenzamos con la tarea de convertir una palabra inglesa en su equivalente en pig latin. El pig latin es un lenguaje secreto: uno que aplica una transformación simple y determinista a cada palabra que oculta el significado de la palabra. Thomas Jefferson supuestamente fue unadoptante tempranoEl equivalente en latín pig de una palabra inglesa desplaza el grupo consonántico inicial (que puede estar vacío) del principio de la palabra al final y lo sigue con la vocal "-ay". Por lo tanto, la palabra "pun" se convierte en "unpay", "stout" en "outstay" y "all" en "allay".

```
def pig_latin(w):
    """Return the Pig Latin equivalent of English word w."""
    if starts_with_a_vowel(w):
        return w + 'ay'
    return pig_latin(w[1:] + w[0])
```

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

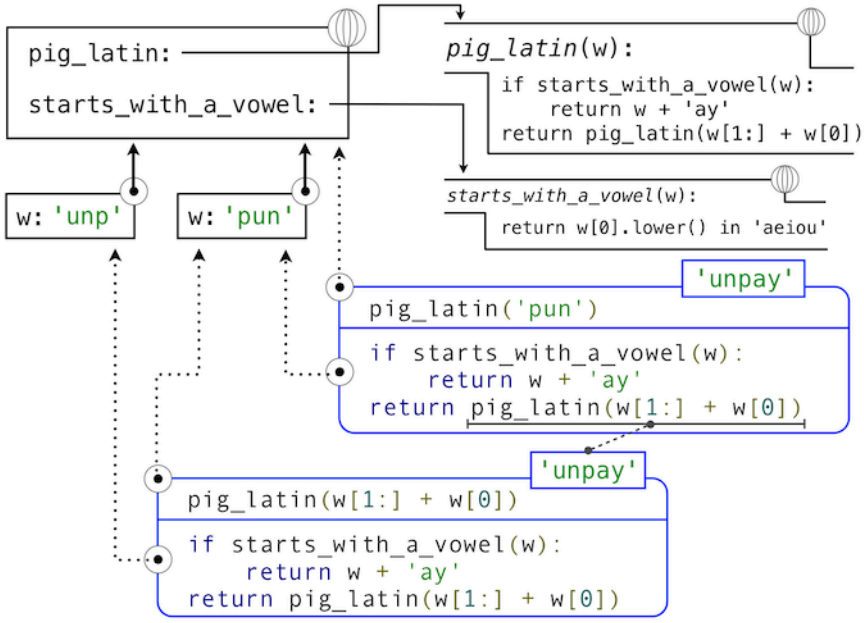
La idea detrás de esta definición es que la variante en pig latin de una cadena que comienza con una consonante es la misma que la variante en pig latin de otra cadena: la que se crea al mover la primera letra al final. Por lo tanto, la palabra pig latin para "enviar" es la misma que para "finales" (*finalizando*), y la palabra latina pig para "asfixiar" es la misma que la palabra latina pig para "madres" (*otrospueden*). Además, mover una consonante del principio de la palabra al final da como resultado un problema más simple con menos consonantes iniciales. En el caso de "sending", mover la "s" al final da como resultado una palabra que comienza con una vocal, y así nuestro trabajo está hecho.

Esta definición de`pig_latin`es a la vez completa y correcta, aunque la`pig_latin`La función se llama dentro de su propio cuerpo.

```
pig_latin('pun')
```

```
'unpay'
```

La idea de poder definir una función en términos de sí misma puede resultar inquietante; puede parecer poco claro cómo una definición tan "circular" podría tener sentido, y mucho menos especificar un proceso bien definido que debe ser llevado a cabo por una computadora. Sin embargo, podemos entender con precisión cómo esta función recursiva se aplica con éxito utilizando nuestro modelo de entorno de computación. El diagrama de entorno y el árbol de expresión que representan la evaluación de`pig_latin('pun')`aparecen a continuación.



Los pasos de los procedimientos de evaluación de Python que producen este resultado son:

1. El`def`Declaración para`pig_latin`se ejecuta, lo cual

2. `1. Crea un nuevo _cerdo_latino_ objeto de función con el cuerpo indicado, y`
   `2. Vincula el nombre`pig_latin`a esa función en el marco actual (global)`

2. El`def`Declaración para`starts_with_a_vowel`se ejecuta de manera similar 3. La expresión de llamada`pig_latin('pun')`se evalúa mediante 4. 1. Evaluación de las subexpresiones de operador y operando mediante 2. 1. Búsqueda del nombre`pig_latin`que está ligado a la *cerdo_latino* Función 2. Evaluación del literal de cadena del operando en el objeto de cadena`'pun'`

```
 
2.  Applying the function *pig\_latin* to the argument `'pun'` by


 
1.  Adding a local frame that extends the global frame
2.  Binding the formal parameter `w` to the argument `'pun'` in that frame
3.  Executing the body of *pig\_latin* in the environment that starts with that frame:
4.  1.  The initial conditional statement has no effect, because the header expression evalu
    2.  The final return expression `pig_latin(w[1:] + w[0])` is evaluated by
    3.  1.  Looking up the name `pig_latin` that is bound to the *pig\_latin* function
        2.  Evaluating the operand expression to the string object `'unp'`
        3.  Applying *pig\_latin* to the argument `'unp'`, which returns the desired result
```

Como ilustra este ejemplo, una función recursiva se aplica correctamente, a pesar de su carácter circular._cerdo_latino_ La función se aplica dos veces, pero con un argumento diferente cada vez. Aunque la segunda llamada proviene del cuerpo de *cerdo_latino* En sí, buscar esa función por nombre tiene éxito porque el nombre`pig_latin`está ligado al entorno antes de que su cuerpo sea ejecutado.

Este ejemplo también ilustra cómo el procedimiento de evaluación recursiva de Python puede interactuar con una función recursiva para desarrollar un proceso computacional complejo con muchos pasos anidados, aunque la definición de la función pueda contener muy pocas líneas de código.

3.2.2 La anatomía de las funciones recursivas

Se puede encontrar un patrón común en el cuerpo de muchas funciones recursivas. El cuerpo comienza con una *caso base* , una declaración condicional que define el comportamiento de la función para las entradas que son más simples de procesar. En el caso de`pig_latin`, el caso base se produce para cualquier argumento que comience con una vocal. En este caso, no queda trabajo por hacer, solo se devuelve el argumento con “ay” agregado al final. Algunas funciones recursivas tendrán múltiples casos base.

Los casos base son luego seguidos por uno o más *llamadas recursivas* Las lla-

madas recursivas requieren un carácter determinado: deben simplificar el problema original. En el caso de`pig_latin`, cuantas más consonantes iniciales haya`w`, cuanto más trabajo queda por hacer, en la llamada recursiva,`pig_latin(w[1:] + w[0])`, nosotros llamamos`pig_latin`En una palabra que tiene una consonante inicial menos, un problema más simple. Cada llamada sucesiva a`pig_latin`Será aún más sencillo hasta llegar al caso base: una palabra sin consonantes iniciales.

Las funciones recursivas expresan el cálculo simplificando los problemas de forma incremental. A menudo, actúan sobre los problemas de una manera diferente a los enfoques iterativos que hemos utilizado en el pasado. Considere una función`fact`Para calcular`n`factorial, donde por ejemplo`fact(4)`calcula $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Una implementación natural utilizando un`while`La declaración acumula el total multiplicando cada entero positivo hasta`n`.

```
def fact_iter(n):
    total, k = 1, 1
    while k <= n:
        total, k = total * k, k + 1
    return total

fact_iter(4)
```

Por otro lado, una implementación recursiva de factorial puede expresar`fact(n)`en términos de`fact(n-1)`, un problema más simple. El caso base de la recursión es la forma más simple del problema:`fact(1)`es`1`.

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n-1)

fact(4)
```

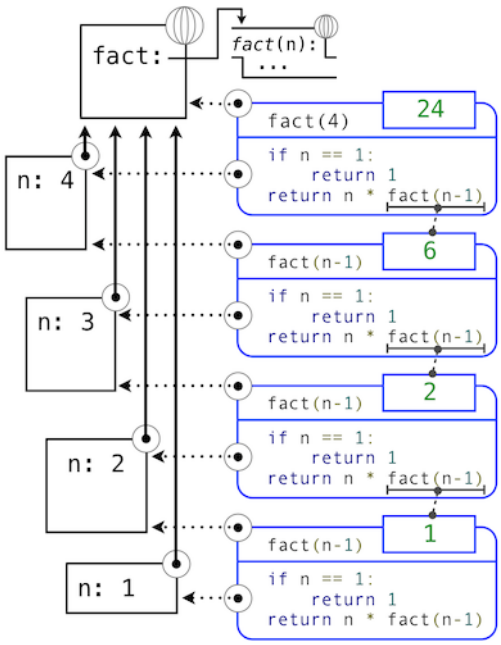
La exactitud de esta función es fácil de verificar a partir de la definición estándar de la función matemática factorial:

```
$$\begin{array}{l l} (n-1)! &= (n-1) \cdot (n-2) \cdot \dots \cdot 1 \\ n! &= n \cdot (n-1)
```

Estas dos funciones factoriales difieren conceptualmente. La función iterativa construye el resultado a partir del caso base de 1 hasta el total final multiplicando sucesivamente cada término. La función recursiva, por otro lado, construye el resultado directamente a partir del término final.`n`, y el resultado del problema más simple,`fact(n-1)`.

A medida que la recursión se "desenrolla" a través de aplicaciones sucesivas de la *hecho* función a instancias de problemas cada vez más simples, el resultado se construye eventualmente a partir del caso base. El diagrama a continuación muestra cómo termina la recursión al pasar el argumento`1`a`fact`, y cómo el

resultado de cada llamada depende de la siguiente hasta que se alcanza el caso base.



Si bien podemos desenredar la recursión utilizando nuestro modelo de computación, a menudo es más claro pensar en las llamadas recursivas como abstracciones funcionales. Es decir, no deberíamos preocuparnos por cómo`fact(n-1)`se implementa en el cuerpo de`fact`;deberíamos simplemente confiar en que calcula el factorial de`n-1`Tratar una llamada recursiva como una abstracción funcional se ha denominado *Salto de fe recursivo* Definimos una función en términos de sí misma, pero simplemente confiamos en que los casos más simples funcionarán correctamente al verificar la corrección de la función. En este ejemplo, confiamos en que`fact(n-1)`se calculará correctamente`(n-1)!`;solo debemos comprobar que`n!`se calcula correctamente si se cumple esta suposición. De esta manera, verificar la corrección de una función recursiva es una forma de prueba por inducción.

Las funciones *hecho_iter* y *hecho* También difieren porque el primero debe introducir dos nombres adicionales,`total`y`k`, que no son necesarios en la implementación recursiva. En general, las funciones iterativas deben mantener algún estado local que cambia a lo largo del curso del cálculo. En cualquier punto de la iteración, ese estado caracteriza el resultado del trabajo completado y la cantidad de trabajo restante. Por ejemplo, cuando`k`es`3`y el total es`2`, aún quedan dos términos por procesar,`3`y`4`. Por otro lado,*hecho* Se caracteriza por su argumento único`n`El estado del cálculo está completamente contenido dentro de la estructura del árbol de expresión, que tiene valores de retorno que toman el papel de`total`, y se une`n`a diferentes valores en diferentes marcos en lugar de

realizar un seguimiento explícito`k`.

Las funciones recursivas pueden depender más del propio intérprete, almacenando el estado del cálculo como parte del árbol de expresión y del entorno, en lugar de utilizar nombres explícitamente en el marco local. Por este motivo, las funciones recursivas suelen ser más fáciles de definir, porque no necesitamos intentar determinar el estado local que debe mantenerse a lo largo de las iteraciones. Por otro lado, aprender a reconocer los procesos computacionales desarrollados por las funciones recursivas puede requerir algo de práctica.

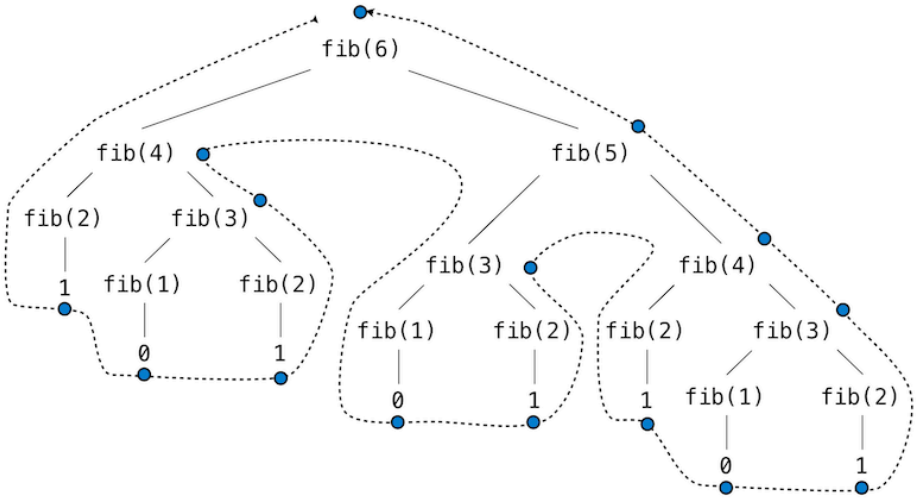
3.2.3 Recursión de árboles

Otro patrón común de cálculo se denomina recursión de árbol. Como ejemplo, considere el cálculo de la secuencia de números de Fibonacci, en la que cada número es la suma de los dos anteriores.

```
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

```
fib(6)
```

Esta definición recursiva es tremendamente atractiva en relación con nuestros intentos anteriores: refleja exactamente la definición familiar de los números de Fibonacci. Considere el patrón de cálculo que resulta de la evaluación`fib(6)`, que se muestra a continuación. Para calcular`fib(6)`, calculamos`fib(5)`y`fib(4)`Para calcular`fib(5)`, calculamos`fib(4)`y`fib(3)`En general, el proceso evolucionado parece un árbol (el diagrama que aparece a continuación no es un árbol de expresión completo, sino una representación simplificada del proceso; un árbol de expresión completo tendría la misma estructura general). Cada punto azul indica un cálculo completo de un número de Fibonacci en el recorrido de este árbol.



Las funciones que se llaman a sí mismas varias veces de esta manera se denominan *árbol recursivo* Esta función es instructiva como recursión de árbol prototípica, pero es una forma terrible de calcular números de Fibonacci porque realiza muchos cálculos redundantes. Observe que todo el cálculo de`fib(4)`-- casi la mitad del trabajo – se duplica. De hecho, no es difícil demostrar que la cantidad de veces que la función calculará`fib(1)`o`fib(2)`(el número de hojas del árbol, en general) es precisamente`fib(n+1)`Para tener una idea de lo malo que es esto, se puede demostrar que el valor de`fib(n)`crece exponencialmente con`n`. De esta forma, el proceso utiliza un número de pasos que crece exponencialmente con la entrada.

Ya hemos visto una implementación iterativa de los números de Fibonacci, repetida aquí para mayor comodidad.

```
def fib_iter(n):
    prev, curr = 1, 0  # curr is the first Fibonacci number.
    for _ in range(n-1):
         prev, curr = curr, prev + curr
    return curr
```

El estado que debemos mantener en este caso consiste en los números de Fibonacci actuales y anteriores. Implícitamente,`for`La declaración también lleva un registro del recuento de iteraciones. Esta definición no refleja la definición matemática estándar de los números de Fibonacci tan claramente como el enfoque recursivo. Sin embargo, la cantidad de cálculo requerida en la implementación iterativa es solo lineal en`n`, en lugar de exponencial. Incluso para valores pequeños de`n`, esta diferencia puede ser enorme.

De esta diferencia no se puede concluir que los procesos recursivos en forma de árbol son inútiles. Si consideramos procesos que operan con datos estructurados jerárquicamente en lugar de números, descubriremos que la recursión en forma

de árbol es una herramienta natural y poderosa. Además, los procesos recursivos en forma de árbol a menudo se pueden hacer más eficientes.

Memorización. Una técnica poderosa para aumentar la eficiencia de las funciones recursivas que repiten el cálculo se llama *memorización* Una función memorizada almacenará el valor de retorno de cualquier argumento que haya recibido previamente. Una segunda llamada a`fib(4)`no desarrollaría el mismo proceso complejo que el primero, sino que devolvería inmediatamente el resultado almacenado calculado por la primera llamada.

La memorización se puede expresar de forma natural como una función de orden superior, que también se puede utilizar como decorador. La siguiente definición crea una *cache* de resultados calculados previamente, indexados por los argumentos a partir de los cuales se calcularon. El uso de un diccionario requerirá que el argumento de la función memorizada sea inmutable en esta implementación.

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

```
63245986
```

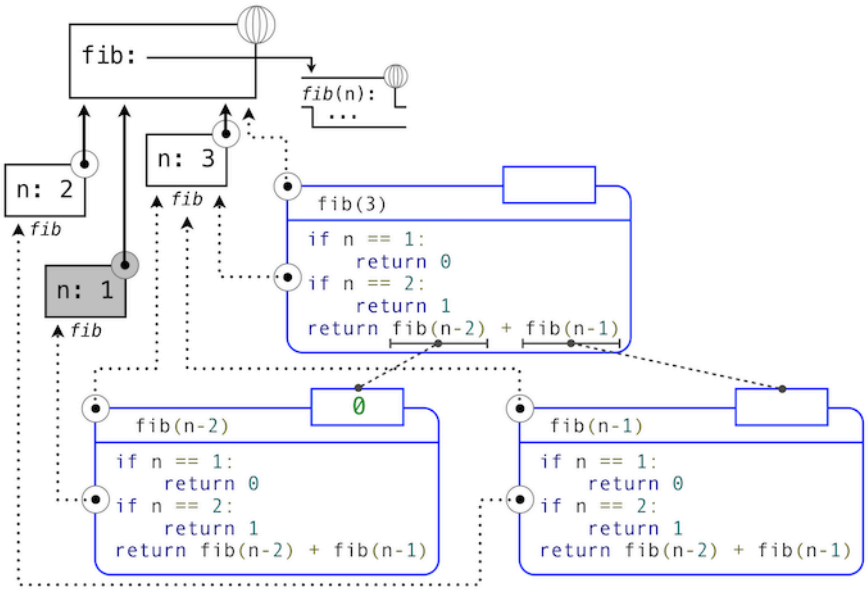
La cantidad de tiempo de cálculo que se ahorra con la memorización en este caso es sustancial. La memorización recursiva *mentira* Función y la iterativa *fib_iter* Ambas funciones requieren una cantidad de tiempo para calcularse que es solo una función lineal de su entrada.`n`Para calcular`fib(40)`, el cuerpo de`fib`se ejecuta 40 veces, en lugar de 102.334.155 veces en el caso recursivo no memorizado.

Espacio. Para comprender los requisitos de espacio de una función, debemos especificar en general cómo se utiliza, se conserva y se recupera la memoria en nuestro modelo de entorno de computación. Al evaluar una expresión, debemos preservar todos los *activo* entornos y todos los valores y marcos a los que hacen referencia esos entornos. Un entorno está activo si proporciona el contexto de evaluación para alguna expresión en la rama actual del árbol de expresiones.

Por ejemplo, al evaluar`fib`, el intérprete procede a calcular cada valor en el orden mostrado anteriormente, recorriendo la estructura del árbol. Para ello, solo necesita llevar un registro de aquellos nodos que se encuentran por encima del nodo actual en el árbol en cualquier punto del cálculo. La memoria utilizada para evaluar el resto de las ramas se puede recuperar porque no puede afectar a los cálculos futuros. En general, el espacio requerido para las funciones recursivas del árbol será proporcional a la profundidad máxima del árbol.

El diagrama a continuación representa el entorno y el árbol de expresión generado al evaluar`fib(3)`. En el proceso de evaluación de la expresión de retorno para la aplicación inicial de`fib`, la expresión`fib(n-2)`se evalúa, dando como

resultado un valor de`0`Una vez calculado este valor, el marco de entorno correspondiente (atenuado) ya no es necesario: no forma parte de un entorno activo. Por lo tanto, un intérprete bien diseñado puede recuperar la memoria que se utilizó para almacenar este marco. Por otro lado, si el intérprete está evaluando actualmente`fib(n-1)`, entonces el entorno creado por esta aplicación de`fib`(en el cual`n`es`2`) está activo. A su vez, el entorno creado originalmente para aplicar`fib`a`3`está activo porque su valor aún no se ha calculado correctamente.



En el caso de`memo`, el entorno asociado con la función que devuelve (que contiene`cache`) debe conservarse mientras exista algún nombre vinculado a esa función en un entorno activo. El número de entradas en el`cache`El diccionario crece linealmente con el número de argumentos únicos que se le pasan.`fib`, que escala linealmente con la entrada. Por otro lado, la implementación iterativa requiere que solo se rastreen dos números durante el cálculo:`prev`y`curr`, dándole un tamaño constante.

La memorización ejemplifica un patrón común en programación: el tiempo de cálculo a menudo se puede reducir a expensas de un mayor uso del espacio, o viceversa.

3.2.4 Ejemplo: Conteo de cambio

Considere el siguiente problema: ¿de cuántas maneras diferentes podemos cambiar $1,00, si tenemos monedas de 50 centavos, 25 centavos, 10 centavos, 5 y 10 centavos? En términos más generales, ¿podemos escribir una función para calcular la cantidad de maneras de cambiar una cantidad determinada de dinero utilizando cualquier conjunto de denominaciones monetarias?

Este problema tiene una solución sencilla como función recursiva. Supongamos que pensamos en los tipos de monedas disponibles dispuestos en algún orden, por ejemplo, de mayor a menor valor.

La cantidad de formas de cambiar una cantidadausandontipos de monedas iguales

1. la cantidad de formas de cambiarautilizando todos los tipos de monedas excepto el primero, más
2. la cantidad de formas de cambiar la cantidad más pequeñaa - dusando todontipos de monedas, dondedes la denominación del primer tipo de moneda.

Para ver por qué esto es cierto, observe que las formas de dar cambio se pueden dividir en dos grupos: las que no utilizan ninguna de las primeras monedas y las que sí las utilizan. Por lo tanto, el número total de formas de dar cambio por una determinada cantidad es igual al número de formas de dar cambio por esa cantidad sin utilizar ninguna de las primeras monedas, más el número de formas de dar cambio suponiendo que sí utilizamos las primeras monedas al menos una vez. Pero este último número es igual al número de formas de dar cambio por la cantidad que queda después de utilizar una moneda de las primeras.

De este modo, podemos reducir recursivamente el problema de cambiar una cantidad dada al problema de cambiar cantidades más pequeñas utilizando menos tipos de monedas. Considere esta regla de reducción con atención y convénzase de que podemos usarla para describir un algoritmo si especificamos los siguientes casos base:

1. Siaes exactamente0Deberíamos contar eso como1manera de hacer el cambio.
2. Siaes menor que0Deberíamos contar eso como0Maneras de hacer el cambio.
3. Sines0Deberíamos contar eso como0Maneras de hacer el cambio.

Podemos traducir fácilmente esta descripción en una función recursiva:

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

```
292
```

Elcount_changeLa función genera un proceso recursivo en forma de árbol con redundancias similares a las de nuestra primera implementación defibTomará bastante tiempo para eso292que se debe calcular, a menos que memoricemos la función. Por otro lado, no resulta obvio cómo diseñar un algoritmo iterativo para calcular el resultado, y dejamos este problema como un desafío.

3.2.5 Órdenes de crecimiento

Los ejemplos anteriores ilustran que los procesos pueden diferir considerablemente en las tasas a las que consumen los recursos computacionales del espacio y el tiempo. Una forma conveniente de describir esta diferencia es utilizar la noción de *orden de crecimiento* para obtener una medida aproximada de los recursos que requiere un proceso a medida que las entradas se hacen mayores.

Sea n un parámetro que mide el tamaño del problema, y sea $R(n)$ la cantidad de recursos que requiere el proceso para un problema de tamaño n. En nuestros ejemplos anteriores tomamos n como el número para el cual se debe calcular una función dada, pero hay otras posibilidades. Por ejemplo, si nuestro objetivo es calcular una aproximación a la raíz cuadrada de un número, podemos tomar n como el número de dígitos de precisión requeridos. En general, hay una serie de propiedades del problema con respecto a las cuales será deseable analizar un proceso dado. De manera similar, $R(n)$ puede medir la cantidad de memoria utilizada, el número de operaciones de máquina elementales realizadas, etc. En computadoras que realizan solo un número fijo de operaciones a la vez, el tiempo requerido para evaluar una expresión será proporcional al número de operaciones de máquina elementales realizadas en el proceso de evaluación.

Decimos que $R(n)$ tiene orden de crecimiento $\Theta(f(n))$, escrito $R(n) = \Theta(f(n))$ (pronunciado "theta de $f(n)$"), si hay constantes positivas k_1 y k_2 independientes de n tales que

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

para cualquier valor suficientemente grande de n. En otras palabras, para n grandes, el

* Un límite inferior $k_1 \cdot f(n)$ y
* Un límite superior $k_2 \cdot f(n)$

Por ejemplo, el número de pasos para calcular $n!$ crece proporcionalmente a la entrada n.

El número de pasos en nuestro cálculo recursivo de Fibonacci en árbol`fib`crece exponencialm

$$\frac{\phi^{n-2}}{\sqrt{5}}$$

donde ϕ es la proporción áurea:

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.6180$$

También afirmamos que el número de pasos escala con el valor resultante y, por lo tanto, el

Los órdenes de crecimiento proporcionan sólo una descripción burda del comportamiento de un

Sin embargo, el orden de crecimiento proporciona una indicación útil de cómo podemos esperar

3.2.6 Ejemplo: exponenciación

Consideremos el problema de calcular la exponencial de un número dado. Nos gustaría una func

$$\begin{array}{l l} b^n &= b \cdot b^{n-1} \ b^0 &= 1 \end{array}$$

que se traduce fácilmente en la función recursiva

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

Este es un proceso recursivo lineal que requiere $\Theta(n)$ pasos y $\Theta(n)$ espacio. Al igual que con el factorial, podemos formular fácilmente una iteración lineal equivalente que requiere una cantidad similar de pasos pero un espacio constante.

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

Podemos calcular exponenciales en menos pasos utilizando el cuadrado sucesivo. Por ejemplo, en lugar de calcular b^8 como

$$b \cdot (b \cdot (b \cdot (b \cdot (b \cdot (b \cdot b))))))$$

Podemos calcularlo usando tres multiplicaciones:

$$\begin{array}{l l} b^2 &= b \cdot b \ b^4 &= b^2 \cdot b^2 \ b^8 &= b^4 \cdot b^4 \end{ar

Este método funciona bien para exponentes que son potencias de2También podemos aprovechar el cuadrado sucesivo al calcular exponenciales en general si usamos la regla recursiva

$$b^n = \left{\begin{array}{l l} (b^{\frac{1}{2} n})^2 & \mbox{si \$n\$ es par} \ b \cdot b

También podemos expresar este método como una función recursiva:

Número de serie 16

```
def starts_with_a_vowel(w):
    """Return whether w begins with a vowel."""
    return w[0].lower() in 'aeiou'
```

Número de serie 18

126765060022822940149670320537 6

El proceso evolucionó mediante`fast_exp`crece logarítmicamente con`n`tanto en el espacio como en el número de pasos. Para ver esto, observe que calcular b^{2n} usando`fast_exp`requiere solo una multiplicación más que calcular b^n.

Por lo tanto, el tamaño del exponente que podemos calcular se duplica (aproximadamente) con cada nueva multiplicación que se nos permite. Por lo tanto, la cantidad de multiplicaciones necesarias para un exponente de`n`crece aproximadamente tan rápido como el logaritmo de`n`base2El proceso tiene un crecimiento $\Theta(\log n)$. La diferencia entre el crecimiento $\Theta(\log n)$ y el crecimiento $\Theta(n)$ se vuelve sorprendente a medida que n se hace grande. Por ejemplo,`fast_exp`para`n`de`1000`requiere solo`14`multiplicaciones en lugar de`1000`.

Contents

3.3 Estructuras de datos recursivas **1**
3.3.1 Procesamiento de listas recursivas 1
3.3.2 Estructuras jerárquicas 3
3.3.3 Conjuntos 5

\tableofcontents

3.3 Estructuras de datos recursivas

En el Capítulo 2, presentamos la noción de par como un mecanismo primitivo para unir dos objetos y formar uno solo. Demostramos que un par se puede implementar utilizando una tupla incorporada._cierre_ La propiedad de los pares indicaba que cualquiera de los elementos de un par podía ser en sí mismo un par.

Esta propiedad de cierre nos permitió implementar la abstracción de datos de lista recursiva, que sirvió como nuestro primer tipo de secuencia. Las listas recursivas se manipulan de forma más natural mediante funciones recursivas, como su nombre y estructura sugieren. En esta sección, analizamos las funciones para crear y manipular listas recursivas y otras estructuras de datos recursivas.

3.3.1 Procesamiento de listas recursivas

Recuerde que el tipo de datos abstracto de lista recursiva representaba una lista como primer elemento y el resto de la lista. Anteriormente implementamos listas recursivas usando funciones, pero en este punto podemos volver a implementarlas usando una clase. A continuación, la longitud (`__len__`) y selección de elementos (`__getitem__`) Las funciones se escriben de forma recursiva para demostrar patrones típicos para procesar listas recursivas.

```
class Rlist(object):
    """A recursive list consisting of a first element and the rest."""
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __repr__(self):
        args = repr(self.first)
        if self.rest is not Rlist.empty:
            args += ', {0}'.format(repr(self.rest))
        return 'Rlist({0})'.format(args)
    def __len__(self):
```

```
        return 1 + len(self.rest)
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
```

Las definiciones de`__len__`y`__getitem__`son de hecho recursivos, aunque no de forma explícita. La función incorporada de Python`len`busca un método llamado`__len__`cuando se aplica a un argumento de objeto definido por el usuario. Del mismo modo, el operador de subíndice busca un método llamado`__getitem__`. Por lo tanto, estas definiciones terminarán llamándose a sí mismas. Las llamadas recursivas al resto de la lista son un patrón omnipresente en el procesamiento recursivo de listas. Esta definición de clase de una lista recursiva interactúa correctamente con las operaciones de impresión y secuencia integradas de Python.

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
Rlist(2, Rlist(3))
```

```
len(s)
```

```
s[1]
```

Las operaciones que crean nuevas listas son particularmente sencillas de expresar mediante recursión. Por ejemplo, podemos definir una función`extend_rlist`, que toma dos listas recursivas como argumentos y combina los elementos de ambas en una nueva lista.

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

```
extend_rlist(s.rest, s)
```

```
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

De manera similar, mapear una función sobre una lista recursiva exhibe un patrón similar.

```
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))
```

```
map_rlist(s, square)
```

```
Rlist(1, Rlist(4, Rlist(9)))
```

El filtrado incluye una declaración condicional adicional, pero por lo demás tiene una estructura recursiva similar.

```
def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
        return Rlist(s.first, rest)
    return rest
```

```
filter_rlist(s, lambda x: x % 2 == 1)
```

```
Rlist(1, Rlist(3))
```

Las implementaciones recursivas de operaciones de lista, en general, no requieren asignación local o`while`En cambio, las listas recursivas se desarman y se construyen de manera incremental como consecuencia de la aplicación de funciones. Como resultado, tienen órdenes lineales de crecimiento tanto en el número de pasos como en el espacio requerido.

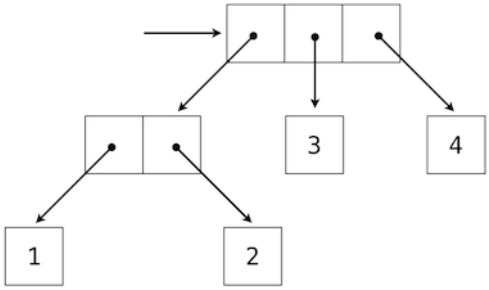
3.3.2 Estructuras jerárquicas

Las estructuras jerárquicas son el resultado de la propiedad de clausura de los datos, que establece, por ejemplo, que las tuplas pueden contener otras tuplas. Por ejemplo, considere esta representación anidada de los números del 1 al 4.

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
((1, 2), 3, 4)
```

Esta tupla es una secuencia de longitud tres, cuyo primer elemento es en sí mismo una tupla. Un diagrama de caja y puntero de esta estructura anidada muestra que también se la puede considerar como un árbol con cuatro hojas, cada una de las cuales es un número.



En un árbol, cada subárbol es en sí mismo un árbol. Como condición básica, cualquier elemento desnudo que no sea una tupla es en sí mismo un árbol simple, uno sin ramas. Es decir, los números son todos árboles, al igual que el par.`(1, 2)`y la estructura en su conjunto.

La recursión es una herramienta natural para tratar con estructuras de árbol, ya

que a menudo podemos reducir las operaciones sobre los árboles a operaciones sobre sus ramas, que a su vez se reducen a operaciones sobre las ramas de las ramas, y así sucesivamente, hasta llegar a las hojas del árbol. Como ejemplo, podemos implementar una`count_leaves`función, que devuelve el número total de hojas de un árbol.

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
4
```

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
((((1, 2), 3, 4), ((1, 2), 3, 4)), 5)
```

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
9
```

Precisamente como`map`es una herramienta poderosa para trabajar con secuencias, el mapeo y la recursión juntos proporcionan una forma general poderosa de cálculo para manipular árboles. Por ejemplo, podemos elevar al cuadrado todas las hojas de un árbol usando una función recursiva de orden superior`map_tree`que está estructurado de manera bastante similar a`count_leaves`.

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

Número de serie 16

```
((((1, 4), 9, 16), ((1, 4), 9, 16)), 25)
```

Valores internos. Los árboles descritos anteriormente tienen valores solo en las hojas. Otra representación común de datos estructurados en forma de árbol también tiene valores para los nodos internos del árbol. Podemos representar dichos árboles mediante una clase.

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

El`Tree`La clase puede representar, por ejemplo, los valores calculados en un árbol de expresión para la implementación recursiva de`fib`, la función para calcular los números de Fibonacci. La función`fib_tree(n)`A continuación se muestra un`Tree`que tiene como número n-ésimo de Fibonacci`entry`y un rastro de todos los números de Fibonacci calculados previamente dentro de sus ramas.

Número de serie 18

```
s = Rlist(1, Rlist(2, Rlist(3)))
s.rest
```

```
Tree(3, Tree(1, Tree(0), Tree(1)), Tree(2, Tree(1), Tree(1, Tree(0), Tree(1))))
```

Este ejemplo muestra que los árboles de expresión se pueden representar mediante programación utilizando datos con estructura de árbol. Esta conexión entre expresiones anidadas y tipos de datos con estructura de árbol desempeña un papel central en nuestro análisis del diseño de intérpretes más adelante en este capítulo.

3.3.3 Conjuntos

Además de la lista, la tupla y el diccionario, Python tiene un cuarto tipo de contenedor incorporado llamado`set`Los conjuntos literales siguen la notación matemática de los elementos encerrados entre llaves. Los elementos duplicados se eliminan durante la construcción. Los conjuntos son colecciones desordenadas, por lo que el orden impreso puede diferir del orden de los elementos en el conjunto literal.

```
len(s)
```

```
{1, 2, 3, 4}
```

Los conjuntos de Python admiten una variedad de operaciones, incluidas pruebas de pertenencia, cálculo de longitud y las operaciones de conjunto estándar de unión e intersección.

```
len(s)
```

```
True
```

```
len(s)
```

```
4
```

```
len(s)
```

```
{1, 2, 3, 4, 5}
```

```
len(s)
```

```
{3, 4}
```

Además de`union`y`intersection`Los conjuntos de Python admiten varios otros métodos. Los predicados`isdisjoint`, `issubset`, y`issuperset`Proporcionar una comparación de conjuntos. Los conjuntos son mutables y se pueden cambiar un elemento a la vez utilizando`add`, `remove`, `discard`, y`pop`. Otros métodos proporcionan mutaciones de múltiples elementos, como`clear`y`update`. La pitóndocumentación para conjuntosDebería ser lo suficientemente inteligible en este punto del curso para completar los detalles.

Implementando conjuntos. De manera abstracta, un conjunto es una colección de objetos distintos que admite pruebas de pertenencia, unión, intersección y adjunción. La unión de un elemento y un conjunto devuelve un nuevo conjunto que contiene todos los elementos del conjunto original junto con el nuevo elemento, si es distinto. La unión y la intersección devuelven el conjunto de elementos que aparecen en uno o ambos conjuntos, respectivamente. Como ocurre con cualquier abstracción de datos, tenemos la libertad de implementar cualquier función sobre cualquier representación de conjuntos que proporcione esta colección de comportamientos.

En el resto de esta sección, consideramos tres métodos diferentes de implementación de conjuntos que varían en su representación. Caracterizaremos la eficiencia de estas diferentes representaciones analizando el orden de crecimiento de las operaciones de conjuntos. Usaremos nuestro`RlistyTree`clases de antes en esta sección, que permiten soluciones recursivas simples y elegantes para operaciones de conjuntos elementales.

Conjuntos como secuencias desordenadas. Una forma de representar un conjunto es como una secuencia en la que ningún elemento aparece más de una vez. El conjunto vacío se representa mediante la secuencia vacía. La prueba de pertenencia recorre la lista de forma recursiva.

```
len(s)
```

```
len(s)
```

```
len(s)
```

```
True
```

```
len(s)
```

```
False
```

Esta implementación de`set_contains`requiere tiempo $\Theta(n)$ para probar la pertenencia de un elemento, donde n es el tamaño del conjunto`s`Usando esta función de tiempo lineal para la pertenencia, podemos agregar un elemento a un conjunto, también en tiempo lineal.

```
len(s)
```

```
s[1]
```

```
Rlist(4, Rlist(1, Rlist(2, Rlist(3))))
```

Al diseñar una representación, una de las cuestiones que nos debe preocupar es la eficiencia. La intersección de dos conjuntos`set1`y`set2`También requiere pruebas de membresía, pero esta vez cada elemento de`set1`Debe ser examinado para ser miembro`set2`, lo que conduce a un orden cuadrático de crecimiento en el número de pasos, $\Theta(n^2)$, para dos conjuntos de tamaño n.

```
s[1]
```

```
s[1]
```

```
Rlist(4, Rlist(1))
```

Al calcular la unión de dos conjuntos, debemos tener cuidado de no incluir ningún elemento dos veces.`union_set`La función también requiere un número lineal de pruebas de pertenencia, creando un proceso que también incluye $\Theta(n^2)$ pasos.

```
s[1]
```

```
s[1]
```

```
Rlist(4, Rlist(1, Rlist(2, Rlist(3))))
```

Conjuntos como tuplas ordenadas. Una forma de acelerar nuestras operaciones de conjunto es cambiar la representación para que los elementos del conjunto se enumeren en orden creciente. Para ello, necesitamos alguna forma de comparar dos objetos para poder decir cuál es más grande. En Python, se pueden comparar muchos tipos diferentes de objetos usando<y>operadores, pero en este ejemplo nos concentraremos en los números. Representaremos un conjunto de números enumerando sus elementos en orden creciente.

Una ventaja de realizar pedidos se muestra en`set_contains`:Para comprobar la presencia de un objeto, ya no tenemos que explorar todo el conjunto. Si encontramos un elemento del conjunto que es más grande que el elemento que estamos buscando, entonces sabemos que el elemento no está en el conjunto:

```
s[1]
```

```
s[1]
```

```
False
```

¿Cuántos pasos se ahorran con esto? En el peor de los casos, el elemento que buscamos puede ser el más grande del conjunto, por lo que el número de pasos es el mismo que para la representación desordenada. Por otro lado, si buscamos elementos de muchos tamaños diferentes, podemos esperar que a veces podamos detener la búsqueda en un punto cerca del comienzo de la lista y que otras veces aún necesitemos examinar la mayor parte de la lista. En promedio, deberíamos esperar tener que examinar aproximadamente la mitad de los elementos del conjunto. Por lo tanto, el número promedio de pasos necesarios será de aproximadamente $\frac{n}{2}$. Esto sigue siendo un crecimiento $\Theta(n)$, pero nos ahorra, en promedio, un factor de2en el número de pasos con respecto a la implementación anterior.

Podemos obtener una aceleración más impresionante si volvemos a implementar`intersect_set`En la representación desordenada, esta operación requirió $\Theta(n^2)$ pasos porque realizamos un escaneo completo de`set2`para cada elemento de`set1`Pero con la representación ordenada, podemos utilizar un método más inteligente. Iteramos a través de ambos conjuntos simultáneamente, rastreando

un elemento.e1ense1ye2enset2. Cuandoe1ye2son iguales, incluimos ese elemento en la intersección.

Supongamos, sin embargo, quee1es menor quee2. Desdee2es más pequeño que los elementos restantes dese2, podemos concluir inmediatamente quee1No puede aparecer en ningún lugar del resto dese2y por lo tanto no está en la intersección. Por lo tanto, ya no necesitamos considerare1;lo descartamos y procedemos al siguiente elemento dese1Una lógica similar avanza a través de los elementos dese2cuandoe2 < e1Aquí está la función:

```
s[1]
```

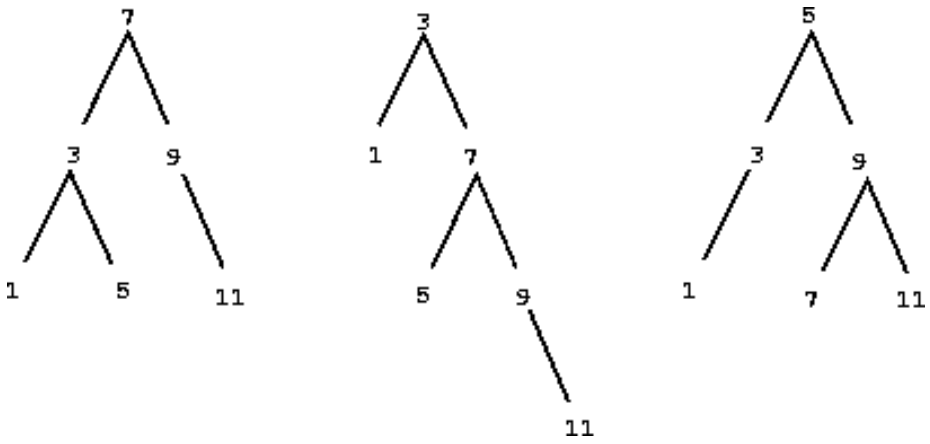
```
s[1]
```

```
Rlist(2, Rlist(3))
```

Para estimar el número de pasos que requiere este proceso, observe que en cada paso reducimos el tamaño de al menos uno de los conjuntos. Por lo tanto, el número de pasos requeridos es como máximo la suma de los tamaños dese1yset2, en lugar del producto de los tamaños, como en la representación desordenada. Esto es un crecimiento de $\Theta(n)$ en lugar de $\Theta(n^2)$, una aceleración considerable, incluso para conjuntos de tamaño moderado. Por ejemplo, la intersección de dos conjuntos de tamaño100tomará alrededor200pasos, en lugar de10,000para la representación desordenada.

La adjunción y la unión de conjuntos representados como secuencias ordenadas también se pueden calcular en tiempo lineal. Estas implementaciones se dejan como ejercicio.

Conjuntos como árboles binarios. Podemos hacerlo mejor que la representación de lista ordenada organizando los elementos del conjunto en forma de árbol. Usamos elTreeClase introducida anteriormente. Laentryde la raíz del árbol contiene un elemento del conjunto. Las entradas dentro de laleftLa rama incluye todos los elementos más pequeños que el de la raíz. Las entradas en larightLa rama incluye todos los elementos mayores que el que está en la raíz. La figura siguiente muestra algunos árboles que representan el conjunto.{1, 3, 5, 7, 9, 11}El mismo conjunto puede representarse mediante un árbol de varias maneras diferentes. Lo único que necesitamos para una representación válida es que todos los elementos del árbolleftEl subárbol debe ser más pequeño que el árbol.entryy que todos los elementos de larightel subárbol será más grande.



La ventaja de la representación en árbol es la siguiente: supongamos que queremos comprobar si un valorvestá contenido en un conjunto. Empecemos comparandovcon`entry`. Sives menor que esto, sabemos que solo necesitamos buscar el`left`subárbol; sives mayor, solo necesitamos buscar el`right`subárbol. Ahora bien, si el árbol está "equilibrado", cada uno de estos subárboles tendrá aproximadamente la mitad del tamaño del original. Por lo tanto, en un solo paso hemos reducido el problema de buscar un árbol de tamaño n a buscar un árbol de tamaño $\frac{n}{2}$. Dado que el tamaño del árbol se reduce a la mitad en cada paso, deberíamos esperar que el número de pasos necesarios para buscar un árbol crezca como $\Theta(\log n)$. Para conjuntos grandes, esto será una aceleración significativa con respecto a las representaciones anteriores.`set_contains`La función explota la estructura de ordenamiento del conjunto estructurado en árbol.

`s[1]`

La anexión de un elemento a un conjunto se implementa de manera similar y también requiere $\Theta(\log n)$ pasos. Para anexar un valorv, comparamosvcon`entry`Para determinar sivDebería añadirse a la`right`o al`left`trama, y habiéndosele anexadovEn la rama correspondiente, unimos esta rama recién construida con la original.`entry`y la otra rama. Sives igual a la`entry`, simplemente devolvemos el nodo. Si se nos pide que lo adjuntemosvA un árbol vacío, le generamos un`Tree`Eso tienevcomo el`entry`y vacio`right`y`left`tramas. Aquí está la función:

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

```
Tree(2, Tree(1), Tree(3))
```

Nuestra afirmación de que la búsqueda en el árbol se puede realizar en un número logarítmico de pasos se basa en el supuesto de que el árbol está "equilibrado", es decir, que el subárbol izquierdo y el derecho de cada árbol tienen aproximadamente el mismo número de elementos, de modo que cada subárbol contiene aproximadamente la mitad de los elementos de su padre. Pero, ¿cómo podemos estar seguros de que los árboles que construimos estarán equilibrados? Incluso si comenzamos con un árbol equilibrado, agregando elementos con`adjoin_set`Puede producir un resultado desequilibrado. Dado que la posición de un elemento recién añadido depende de cómo se compara el elemento con los elementos que ya están en el conjunto, podemos esperar que si agregamos elementos "al azar", el árbol tenderá a estar equilibrado en promedio.

Pero esto no es una garantía. Por ejemplo, si empezamos con un conjunto vacío y unimos los números del 1 al 7 en secuencia, terminamos con un árbol altamente desequilibrado en el que todos los subárboles izquierdos están vacíos, por lo que no tiene ninguna ventaja sobre una lista ordenada simple. Una forma de resolver este problema es definir una operación que transforme un árbol arbitrario en un árbol equilibrado con los mismos elementos. Podemos realizar esta transformación cada poco`adjoin_set`operaciones para mantener nuestro conjunto en equilibrio.

Las operaciones de intersección y unión se pueden realizar en conjuntos con estructura de árbol en tiempo lineal convirtiéndolos en listas ordenadas y viceversa. Los detalles se dejan como ejercicio.

Implementación de conjuntos de Python. El`set`El tipo que está integrado en Python no utiliza ninguna de estas representaciones internamente. En su lugar, Python utiliza una representación que proporciona pruebas de pertenencia en tiempo constante y operaciones adjuntas basadas en una técnica llamada *Hash* , que es un tema para otro curso. Los conjuntos integrados de Python no pueden contener tipos de datos mutables, como listas, diccionarios u otros conjuntos. Para permitir conjuntos anidados, Python también incluye un conjunto inmutable integrado`frozenset`clase que comparte métodos con la`set`clase pero excluye los métodos y operadores de mutación.

Contents

3.4 Excepciones **1**
3.4.1 Objetos de excepción 3

\tableofcontents

3.4 Excepciones

Los programadores deben estar siempre atentos a los posibles errores que puedan surgir en sus programas. Los ejemplos abundan: una función puede no recibir los argumentos que está diseñada para aceptar, puede faltar un recurso necesario o puede perderse una conexión a través de una red. Al diseñar un programa, uno debe anticipar las circunstancias excepcionales que pueden surgir y tomar las medidas adecuadas para manejarlas.

No existe un único enfoque correcto para gestionar los errores en un programa. Los programas diseñados para proporcionar algún servicio persistente, como un servidor web, deben ser resistentes a los errores, registrándolos para su posterior consideración, pero continuando con el servicio de nuevas solicitudes durante el mayor tiempo posible. Por otro lado, el intérprete de Python gestiona los errores finalizando inmediatamente e imprimiendo un mensaje de error, de modo que los programadores puedan solucionar los problemas tan pronto como surjan. En cualquier caso, los programadores deben tomar decisiones conscientes sobre cómo deben reaccionar sus programas ante condiciones excepcionales.

Excepciones , el tema de esta sección, proporciona un mecanismo general para agregar lógica de manejo de errores a los programas._Generar una excepción_ es una técnica para interrumpir el flujo normal de ejecución de un programa, señalando que ha surgido alguna circunstancia excepcional y volviendo directamente a una parte del programa que estaba designada para reaccionar ante esa circunstancia. El intérprete de Python lanza una excepción cada vez que detecta un error en una expresión o declaración. Los usuarios también pueden lanzar excepciones con`raise`y`assert`declaraciones.

Plantear excepciones. Una excepción es una instancia de objeto con una clase que hereda, ya sea directa o indirectamente, de la`BaseException`clase. La`assert`La declaración introducida en el Capítulo 1 genera una excepción con la clase`AssertionError`En general, cualquier instancia de excepción se puede generar con el`raise`Declaración. La forma general de las declaraciones raise se describe en laDocumentación de PythonEl uso más común de`raise`construye una instancia de excepción y la genera.

```
raise Exception('An error occurred')
```

```
Traceback (most recent call last):
  File "", línea 1, enExcepción: se produjo un error
```

Cuando se genera una excepción, no se ejecutan más instrucciones en el bloque de código actual. A menos que se genere la excepción._manejado_(descrito a continuación), el intérprete regresará directamente al bucle interactivo read-eval-print o finalizará por completo si Python se inició con un argumento de archivo. Además, el intérprete imprimirá un *seguimiento de pila* , que es un bloque de texto estructurado que describe el conjunto anidado de llamadas de función activas en la rama de ejecución en la que se generó la excepción. En el ejemplo anterior, el nombre del archivo`<stdin>`indica que la excepción fue generada por el usuario en una sesión interactiva, en lugar de desde el código en un archivo.

Manejo de excepciones. Una excepción puede ser manejada mediante un envoltorio`try`Declaración. A`try`La declaración consta de varias cláusulas; la primera comienza con`try`y el resto empieza con`except`:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

El`<try suite>`siempre se ejecuta inmediatamente cuando el`try`Se ejecuta la sentencia. Suites de la`except`Las cláusulas solo se ejecutan cuando se genera una excepción durante el curso de la ejecución de la`<try suite>`. Cada`except`La cláusula especifica la clase particular de excepción que se debe manejar. Por ejemplo, si la`<exception class>`es`AssertionError`, entonces cualquier instancia de una clase que herede de`AssertionError`que se genera durante el transcurso de la ejecución de la`<try suite>`será manejado por los siguientes`<except suite>`. Dentro de la`<except suite>`, el identificador`<name>`está vinculado al objeto de excepción que se generó, pero este vínculo no persiste más allá del`<except suite>`.

Por ejemplo, podemos manejar un`ZeroDivisionError`excepción usando un`try`Declaración que vincula el nombre`x`a`0`cuando se plantea la excepción.

```
try:
    x = 1/0
except ZeroDivisionError as e:
    print('handling a', type(e))
    x = 0
```

```
handling a
```

```
x
```

A`try`La declaración manejará las excepciones que ocurran dentro del cuerpo de una función que se aplica (ya sea directa o indirectamente) dentro de la`<try suite>`Cuando se genera una excepción, el control salta directamente al cuerpo de la`<except suite>`De lo más reciente`try`declaración que maneja ese tipo de excepción.

```
def invert(x):
    result = 1/x  # Raises a ZeroDivisionError if x is 0
    print('Never printed if x is 0')
    return result

def invert_safe(x):
    try:
        return invert(x)
    except ZeroDivisionError as e:
        return str(e)

invert_safe(2)
```

```
Never printed if x is 0
0.5
```

```
invert_safe(0)
```

```
'division by zero'
```

Este ejemplo ilustra que la`print`expresión en`invert`nunca se evalúa y, en su lugar, el control se transfiere a la suite de`except`cláusula en`handler`Coaccionar a la`ZeroDivisionError` ea una cadena da la cadena interpretable por humanos devuelta por`handler`: `'division by zero'`.

3.4.1 Objetos de excepción

Los objetos de excepción en sí mismos llevan atributos, como el mensaje de error indicado en una`assert`Declaración e información sobre en qué momento de la ejecución se generó la excepción. Las clases de excepción definidas por el usuario pueden incluir atributos adicionales.

En el Capítulo 1, implementamos el método de Newton para encontrar los ceros de funciones arbitrarias. El siguiente ejemplo define una clase de excepción que devuelve la mejor estimación descubierta en el curso de la mejora iterativa siempre que una`ValueError`ocurre. Un error de dominio matemático (un tipo de`ValueError`) se eleva cuando`sqrt`se aplica a un número negativo. Esta excepción se maneja generando una`IterImproveError`que almacena la estimación más reciente del método de Newton como un atributo.

Primero, definimos una nueva clase que hereda de`Exception`.

```
class IterImproveError(Exception):
    def __init__(self, last_guess):
        self.last_guess = last_guess
```

A continuación, definimos una versión de`IterImprove`, nuestro algoritmo genérico de mejora iterativa. Esta versión maneja cualquier`ValueError`levantando una`IterImproveError`que almacena la estimación más reciente. Como antes,`iter_improve`toma como argumentos dos funciones, cada una de las cuales toma un único argumento numérico.`update`La función devuelve nuevas

conjeturas, mientras que la`done`La función devuelve un valor booleano que indica que la mejora ha convergido a un valor correcto.

```
def iter_improve(update, done, guess=1, max_updates=1000):
    k = 0
    try:
        while not done(guess) and k < max_updates:
            guess = update(guess)
            k = k + 1
        return guess
    except ValueError:
        raise IterImproveError(guess)
```

Finalmente, definimos`find_root`, que devuelve el resultado de`iter_improve`aplicado a una función de actualización de Newton devuelta por`newton_update`, que se define en el Capítulo 1 y no requiere cambios para este ejemplo. Esta versión de`find_root`maneja un`IterImproveError`devolviendo su última conjetura.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

Considere aplicar`find_root`para encontrar el cero de la función $2x^2 + \sqrt{x}$. Esta función tiene un cero en`0`, pero evaluarlo en cualquier número negativo generará un`ValueError`Nuestra implementación del método de Newton del Capítulo 1 generaría ese error y no devolvería ninguna aproximación al cero. Nuestra implementación revisada devuelve la última aproximación encontrada antes del error.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

```
-0.030211203830201594
```

Si bien esta aproximación aún está lejos de la respuesta correcta de`0`Algunas aplicaciones preferirían esta aproximación aproximada a una`ValueError`.

Las excepciones son otra técnica que nos ayuda como programas a separar las preocupaciones de nuestro programa en partes modulares. En este ejemplo, el mecanismo de excepción de Python nos permitió separar la lógica para la mejora iterativa, que aparece sin cambios en el conjunto de`try`cláusula, de la lógica de manejo de errores, que aparece en`except`cláusulas. También descubriremos que las excepciones son una característica muy útil al implementar intérpretes en Python.

Contents

3.5 Intérpretes para lenguas con combinación **1**
3.5.1 Calculadora 2
3.5.2 Análisis 7

\tableofcontents

3.5 Intérpretes para lenguas con combinación

El software que se ejecuta en cualquier computadora moderna está escrito en una variedad de lenguajes de programación. Existen lenguajes físicos, como los lenguajes de máquina para computadoras particulares. Estos lenguajes se ocupan de la representación de datos y el control en términos de bits individuales de almacenamiento e instrucciones de máquina primitivas. El programador de lenguaje de máquina se ocupa de utilizar el hardware dado para construir sistemas y utilidades para la implementación eficiente de cálculos con recursos limitados. Los lenguajes de alto nivel, erigidos sobre un sustrato de lenguaje de máquina, ocultan preocupaciones sobre la representación de datos como colecciones de bits y la representación de programas como secuencias de instrucciones primitivas. Estos lenguajes tienen medios de combinación y abstracción, como la definición de procedimientos, que son apropiados para la organización a gran escala de sistemas de software.

Abstracción metalingüística -- el establecimiento de nuevos lenguajes – desempeña un papel importante en todas las ramas del diseño de ingeniería. Es particularmente importante para la programación informática, porque en la programación no sólo podemos formular nuevos lenguajes, sino que también podemos implementar estos lenguajes mediante la construcción de intérpretes. Un intérprete de un lenguaje de programación es una función que, cuando se aplica a una expresión del lenguaje, realiza las acciones necesarias para evaluar esa expresión.

Ahora nos embarcaremos en un recorrido por la tecnología mediante la cual se establecen los lenguajes en términos de otros lenguajes. Primero definiremos un intérprete para un lenguaje limitado llamado Calculator que comparte la sintaxis de las expresiones de llamada de Python. Luego desarrollaremos un esbozo de intérpretes para los lenguajes Scheme y Logo, que son dialectos de Lisp, el segundo lenguaje más antiguo que todavía se usa ampliamente en la actualidad. El intérprete que creemos será completo en el sentido de que nos permitirá escribir programas totalmente generales en Logo. Para ello, implementará el modelo de entorno de evaluación que hemos desarrollado a lo largo de este texto.

3.5.1 Calculadora

Nuestro primer lenguaje nuevo es Calculator, un lenguaje de expresión para las operaciones aritméticas de suma, resta, multiplicación y división. Calculator comparte la sintaxis de expresión de llamada de Python, pero sus operadores son más flexibles en cuanto al número de argumentos que aceptan. Por ejemplo, los operadores de Calculator`add`y`mul`Tome un número arbitrario de argumentos:

```
calc> add(1, 2, 3, 4)
10
calc> mul()
1
```

El`sub`El operador tiene dos comportamientos. Con un argumento, niega el argumento. Con al menos dos argumentos, resta todos menos el primero del primero.`div`El operador tiene la semántica de Python.`operator.truediv`función y toma exactamente dos argumentos:

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

Al igual que en Python, la anidación de expresiones de llamada proporciona un medio de combinación en el lenguaje Calculator. Para condensar la notación, los nombres de los operadores también se pueden reemplazar por sus símbolos estándar:

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

Implementaremos un intérprete para Calculator en Python. Es decir, escribiremos un programa Python que tome una cadena como entrada y devuelva el resultado de evaluar esa cadena si es una expresión de Calculator bien formada o genere una excepción apropiada si no lo es. El núcleo del intérprete para el lenguaje Calculator es una función recursiva llamada`calc_eval`que evalúa un objeto de expresión estructurado en árbol.

Árboles de expresión. Hasta este punto del curso, los árboles de expresión han sido entidades conceptuales a las que nos hemos referido al describir el proceso de evaluación; nunca antes habíamos representado explícitamente árboles de expresión como datos en nuestros programas. Para escribir un intérprete, debemos operar sobre expresiones como datos. En el transcurso de este capítulo, muchos de los conceptos introducidos en los capítulos anteriores finalmente se implementarán en código.

Una expresión primitiva es simplemente un número en la Calculadora, ya sea un`int`o`float`tipo. Todas las expresiones combinadas son expresiones de llamada. Una expresión de llamada se representa como una clase`Exp`que tiene dos instancias de atributo.`operator`En Calculadora siempre hay una cadena: un nombre o símbolo de operador aritmético.`operands`son expresiones primitivas o en sí mismas instancias de`Exp`.

```
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
    def __repr__(self):
        return 'Exp({0}, {1})'.format(repr(self.operator), repr(self.operands))
    def __str__(self):
        operand_strs = ', '.join(map(str, self.operands))
        return '{0}({1})'.format(self.operator, operand_strs)
```

Un`Exp`La instancia define dos métodos de cadena.`__repr__`El método devuelve una expresión de Python, mientras que`__str__`El método devuelve una expresión de calculadora.

```
Exp('add', [1, 2])
```

```
Exp('add', [1, 2])
```

```
str(Exp('add', [1, 2]))
```

```
'add(1, 2)'
```

```
Exp('add', [1, Exp('mul', [2, 3, 4])])
```

```
Exp('add', [1, Exp('mul', [2, 3, 4])])
```

```
str(Exp('add', [1, Exp('mul', [2, 3, 4])]))
```

```
'add(1, mul(2, 3, 4))'
```

Este último ejemplo demuestra cómo el`Exp`La clase representa la estructura jerárquica en árboles de expresión al incluir instancias de`Exp`como elementos de`operands`.

Evaluación. El`calc_eval`La función toma una expresión como argumento y devuelve su valor. Clasifica la expresión por su forma y dirige su evaluación. Para Calculator, las únicas dos formas sintácticas de expresiones son números y expresiones de llamada, que son`Exp`instancias. Los números son *autoevaluación* ; se pueden devolver directamente desde`calc_eval`Las expresiones de llamada requieren la aplicación de la función.

```
def calc_eval(exp):
    """Evaluate a Calculator expression."""
    if type(exp) in (int, float):
```

```
        return exp
    elif type(exp) == Exp:
        arguments = list(map(calc_eval, exp.operands))
        return calc_apply(exp.operator, arguments)
```

Las expresiones de llamada se evalúan primero asignando recursivamente la`calc_eval`función a la lista de operandos para calcular una lista de`arguments`Luego, el operador se aplica a esos argumentos en una segunda función,`calc_apply`.

El lenguaje de la calculadora es lo suficientemente simple como para que podamos expresar fácilmente la lógica de aplicar cada operador en el cuerpo de una sola función.`calc_apply`, cada cláusula condicional corresponde a la aplicación de un operador.

```
from operator import mul
from functools import reduce
def calc_apply(operator, args):
    """Apply the named operator to a list of args."""
    if operator in ('add', '+'):
        return sum(args)
    if operator in ('sub', '-'):
        if len(args) == 0:
            raise TypeError(operator + ' requires at least 1 argument')
        if len(args) == 1:
            return -args[0]
        return sum(args[:1] + [-arg for arg in args[1:]])
    if operator in ('mul', '*'):
        return reduce(mul, args, 1)
    if operator in ('div', '/'):
        if len(args) != 2:
            raise TypeError(operator + ' requires exactly 2 arguments')
        numer, denom = args
        return numer/denom
```

Arriba, cada suite calcula el resultado de un operador diferente, o genera un operador apropiado.`TypeError`cuando se proporciona un número incorrecto de argumentos.`calc_apply`La función se puede aplicar directamente, pero se le debe pasar una lista de *valores* como argumentos en lugar de una lista de expresiones de operandos.

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

```
6
```

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

```
4
```

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

```
1
```

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

```
8.0
```

El papel de`calc_eval`es hacer las llamadas adecuadas a`calc_apply`calculando primero el valor de las subexpresiones del operando antes de pasarlas como argumentos a`calc_apply`. De este modo,`calc_eval`Puede aceptar una expresión anidada.

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

```
'add(2, mul(4, 6))'
```

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

```
26
```

La estructura de`calc_eval`es un ejemplo de envío por tipo: la forma de la expresión. La primera forma de expresión es un número, que no requiere un paso de evaluación adicional. En general, las expresiones primitivas que no requieren un paso de evaluación adicional se denominan *autoevaluación* Las únicas expresiones autoevaluables en nuestro lenguaje Calculadora son números, pero un lenguaje de programación general también podría incluir cadenas, valores booleanos, etc.

Bucles de lectura-evaluación-impresión. Un enfoque típico para interactuar con un intérprete es a través de un bucle de lectura-evaluación-impresión, o REPL, que es un modo de interacción que lee una expresión, la evalúa e imprime el resultado para el usuario. La sesión interactiva de Python es un ejemplo de este tipo de bucle.

Una implementación de un REPL puede ser en gran medida independiente del intérprete que utiliza. La función`read_eval_print_loop`A continuación se toma como entrada una línea de texto del usuario con la función incorporada`input`Función. Construye un árbol de expresión utilizando la función específica del lenguaje.`calc_parse`función, definida en la siguiente sección sobre análisis. Finalmente, imprime el resultado de aplicar`calc_eval`al árbol de expresión devuelto por`calc_parse`.

Número de serie 16

Esta versión de`read_eval_print_loop`Contiene todos los componentes esenciales de una interfaz interactiva. Una sesión de ejemplo sería la siguiente:

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

Esta implementación de bucle no tiene ningún mecanismo para la terminación o el manejo de errores. Podemos mejorar la interfaz informando los errores al usuario. También podemos permitir que el usuario salga del bucle mediante una señal de interrupción del teclado (`Control-C`en UNIX) o excepción de fin de archivo (`Control-D`en UNIX). Para permitir estas mejoras, colocamos la suite original de`while`declaración dentro de una`try`Declaración. La primera`except`Cláusula maneja`SyntaxError`excepciones planteadas por`calc_parse`así como`TypeError`y`ZeroDivisionError`excepciones planteadas por`calc_eval`.

Número de serie 18

Esta implementación de bucle informa los errores sin salir del bucle. En lugar de salir del programa en caso de error, reiniciar el bucle después de un mensaje de error permite a los usuarios revisar sus expresiones. Al importar

el`readline`módulo, los usuarios pueden incluso recordar sus entradas anteriores usando la flecha hacia arriba o`Control-P`El resultado final proporciona una interfaz informativa de informes de errores:

```
calc> sub(10, 1, 2, 3)
4
calc> sub(3)
-3
calc> div(15, 12)
1.25
```

A medida que generalizamos nuestro intérprete a nuevos lenguajes distintos de Calculator, veremos que`read_eval_print_loop`está parametrizado por una función de análisis, una función de evaluación y los tipos de excepción manejados por la`try`Declaración. Más allá de estos cambios, todos los REPL se pueden implementar utilizando la misma estructura.

3.5.2 Análisis

El análisis sintáctico es el proceso de generar árboles de expresión a partir de una entrada de texto sin formato. La función de evaluación tiene la tarea de interpretar esos árboles de expresión, pero el analizador sintáctico debe proporcionar árboles de expresión bien formados al evaluador. Un analizador sintáctico es, de hecho, una composición de dos componentes: un analizador léxico y un analizador sintáctico. En primer lugar, el analizador léxico divide la cadena de entrada en *fichas* , que son las unidades sintácticas mínimas del lenguaje, como los nombres y los símbolos. En segundo lugar, el analizador sintáctico construye un árbol de expresiones a partir de esta secuencia de tokens.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

La secuencia de tokens producida por el analizador léxico, llamada`tokenize`, es consumido por el analizador sintáctico, llamado`analyze`En este caso, definimos`calc_parse`esperar solo una expresión de Calculator bien formada. Los analizadores de algunos lenguajes están diseñados para aceptar múltiples expresiones delimitadas por caracteres de nueva línea, punto y coma o incluso espacios. Aplazamos esta complejidad adicional hasta que presentemos el lenguaje Logo a continuación.

Análisis léxico. El componente que interpreta una cadena como una secuencia de tokens se llama *Tokenizador* o *analizador léxico* En nuestra implementación, el tokenizador es una función llamada`tokenize`El lenguaje de la calculadora consta de símbolos que incluyen números, nombres de operadores y símbolos de operadores, como`+`Estos símbolos siempre están separados por dos tipos de delimitadores: comas y paréntesis. Cada símbolo es su propio token, al igual

que cada coma y paréntesis. Los tokens se pueden separar agregando espacios a la cadena de entrada y luego dividiendo la cadena en cada espacio.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

Al tokenizar una expresión de Calculadora bien formada se conservan los nombres intactos, pero se separan todos los símbolos y delimitadores.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

```
['add', '(', '2', ',', 'mul', '(', '4', ',', '6', ')', ')']
```

Los lenguajes con una sintaxis más complicada pueden requerir un tokenizador más sofisticado. En particular, muchos tokenizadores resuelven el tipo sintáctico de cada token devuelto. Por ejemplo, el tipo de un token en Calculator puede ser un operador, un nombre, un número o un delimitador. Esta clasificación puede simplificar el proceso de *Analizando* La secuencia de tokens.

Análisis sintáctico. El componente que interpreta una secuencia de tokens como un árbol de expresión se denomina *analizador sintáctico* En nuestra implementación, el análisis sintáctico se realiza mediante una función recursiva llamada`analyze`Es recursiva porque analizar una secuencia de tokens a menudo implica analizar una subsecuencia de esos tokens en un árbol de expresión, que a su vez sirve como una rama (es decir, operando) de un árbol de expresión más grande. La recursión genera las estructuras jerárquicas que consume el evaluador.

El`analyze`La función espera una lista de tokens que comience con una expresión bien formada. Analiza el primer token, convirtiendo las cadenas que representan números en valores numéricos. Luego debe considerar los dos tipos de expresión legales en el lenguaje Calculator. Los tokens numéricos son en sí mismos árboles de expresión primitivos y completos. Las expresiones combinadas comienzan con un operador y siguen con una lista de expresiones de operandos delimitadas por paréntesis. Los operandos son analizados por el`analyze_operands`función, que llama recursivamente`analyze`en cada expresión de operando. Comenzamos con una implementación que no verifica errores de sintaxis.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
```

```
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

Por último, necesitamos implementar`analyze_token`. El`analyze_token`Función que convierte literales numéricos en números. En lugar de implementar esta lógica nosotros mismos, confiamos en la coerción de tipos incorporada de Python, utilizando el`int`y`float`constructores para convertir tokens a esos tipos.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

Nuestra implementación de`analyze`está completo; analiza correctamente expresiones de Calculadora bien formadas en árboles de expresión. Estos árboles se pueden convertir nuevamente en expresiones de Calculadora mediante el comando`str`función.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

```
Exp('add', [2, Exp('mul', [4, 6])])
```

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

```
'add(2, mul(4, 6))'
```

El`analyze`La función está pensada para devolver únicamente árboles de expresión bien formados, por lo que debe detectar errores en la sintaxis de su entrada. En particular, debe detectar que las expresiones estén completas, correctamente delimitadas y utilicen únicamente operadores conocidos. Las siguientes revisiones garantizan que cada paso del análisis sintáctico encuentre el token que espera.

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

```
calc> sub(100, mul(7, add(8, div(-12, -3))))
16.0
calc> -(100, *(7, +(8, /(-12, -3))))
16.0
```

```
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
    def __repr__(self):
        return 'Exp({0}, {1})'.format(repr(self.operator), repr(self.operands))
    def __str__(self):
        operand_strs = ', '.join(map(str, self.operands))
        return '{0}({1})'.format(self.operator, operand_strs)
```


```
class Exp(object):
    """A call expression in Calculator."""
    def __init__(self, operator, operands):
        self.operator = operator
        self.operands = operands
    def __repr__(self):
        return 'Exp({0}, {1})'.format(repr(self.operator), repr(self.operands))
    def __str__(self):
        operand_strs = ', '.join(map(str, self.operands))
        return '{0}({1})'.format(self.operator, operand_strs)
```


Los errores de sintaxis informativos mejoran sustancialmente la usabilidad de un intérprete. Arriba, el`SyntaxError`Las excepciones que se generan incluyen una descripción del problema detectado. Estas cadenas de error también sirven para documentar las definiciones de estas funciones de análisis.

Esta definición completa nuestro intérprete de Calculator. Un solo archivo fuente de Python 3calc.pyestá disponible para su experimentación. Nuestro intérprete es resistente a los errores, en el sentido de que cada entrada que un usuario ingresa en el`calc>`El mensaje se evaluará como un número o generará un error apropiado que describe por qué la entrada no es una expresión de Calculadora bien formada.

Contents

4.1 Introducción **1**

\tableofcontents

4.1 Introducción

Hasta ahora, nos hemos centrado en cómo crear, interpretar y ejecutar programas. En el capítulo 1, aprendimos a utilizar funciones como medio de combinación y abstracción. El capítulo 2 nos mostró cómo representar datos y manipularlos con estructuras y objetos de datos, y nos presentó el concepto de abstracción de datos. En el capítulo 3, aprendimos cómo se interpretan y ejecutan los programas informáticos. El resultado es que entendemos cómo diseñar programas para que los ejecute un solo procesador.

En este capítulo, abordaremos el problema de la coordinación de múltiples computadoras y procesadores. Primero, analizaremos los sistemas distribuidos. Se trata de grupos interconectados de computadoras independientes que necesitan comunicarse entre sí para realizar un trabajo. Es posible que deban coordinarse para proporcionar un servicio, compartir datos o incluso almacenar conjuntos de datos que son demasiado grandes para caber en una sola máquina. Analizaremos las diferentes funciones que pueden desempeñar las computadoras en los sistemas distribuidos y aprenderemos sobre los tipos de información que las computadoras necesitan intercambiar para trabajar juntas.

A continuación, analizaremos la computación concurrente, también conocida como computación paralela. La computación concurrente se produce cuando varios procesadores con una memoria compartida ejecutan un solo programa y todos trabajan juntos en paralelo para realizar el trabajo más rápido. La concurrencia presenta nuevos desafíos, por lo que desarrollaremos nuevas técnicas para gestionar la complejidad de los programas concurrentes.

Contents

4.2 Computación distribuida **1**
4.2.1 Sistemas cliente/servidor 1
4.2.2 Sistemas peer-to-peer 3
4.2.3 Modularidad 4
4.2.4 Paso de mensajes 5
4.2.5 Mensajes en la World Wide Web 6

\tableofcontents

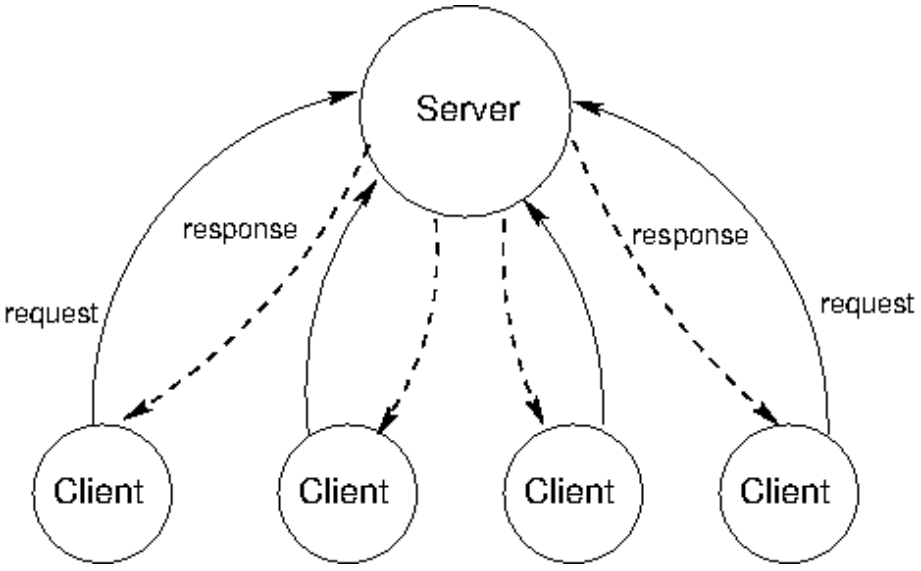
4.2 Computación distribuida

Un sistema distribuido es una red de computadoras autónomas que se comunican entre sí para lograr un objetivo. Las computadoras en un sistema distribuido son independientes y no comparten físicamente memoria ni procesadores. Se comunican entre sí mediante *mensajes* , fragmentos de información transferidos de una computadora a otra a través de una red. Los mensajes pueden comunicar muchas cosas: las computadoras pueden decirle a otras computadoras que ejecuten procedimientos con argumentos específicos, pueden enviar y recibir paquetes de datos o enviar señales que les dicen a otras computadoras que se comporten de cierta manera.

Los ordenadores de un sistema distribuido pueden desempeñar distintas funciones. La función de un ordenador depende del objetivo del sistema y de las propiedades de hardware y software del propio ordenador. Existen dos formas predominantes de organizar los ordenadores en un sistema distribuido. La primera es la arquitectura cliente-servidor y la segunda es la arquitectura punto a punto.

4.2.1 Sistemas cliente/servidor

La arquitectura cliente-servidor es una forma de distribuir un servicio desde una fuente central. Hay un único *servidor* que proporciona un servicio, y múltiples *clientela* que se comunican con el servidor para consumir sus productos. En esta arquitectura, los clientes y los servidores tienen diferentes funciones. La función del servidor es responder a las solicitudes de servicio de los clientes, mientras que la función del cliente es utilizar los datos proporcionados en respuesta para realizar alguna tarea.



El modelo de comunicación cliente-servidor se remonta a la introducción de UNIX en la década de 1970, pero quizás el uso más influyente de este modelo sea la World Wide Web moderna. Un ejemplo de interacción cliente-servidor es la lectura del New York Times en línea. Cuando un cliente de navegación web (como Firefox) contacta con el servidor web de www.nytimes.com, su trabajo es enviar de vuelta el HTML de la página principal del New York Times. Esto podría implicar calcular contenido personalizado en función de la información de la cuenta de usuario enviada por el cliente y obtener los anuncios adecuados. El trabajo del cliente de navegación web es reproducir el código HTML enviado por el servidor. Esto significa mostrar las imágenes, organizar el contenido visualmente, mostrar diferentes colores, fuentes y formas y permitir que los usuarios interactúen con la página web reproducida.

Los conceptos de *cliente* y *servidor* Son abstracciones funcionales potentes. Un servidor es simplemente una unidad que proporciona un servicio, posiblemente a varios clientes simultáneamente, y un cliente es una unidad que consume el servicio. Los clientes no necesitan saber los detalles de cómo se proporciona el servicio, o cómo se almacenan o calculan los datos que reciben, y el servidor no necesita saber cómo se van a utilizar los datos.

En la Web, pensamos que los clientes y los servidores están en máquinas diferentes, pero incluso los sistemas en una sola máquina pueden tener arquitecturas cliente/servidor. Por ejemplo, las señales de los dispositivos de entrada de una computadora deben estar disponibles en general para los programas que se ejecutan en la computadora. Los programas son clientes, que consumen datos de entrada del mouse y del teclado. Los controladores de dispositivos del sistema operativo son los servidores, que toman las señales físicas y las ofrecen como entrada utilizable.

Una desventaja de los sistemas cliente-servidor es que el servidor es un único punto de fallo. Es el único componente con la capacidad de proporcionar el servicio. Puede haber cualquier cantidad de clientes, que son intercambiables y pueden ir y venir según sea necesario. Sin embargo, si el servidor falla, el sistema deja de funcionar. Por lo tanto, la abstracción funcional creada por la arquitectura cliente-servidor también lo hace vulnerable a los fallos.

Otro inconveniente de los sistemas cliente-servidor es que los recursos escasean si hay demasiados clientes. Los clientes aumentan la demanda del sistema sin aportar recursos informáticos. Los sistemas cliente-servidor no pueden reducirse ni crecer con la demanda cambiante.

4.2.2 Sistemas peer-to-peer

El modelo cliente-servidor es adecuado para situaciones orientadas a servicios. Sin embargo, existen otros objetivos computacionales para los cuales una división más equitativa del trabajo es una mejor opción. El término *de igual a igual* se utiliza para describir sistemas distribuidos en los que el trabajo se divide entre todos los componentes del sistema. Todos los ordenadores envían y reciben datos, y todos aportan algo de potencia de procesamiento y memoria. A medida que aumenta el tamaño de un sistema distribuido, aumenta su capacidad de recursos computacionales. En un sistema peer-to-peer, todos los componentes del sistema aportan algo de potencia de procesamiento y memoria a un cómputo distribuido.

División del trabajo entre *todo* Los participantes son la característica distintiva de un sistema peer to peer. Esto significa que los pares deben poder comunicarse entre sí de manera confiable. Para asegurarse de que los mensajes lleguen a sus destinos previstos, los sistemas peer to peer deben tener una estructura de red organizada. Los componentes de estos sistemas cooperan para mantener suficiente información sobre las ubicaciones de otros componentes para enviar mensajes a los destinos previstos.

En algunos sistemas peer-to-peer, la tarea de mantener la salud de la red la asume un conjunto de componentes especializados. Estos sistemas no son sistemas peer-to-peer puros, porque tienen distintos tipos de componentes que cumplen distintas funciones. Los componentes que sustentan una red peer-to-peer actúan como un andamiaje: ayudan a que la red se mantenga conectada, mantienen información sobre las ubicaciones de los distintos equipos y ayudan a los recién llegados a ocupar su lugar dentro de su vecindario.

Las aplicaciones más comunes de los sistemas peer to peer son la transferencia y el almacenamiento de datos. Para la transferencia de datos, cada computadora del sistema contribuye a enviar datos a través de la red. Si la computadora de destino está cerca de una computadora en particular, esa computadora ayuda a enviar los datos. Para el almacenamiento de datos, el conjunto de datos puede ser demasiado grande para caber en una sola computadora, o demasiado valioso para almacenarlo en una sola computadora. Cada computadora almacena una

pequeña parte de los datos, y puede haber múltiples copias de los mismos datos repartidas en diferentes computadoras. Cuando una computadora falla, los datos que estaban en ella se pueden restaurar a partir de otras copias y volver a colocar cuando llegue una de reemplazo.

Skype, el servicio de chat de voz y video, es un ejemplo de una aplicación de transferencia de datos con una arquitectura peer-to-peer. Cuando dos personas en diferentes computadoras mantienen una conversación por Skype, sus comunicaciones se dividen en paquetes de 1 y 0 y se transmiten a través de una red peer-to-peer. Esta red está compuesta por otras personas cuyas computadoras están conectadas a Skype. Cada computadora conoce la ubicación de algunas otras computadoras en su vecindario. Una computadora ayuda a enviar un paquete a su destino pasándolo a un vecino, que lo pasa a otro vecino, y así sucesivamente, hasta que el paquete llega a su destino previsto. Skype no es un sistema peer-to-peer puro. Una red de andamiaje de *supernodos* es responsable de iniciar y cerrar sesión de los usuarios, mantener información sobre las ubicaciones de sus computadoras y modificar la estructura de la red para manejar los usuarios que entran y salen.

4.2.3 Modularidad

Las dos arquitecturas que acabamos de considerar (peer-to-peer y cliente-servidor) están diseñadas para hacer cumplir *modularidad* La modularidad es la idea de que los componentes de un sistema deben ser cajas negras entre sí. No debería importar cómo implementa su comportamiento un componente, siempre que mantenga un *interfaz* :una especificación de qué resultados se obtendrán de las entradas.

En el capítulo 2, nos encontramos con interfaces en el contexto de las funciones de despacho y la programación orientada a objetos. Allí, las interfaces tomaron la forma de especificar los mensajes que los objetos deberían recibir y cómo deberían comportarse en respuesta a ellos. Por ejemplo, para mantener la interfaz "representable como cadenas", un objeto debe poder responder a la`__repr__`y`__str__`mensajes y generar cadenas apropiadas como respuesta. La forma en que se implementa la generación de esas cadenas no forma parte de la interfaz.

En los sistemas distribuidos, debemos considerar el diseño de programas que involucran múltiples computadoras, por lo que ampliamos esta noción de una interfaz de objetos y mensajes a programas completos. Una interfaz especifica las entradas que deben aceptarse y las salidas que deben devolverse en respuesta a las entradas. Las interfaces están en todas partes en el mundo real y, a menudo, las damos por sentado. Un ejemplo conocido son los controles remotos de TV. Puede comprar muchas marcas diferentes de controles remotos para un televisor moderno y todos funcionarán. El único punto en común entre ellos es la interfaz de "control remoto de TV". Un dispositivo electrónico obedece a la interfaz de "control remoto de TV" siempre que envíe las señales correctas a su televisor (la

salida) en respuesta a cuando presiona el botón de encendido, volumen, canal o cualquier otro botón (la entrada).

La modularidad ofrece muchas ventajas a un sistema y es una propiedad de un diseño de sistema bien pensado. En primer lugar, un sistema modular es fácil de entender, lo que facilita su modificación y ampliación. En segundo lugar, si algo va mal con el sistema, solo es necesario sustituir los componentes defectuosos. En tercer lugar, los errores o las averías son fáciles de localizar. Si la salida de un componente no coincide con las especificaciones de su interfaz, aunque las entradas sean correctas, entonces ese componente es la fuente de la avería.

4.2.4 Paso de mensajes

En los sistemas distribuidos, los componentes se comunican entre sí mediante el paso de mensajes. Un mensaje tiene tres partes esenciales: el remitente, el destinatario y el contenido. El remitente debe especificarse para que el destinatario sepa qué componente envió el mensaje y a dónde enviar las respuestas. El destinatario debe especificarse para que cualquier computadora que esté ayudando a enviar el mensaje sepa a dónde dirigirlo. El contenido del mensaje es lo más variable. Dependiendo de la función del sistema general, el contenido puede ser un fragmento de datos, una señal o instrucciones para que la computadora remota evalúe una función con algunos argumentos.

Esta noción de paso de mensajes está estrechamente relacionada con la técnica de paso de mensajes del Capítulo 2, en la que las funciones de despacho o los diccionarios respondían a mensajes con valores de cadena. Dentro de un programa, el emisor y el receptor se identifican mediante las reglas de evaluación. Sin embargo, en un sistema distribuido, el emisor y el receptor deben estar codificados explícitamente en el mensaje. Dentro de un programa, es conveniente utilizar cadenas para controlar el comportamiento de la función de despacho. En un sistema distribuido, es posible que sea necesario enviar mensajes a través de una red y que deban contener muchos tipos diferentes de señales como “datos”, por lo que no siempre se codifican como cadenas. Sin embargo, en ambos casos, los mensajes cumplen la misma función. Diferentes componentes (funciones de despacho o computadoras) los intercambian para lograr un objetivo que requiere la coordinación de múltiples componentes modulares.

En un nivel alto, los contenidos de los mensajes pueden ser estructuras de datos complejas, pero en un nivel bajo, los mensajes son simplemente flujos de 1 y 0 enviados a través de una red. Para que sean utilizables, todos los mensajes enviados a través de una red deben tener un formato coherente._Protocolo de mensajes_.

A**Protocolo de mensajes** es un conjunto de reglas para codificar y decodificar mensajes. Muchos protocolos de mensajes especifican que un mensaje debe cumplir con un formato particular, en el que ciertos bits tienen un significado consistente. Un formato fijo implica reglas de codificación y decodificación fijas para generar y leer ese formato. Todos los componentes del sistema distribuido

deben comprender el protocolo para poder comunicarse entre sí. De esa manera, saben qué parte del mensaje corresponde a qué información.

Los protocolos de mensajes no son programas particulares ni bibliotecas de software, sino reglas que pueden ser aplicadas por una variedad de programas, incluso escritos en lenguajes de programación diferentes. Como resultado, computadoras con sistemas de software muy diferentes pueden participar en el mismo sistema distribuido, simplemente ajustándose a los protocolos de mensajes que gobiernan el sistema.

4.2.5 Mensajes en la World Wide Web

HTTP(abreviatura de Protocolo de transferencia de hipertexto) es el protocolo de mensajes que soporta la World Wide Web. Especifica el formato de los mensajes intercambiados entre un navegador web y un servidor web. Todos los navegadores web utilizan el formato HTTP para solicitar páginas de un servidor web, y todos los servidores web utilizan el formato HTTP para enviar sus respuestas.

Cuando escribes una URL en tu navegador web, dihttp://en.wikipedia.org/wiki/UC_BerkeleyEn realidad, le está indicando a su navegador que debe solicitar la página "wiki/UC_Berkeley" al servidor llamado "en.wikipedia.org" mediante el protocolo "http". El remitente del mensaje es su computadora, el destinatario es en.wikipedia.org y el formato del contenido del mensaje es:

```
GET /wiki/UC_Berkeley HTTP/1.1
```

La primera palabra es el tipo de solicitud, la siguiente palabra es el recurso que se solicita y después el nombre del protocolo (HTTP) y la versión (1.1). (Existen otros tipos de solicitudes, como PUT, POST y HEAD, que los navegadores web también pueden utilizar).

El servidor envía una respuesta. Esta vez, el remitente es en.wikipedia.org, el destinatario es su computadora y el formato del contenido del mensaje es un encabezado, seguido de datos:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2011 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2011 23:11:55 GMT
Content-Type: text/html; charset=UTF-8

... web page content ...
```

En la primera línea, las palabras "200 OK" significan que no hubo errores. Las líneas siguientes del encabezado brindan información sobre el servidor, la fecha y el tipo de contenido que se envía. El encabezado está separado del contenido real de la página web por una línea en blanco.

Si ha escrito una dirección web incorrecta o ha hecho clic en un enlace roto, es posible que haya visto un mensaje de error como este:

```
Error File Not Found
```

Significa que el servidor envió un encabezado HTTP que comenzó así:

```
HTTP/1.1 404 Not Found
```

Un conjunto fijo de códigos de respuesta es una característica común de un protocolo de mensajes. Los diseñadores de protocolos intentan anticipar los mensajes comunes que se enviarán a través del protocolo y asignan códigos fijos para reducir el tamaño de la transmisión y establecer una semántica de mensaje común. En el protocolo HTTP, el código de respuesta 200 indica éxito, mientras que 404 indica un error de que no se encontró un recurso. Una variedad de otroscódigos de respuestaTambién existen en el estándar HTTP 1.1.

HTTP es un formato fijo de comunicación, pero permite transmitir páginas web arbitrarias. Otros protocolos de este tipo en Internet son XMPP, un protocolo popular para mensajes instantáneos, y FTP, un protocolo para descargar y cargar archivos entre el cliente y el servidor.

Contents

5.1 Introducción **1**

\tableofcontents

5.1 Introducción

En este capítulo, continuamos nuestra discusión de aplicaciones del mundo real desarrollando nuevas herramientas para procesar datos secuenciales. En el Capítulo 2, presentamos una interfaz de secuencia, implementada en Python mediante tipos de datos integrados como`tuple`y`list`Las secuencias admitían dos operaciones: consultar su longitud y acceder a un elemento por índice. En el Capítulo 3, desarrollamos una implementación definida por el usuario de la interfaz de secuencia, la`Rlist`Clase para representar listas recursivas. Estos tipos de secuencias demostraron ser eficaces para representar y acceder a una amplia variedad de conjuntos de datos secuenciales.

Sin embargo, la representación de datos secuenciales mediante la abstracción de secuencias tiene dos limitaciones importantes. La primera es que una secuencia de longitud *norte* Por lo general, ocupa una cantidad de memoria proporcional a *norte*. Por lo tanto, cuanto más larga sea una secuencia, más memoria se necesitará para representarla.

La segunda limitación de las secuencias es que sólo pueden representar conjuntos de datos de longitud finita y conocida. Muchas colecciones secuenciales que podemos querer representar no tienen una longitud bien definida, y algunas son incluso infinitas. Dos ejemplos matemáticos de secuencias infinitas son los números enteros positivos y los números de Fibonacci. Los conjuntos de datos secuenciales de longitud ilimitada también aparecen en otros dominios computacionales. Por ejemplo, la secuencia de todas las publicaciones de Twitter se hace más larga con cada segundo y, por lo tanto, no tiene una longitud fija. Del mismo modo, la secuencia de llamadas telefónicas enviadas a través de una torre de telefonía móvil, la secuencia de movimientos del ratón realizados por un usuario de ordenador y la secuencia de mediciones de aceleración de los sensores de un avión se extienden sin límites a medida que el mundo evoluciona.

En este capítulo, presentamos nuevas construcciones para trabajar con datos secuenciales que están diseñadas para acomodar colecciones de longitud desconocida o ilimitada, mientras se utiliza una memoria limitada. También analizamos cómo se pueden usar estas herramientas con una construcción de programación llamada corrutina para crear canales de procesamiento de datos modulares y eficientes.

Contents

5.2 Secuencias implícitas **1**
5.2.1 Iteradores de Python 2
5.2.2 Para declaraciones 4
5.2.3 Generadores y declaraciones de rendimiento 4
5.2.4 Iterables 7
5.2.5 Flujos 8

\tableofcontents

5.2 Secuencias implícitas

La observación central que nos llevará a un procesamiento eficiente de datos secuenciales es que una secuencia puede ser *representado* utilizando construcciones de programación sin que cada elemento sea *almacenado* explícitamente en la memoria de la computadora. Para poner esta idea en práctica, construiremos objetos que proporcionen acceso a todos los elementos de un conjunto de datos secuencial que una aplicación pueda desear, pero sin calcular todos esos elementos de antemano y almacenarlos.

Un ejemplo sencillo de esta idea surge en el`range`tipo de secuencia introducido en el Capítulo 2. A`range`representa una secuencia consecutiva y acotada de números enteros. Sin embargo, no es el caso de que cada elemento de esa secuencia esté representado explícitamente en la memoria. En cambio, cuando se solicita un elemento de un`range`, se calcula. Por lo tanto, podemos representar rangos muy grandes de números enteros sin utilizar grandes bloques de memoria. Solo los puntos finales del rango se almacenan como parte de la`range`objeto y los elementos se calculan sobre la marcha.

```
r = range(10000, 1000000000)
r[45006230]
```

En este ejemplo, no se almacenan todos los 999.990.000 números enteros de este rango cuando se construye la instancia de rango. En su lugar, el objeto de rango agrega el primer elemento 10.000 al índice 45.006.230 para producir el elemento 45.016.230. Calcular valores a pedido, en lugar de recuperarlos de una representación existente, es un ejemplo de *perezoso* computación. La informática es una disciplina que celebra la pereza como una herramienta computacional importante.

Un *iterador* es un objeto que proporciona acceso secuencial a un conjunto de datos secuencial subyacente. Los iteradores son objetos integrados en muchos lenguajes de programación, incluido Python. La abstracción del iterador tiene dos componentes: un mecanismo para recuperar los datos *próximo* Elemento de una serie subyacente de elementos y mecanismo para señalar que se ha llegado al final de la serie y que no quedan más elementos. En los lenguajes de programación con sistemas de objetos integrados, esta abstracción corresponde

normalmente a una interfaz particular que puede implementarse mediante clases. La interfaz de Python para iteradores se describe en la siguiente sección.

La utilidad de los iteradores se deriva del hecho de que la serie de datos subyacente para un iterador puede no estar representada explícitamente en la memoria. Un iterador proporciona un mecanismo para considerar cada uno de los valores de una serie por turno, pero no es necesario almacenar todos esos elementos simultáneamente. En cambio, cuando se solicita el siguiente elemento de un iterador, ese elemento se puede calcular a pedido en lugar de recuperarlo de una fuente de memoria existente.

Los rangos pueden calcular los elementos de una secuencia de forma diferida porque la secuencia representada es uniforme y cualquier elemento es fácil de calcular a partir de los límites inicial y final del rango. Los iteradores permiten la generación diferida de una clase mucho más amplia de conjuntos de datos secuenciales subyacentes, porque no necesitan proporcionar acceso a elementos arbitrarios de la serie subyacente. En cambio, solo deben calcular el siguiente elemento de la serie, en orden, cada vez que se solicita otro elemento. Si bien no es tan flexible como acceder a elementos arbitrarios de una secuencia (llamados *acceso aleatorio*), *acceso secuencial* A menudo, las series de datos secuenciales son suficientes para las aplicaciones de procesamiento de datos.

5.2.1 Iteradores de Python

La interfaz del iterador de Python incluye dos mensajes.`__next__`El mensaje consulta al iterador el siguiente elemento de la serie subyacente que representa. En respuesta a la invocación`__next__`Como método, un iterador puede realizar cálculos arbitrarios para recuperar o calcular el siguiente elemento de una serie subyacente. Llamadas a`__next__`Realizan un cambio de mutación en el iterador: avanzan la posición del iterador. Por lo tanto, múltiples llamadas a`__next__`devolverá elementos secuenciales de una serie subyacente. Python indica que se ha alcanzado el final de una serie subyacente al generar una`StopIteration`excepción durante una llamada a`__next__`.

El`Letters`La clase a continuación itera sobre una serie subyacente de letras de`aad`. La variable miembro`current`almacena la letra actual en la serie y la`__next__`El método devuelve esta letra y la utiliza para calcular un nuevo valor para`current`.

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
```

```
    def __iter__(self):
        return self
```

El`__iter__`message es el segundo mensaje obligatorio de la interfaz del iterador de Python. Simplemente devuelve el iterador; es útil para proporcionar una interfaz común a los iteradores y las secuencias, como se describe en la siguiente sección.

Usando esta clase, podemos acceder a letras en secuencia.

```
letters = Letters()
letters.__next__()
```

```
'a'
```

```
letters.__next__()
```

```
'b'
```

```
letters.__next__()
```

```
'c'
```

```
letters.__next__()
```

```
'd'
```

```
letters.__next__()
```

```
Traceback (most recent call last):
  File "", línea 1, enArchivo "", línea 12, en la siguiente
StopIteration
```

A`Letters`La instancia solo se puede iterar una vez. Una vez que se completa`__next__()`El método plantea una`StopIteration`excepción, continúa haciéndolo desde entonces. No hay forma de restablecerlo; se debe crear una nueva instancia.

Los iteradores también nos permiten representar series infinitas implementando una`__next__`método que nunca genera una`StopIteration`excepción. Por ejemplo, la`Positives`La clase siguiente itera sobre la serie infinita de números enteros positivos.

```
class Positives(object):
    def __init__(self):
        self.current = 0;
    def __next__(self):
        result = self.current
        self.current += 1
        return result
    def __iter__(self):
        return self
```

5.2.2 Para declaraciones

En Python, las secuencias pueden exponerse a la iteración implementando la`__iter__`mensaje. Si un objeto representa datos secuenciales, puede servir como un *iterable* objeto en un`for`declaración devolviendo un objeto iterador en respuesta a la`__iter__`mensaje. Este iterador está destinado a tener un`__next__()`método que devuelve cada elemento de la secuencia a su vez, generando eventualmente una`StopIteration`excepción cuando se alcanza el final de la secuencia.

```
counts = [1, 2, 3]
for item in counts:
    print(item)
```

```
2
3
```

En el ejemplo anterior, el`counts`La lista devuelve un iterador en respuesta a una llamada a su`__iter__()`método. El`for`La declaración luego llama a ese iterador.`__next__()`método repetidamente y asigna el valor devuelto a`item`cada vez. Este proceso continúa hasta que el iterador lanza un`StopIteration`excepción, en cuyo caso el`for`La declaración concluye.

Con nuestro conocimiento de iteradores, podemos implementar la regla de evaluación de un`for`declaración en términos de`while`, asignación, y`try`declaraciones.

```
i = counts.__iter__()
try:
    while True:
        item = i.__next__()
        print(item)
except StopIteration:
    pass
```

```
2
3
```

Arriba, el iterador regresó al invocar el`__iter__`método de`counts`Está ligado a un nombre`i`para que se pueda consultar cada elemento por turno. La cláusula de manejo para el`StopIteration`La excepción no hace nada, pero manejar la excepción proporciona un mecanismo de control para salir de la`while`bucle.

5.2.3 Generadores y declaraciones de rendimiento

El`Letters`y`Positives`Los objetos anteriores requieren que introduzcamos un nuevo campo`self.current`en nuestro objeto para realizar un seguimiento del progreso a través de la secuencia. Con secuencias simples como las que se muestran arriba, esto se puede hacer fácilmente. Sin embargo, con secuencias complejas, puede resultar bastante difícil para el`__next__()`Función para guardar

su lugar en el cálculo. Los generadores nos permiten definir iteraciones más complicadas aprovechando las características del intérprete de Python.

A *generador* es un iterador devuelto por una clase especial de función llamada *función generadora* Las funciones generadoras se distinguen de las funciones regulares en que, en lugar de contener`return`En sus declaraciones, las utilizan`yield`Declaración para devolver elementos de una serie.

Los generadores no utilizan los atributos de un objeto para seguir su progreso a lo largo de una serie. En cambio, controlan la ejecución de la función del generador, que se ejecuta hasta el siguiente`yield`La declaración se ejecuta cada vez que el generador`__next__`Se invoca el método.`Letters`El iterador se puede implementar de forma mucho más compacta utilizando una función generadora.

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
a
b
c
d
```

Aunque nunca lo definimos explícitamente`__iter__()`o`__next__()`métodos, Python entiende que cuando usamos el`yield`En esta declaración, estamos definiendo una función generadora. Cuando se la llama, una función generadora no devuelve un valor determinado, sino un`generator`(que es un tipo de iterador) que puede devolver los valores obtenidos. Un objeto generador tiene`__iter__`y`__next__`métodos y cada llamada a`__next__`continúa la eje-

cución de la función del generador desde donde la dejó anteriormente hasta otra`yield`Se ejecuta la declaración.

La primera vez`__next__`se llama, el programa ejecuta sentencias del cuerpo de la`letters_generator`función hasta que se encuentra con el`yield`declaración. Luego, hace una pausa y devuelve el valor de`current`. `yield`Las declaraciones no destruyen el entorno recién creado, sino que lo preservan para el futuro.`__next__`se vuelve a llamar, la ejecución se reanuda donde se dejó. Los valores de`current`y de cualquier otro nombre vinculado en el ámbito de aplicación`letters_generator`se conservan en llamadas posteriores a`__next__`.

Podemos recorrer el generador llamando manualmente`____next__()`:

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
'a'
```

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
```

```
    def __iter__(self):
        return self
```

```
'b'
```

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
'c'
```

Número de serie 16

```
'd'
```

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
Traceback (most recent call last):
  File "", línea 1, enDetener la iteración
```

El generador no comienza a ejecutar ninguna de las declaraciones del cuerpo de su función generadora hasta la primera vez`__next__()`se llama.

5.2.4 Iterables

En Python, los iteradores solo realizan una única pasada sobre los elementos de una serie subyacente. Después de esa pasada, el iterador continuará generando una`StopIteration`excepción cuando`__next__()`se llama. Muchas aplicaciones requieren iterar sobre los elementos varias veces. Por ejemplo, tenemos que iterar sobre una lista muchas veces para enumerar todos los pares de elementos.

Número de serie 18

```
class Letters(object):
    def __init__(self):
        self.current = 'a'
    def __next__(self):
        if self.current > 'd':
            raise StopIteration
        result = self.current
        self.current = chr(ord(result)+1)
        return result
    def __iter__(self):
        return self
```

```
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Las secuencias no son iteradores en sí mismas, sino que *iterable* objetos. La interfaz iterable en Python consta de un solo mensaje,`__iter__`, que devuelve un iterador. Los tipos de secuencia integrados en Python devuelven nuevas instancias de iteradores cuando sus`__iter__`Se invocan métodos. Si un objeto iterable devuelve una nueva instancia de un iterador cada vez`__iter__`se llama, luego se puede iterar varias veces.

Se pueden definir nuevas clases iterables implementando la interfaz iterable. Por ejemplo,*iterable* `LetterIterable`La clase a continuación devuelve un nuevo iterador sobre las letras cada vez`__iter__`se invoca.

```
letters = Letters()
letters.__next__()
```

El`__iter__`El método es una función generadora; devuelve un objeto generador que produce las letras`'a'`a través de`'d'`.

A`Letters`El objeto iterador se "agota" después de una sola iteración, mientras que el`LetterIterable`El objeto se puede iterar varias veces. Como resultado, un`LetterIterable`La instancia puede servir como argumento para`all_pairs`.

```
letters = Letters()
letters.__next__()
```

```
('a', 'a')
```

5.2.5 Flujos

Arroyos ofrecen una forma final de representar la implicidad de datos secuenciales. Un flujo es una lista recursiva calculada de forma diferida. Al igual que`Rlist`Clase del Capítulo 3, a`Stream`instancia responde a las solicitudes de su`first`elemento y el`rest`del arroyo. Como un`Rlist`, el`rest`de un`Stream`es en sí mismo un`Stream`. A diferencia de un`Rlist`, el`rest`de un flujo solo se calcula cuando se busca, en lugar de almacenarse de antemano. Es decir, el`rest`de una secuencia se calcula de forma perezosa.

Para lograr esta evaluación diferida, un flujo almacena una función que calcula el resto del flujo. Siempre que se llama a esta función, su valor devuelto se almacena en caché como parte del flujo en un atributo llamado`_rest`, nombrado con un guión bajo para indicar que no se debe acceder a él directamente. El atributo accesible`rest`es un método de propiedad que devuelve el resto del flujo, computándolo si es necesario. Con este diseño, un flujo almacena *Cómo calcular* el resto de la transmisión, en lugar de almacenarla siempre explícitamente.

```
letters = Letters()
letters.__next__()
```

```
letters = Letters()
letters.__next__()
```

Una lista recursiva se define mediante una expresión anidada. Por ejemplo, podemos crear una`Rlist`que representa los elementos`1`entonces`5`como sigue:

```
letters = Letters()
letters.__next__()
```

De la misma manera, podemos crear una`Stream`representando la misma serie.`Stream`En realidad no calcula el segundo elemento`5`hasta que se solicite el resto de la transmisión.

```
letters = Letters()
letters.__next__()
```

Aquí,`1`es el primer elemento de la secuencia, y el`lambda`La expresión que sigue devuelve una función para calcular el resto del flujo. El segundo elemento del flujo calculado es una función que devuelve un flujo vacío.

Acceder a los elementos de una lista recursiva`r`y transmitir`s`Proceda de manera similar. Sin embargo, mientras`5`se almacena dentro`r`, se calcula a demanda de`s`mediante adición la primera vez que se solicita.

```
letters = Letters()
letters.__next__()
```

```
1
```

```
letters = Letters()
letters.__next__()
```

```
1
```

```
letters = Letters()
letters.__next__()
```

```
5
```

```
letters = Letters()
letters.__next__()
```

```
5
```

```
letters.__next__()
```

```
Rlist(5)
```

```
letters.__next__()
```

```
Stream(5, )
```

Mientras que el`rest`der`es` una lista recursiva de un elemento, la`rest`de`s`incluye una función para calcular el resto; es posible que aún no se haya descubierto que devolverá el flujo vacío.

Cuando un`Stream`Se construye la instancia, el campo`self._computed`es`False`, lo que significa que el`_rest`del`Stream`aún no se ha calculado. Cuando el`rest`El atributo se solicita a través de una expresión de punto, el`rest`Se invoca el método, que activa el cálculo con`self._rest = self.compute_rest`Debido al mecanismo de almacenamiento en caché dentro de un`Stream`, el`compute_rest`La función solo se llama una vez.

Las propiedades esenciales de un`compute_rest`La función es que no toma argumentos y devuelve un`Stream`.

La evaluación diferida nos brinda la capacidad de representar conjuntos de datos secuenciales infinitos mediante secuencias. Por ejemplo, podemos representar números enteros crecientes, comenzando en cualquier`first`valor.

```
letters.__next__()
```

```
letters.__next__()
```

```
Stream(1, )
```

```
letters.__next__()
```

```
1
```

Cuando`make_integer_stream`se llama por primera vez, devuelve un flujo cuyo`first`es el primer entero en la secuencia (`1`por defecto). Sin embargo,`make_integer_stream`En realidad es recursivo porque esta secuencia`compute_rest`llamadas`make_integer_stream`De nuevo, con un argumento incrementado. Esto hace`make_integer_stream`recursivo, pero también perezoso.

```
letters.__next__()
```

```
1
```

```
letters.__next__()
```

```
2
```

```
letters.__next__()
```

```
Stream(3, )
```

Las llamadas recursivas solo se realizan a`make_integer_stream`siempre que el`rest`Se solicita una secuencia de números enteros.

Las mismas funciones de orden superior que manipulan secuencias…`map`y`filter`- también se aplican a los flujos, aunque sus implementaciones deben cambiar para aplicar sus funciones de argumento de manera diferida. La función`map_stream`asigna una función a un flujo, lo que produce un nuevo flujo. La función definida localmente`compute_rest`La función garantiza que la función se asignará al resto de la secuencia cada vez que se calcule el resto.

```
letters.__next__()
```

Se puede filtrar una secuencia definiendo un`compute_rest`Función que aplica la función de filtro al resto del flujo. Si la función de filtro rechaza el primer elemento del flujo, el resto se calcula inmediatamente. Porque`filter_stream`es recursivo, el resto puede calcularse varias veces hasta obtener un resultado válido.`first`Se encontró el elemento.

```
letters.__next__()
```

El`map_stream`y`filter_stream`Las funciones exhiben un patrón común en el procesamiento de flujo: una definida localmente`compute_rest`La función aplica recursivamente una función de procesamiento al resto del flujo cada vez que se calcula el resto.

Para inspeccionar el contenido de un flujo, podemos truncarlo a una longitud finita y convertirlo en un código Python.`list`.

```
letters.__next__()
```

```
letters.__next__()
```

Estas funciones de conveniencia nos permiten verificar nuestra`map_stream`Implementación con un ejemplo simple que eleva al cuadrado los números enteros de`3`a`7`.

```
letters.__next__()
```

```
Stream(3, )
```

```
letters.__next__()
```

```
Stream(9, )
```

```
letters.__next__()
```

```
[9, 16, 25, 36, 49]
```

Podemos utilizar nuestro`filter_stream`Función para definir un flujo de números primos utilizando la criba de Eratóstenes, que filtra un flujo de números enteros para eliminar todos los números que son múltiplos de su primer elemento. Al filtrar sucesivamente con cada primo, se eliminan del flujo todos los números compuestos.

```
letters.__next__()
```

Al truncar el`primes`corriente, podemos enumerar cualquier prefijo de los números primos.

```
letters.__next__()
```

```
[2, 3, 5, 7, 11, 13, 17]
```

Los flujos se diferencian de los iteradores en que se pueden pasar a funciones puras varias veces y arrojar el mismo resultado cada vez. El flujo de números primos no se "agota" al convertirlo en una lista. Es decir, el`first`elemento de`p1`Sigue siendo`2`después de convertir el prefijo del flujo en una lista.

```
letters.__next__()
```

```
2
```

Así como las listas recursivas proporcionan una implementación simple de la abstracción de secuencia, los flujos proporcionan una estructura de datos recursiva, funcional y simple que implementa la evaluación perezosa mediante el uso de funciones de orden superior.