9
Testes if

Este capítulo apresenta a instrução `if` do Python – a principal instrução usada para selecionar ações alternativas com base em resultados de testes. Como esta é nossa primeira exposição às *instruções compostas* – instruções que incorporam outras instruções –, também exploraremos aqui os conceitos gerais existentes por trás do modelo da sintaxe de instrução do Python. Além disso, como a instrução `if` introduz a noção de testes, também usaremos este capítulo para estudarmos os conceitos de testes de verdade e expressões booleanas em geral.

INSTRUÇÕES if

Em termos simples, a instrução `if` do Python seleciona ações para executar. Ela é a principal ferramenta de linguagem e representa grande parte da *lógica* que um programa em Python possui. Ela também é nossa primeira instrução composta; assim como todas as instruções compostas do Python, `if` pode conter outras instruções, incluindo outras instruções `if`. Na verdade, o Python permite combinar instruções em um programa tanto seqüencialmente (para que sejam executadas uma após a outra) como arbitrariamente aninhadas (para que sejam executadas apenas sob certas condições).

Formato geral

A instrução `if` do Python é típica da maioria das linguagens procedurais. Ela assume a forma de um teste `if`, seguido de um ou mais testes `elif` opcionais (significando "`else if`"), e termina com um bloco `else` opcional. Cada teste e a cláusula `else` têm um bloco associado de instruções aninhadas, endentadas sob uma linha de cabeçalho. Quando a instrução é executada, o Python executa o bloco de código associado que possuir o primeiro teste avaliado como verdadeiro ou o bloco `else`, caso todos os testes sejam falsos. A forma geral de uma instrução `if` é a seguinte:

```
if <teste1>:                # teste if
<instruções1>               # Bloco associado
elif <teste2>:              # Instruções elif opcionais
<instruções2>
else                        # Instrução else opcional
< instruções3>
```

Exemplos

Vamos ver alguns exemplos simples da instrução `if` em funcionamento. Todas as partes são opcionais, exceto o teste `if` inicial e suas instruções associadas. No caso mais simples, as outras partes são omitidas:

```
>>> if 1:
...      print 'true'
...
true
```

Observe como o prompt muda para `"..."`, para continuação de linhas na interface básica usada aqui (no IDLE, em vez disso, você simplesmente descerá para uma linha endentada – pressione Backspace para voltar para cima). Uma linha em branco finaliza e executa a instrução inteira. Lembre-se de que `1` é o valor verdadeiro booleano; portanto, o teste dessa instrução é sempre bem-sucedido; para tratar de um resultado falso, codifique a cláusula `else`:

```
>>> if not 1:
...     print 'true'
... else:
...     print 'false'
...
false
```

Agora, aqui está um exemplo do tipo mais complexo de instrução `if` – com todas as suas partes opcionais presentes:

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print "how's jessica?"
... elif x == 'bugs':
...     print "what's up doc?"
... else:
...     print 'Run away! Run away!'
...
Run away! Run away!
```

Essa instrução de várias linhas vai da linha `if` até o bloco `else`. Quando executada, o Python executa as instruções aninhadas sob o primeiro teste que seja verdadeiro ou a parte `else`, caso todos os testes sejam falsos (neste exemplo, eles são). Na prática, as partes `elif` e `else` podem ser omitidas e pode haver mais de uma instrução aninhada em cada seção. Além disso, as palavras `if`, `elif` e `else` são associadas pelo fato de serem alinhadas verticalmente, com a mesma endentação.

Desvio de vários caminhos

Se você já usou linguagens como C ou Pascal, pode estar interessado em saber que não existe nenhuma instrução "switch" ou "case" no Python, que seleciona uma ação com base no valor de uma variável. Em vez disso, o *desvio de vários caminhos* é codificado como uma série de testes `if/elif`, como foi feito no exemplo anterior, ou pela indexação de dicionários ou pesquisa em listas.

Como os dicionários e as listas podem ser construídos em tempo de execução, às vezes eles são mais flexíveis do que a lógica incorporada na instrução `if`:

```
>>> choice = 'ham'
>>> print {'spam': 1.25,          # Uma instrução 'switch' baseada em
                                    dicionário
```

```
...         'ham': 1.99,    # Usa has_key() ou get() por padrão
...         'eggs': 0.99,
...         'bacon': 1.10}[choice]
1.99
```

Embora normalmente leve alguns instantes para entender isso na primeira vez que você vê, esse dicionário é um desvio de vários caminhos – a indexação na chave `choice` desvia para um valor de um conjunto de valores, exatamente como uma instrução "switch" na linguagem C. Uma instrução `if` quase equivalente e mais longa do Python poderia ser a seguinte:

```
>>> if choice == 'spam':
...     print 1.25
... elif choice == 'ham':
...     print 1.99
... elif choice == 'eggs':
...     print 0.99
... elif choice == 'bacon':
...     print 1.10
... else:
...     print 'Bad choice'
...
1.99
```

Observe aqui a cláusula `else` na instrução `if` para tratar do caso *padrão*, quando nenhuma chave corresponde. Conforme vimos no Capítulo 6, os padrões de dicionários podem ser codificados com testes `has_key`, chamadas de método `get` ou captura de exceção. Todas as mesmas técnicas podem ser usadas aqui para codificar uma ação padrão em um desvio de vários caminhos baseado em dicionário. Aqui está o esquema do método `get` em funcionamento com padrões:

```
>>> branch = {'spam': 1.25,
...           'ham': 1.99,
...           'eggs': 0.99}
>>> print branch.get('spam', 'Bad choice')
1.25
>>> print branch.get('bacon', 'Bad choice')
Bad choice
```

Os dicionários são bons para associar valores a chaves, mas e quanto às ações mais complicadas que você pode codificar em instruções `if`? Na Parte IV, você vai aprender que os dicionários também podem conter *funções* para representar ações de desvio mais complexas, e implementam tabelas de salto gerais. Tais funções que aparecem como valores de dicionário, freqüentemente são escritas como `lambdas` e são chamadas pela adição de parênteses para ativar suas ações.

REGRAS DE SINTAXE DO PYTHON

Em geral, o Python tem uma sintaxe simples, baseada em instruções. Mas existem algumas propriedades que você precisa conhecer:

As instruções são executadas uma após a outra, até que você diga o contrário. Normalmente, o Python executa as instruções de um arquivo ou bloco aninhado, da primeira até a última, mas instruções como `if` (e, conforme você verá, os loops) fazem o interpretador saltar em seu código. Como o caminho do Python por um programa é chamado de fluxo de controle, comandos que o afetam (como a instrução `if`) são chamados de instruções de controle de fluxo.

Os limites de blocos e de instruções são detectados automaticamente. Não existem chaves ou delimitadores tipo "begin/end" em torno de blocos de código. Em vez disso, o Python usa a endentação de instruções sob um cabeçalho para agrupar as instruções em um bloco aninhado. Analogamente, as instruções do Python normalmente não terminam com um ponto-e-vírgula; em vez disso, o fim de uma linha marca o final das instruções escritas nessa linha.

Instruções compostas = cabeçalho. ":", instruções endentadas. No Python, todas as instruções compostas seguem o mesmo padrão: uma linha de cabeçalho terminada com dois-pontos, seguida de uma ou mais instruções aninhadas, normalmente endentadas sob o cabeçalho. As instruções endentadas são chamadas de *bloco* (ou, às vezes, de conjunto). Na instrução `if`, as cláusulas `elif` e `else` fazem parte dela, mas são linhas de cabeçalho com seus próprios blocos aninhados.

Normalmente, linhas em branco, espaços e comentários são ignorados. As linhas em branco são ignoradas em arquivos (mas não no prompt interativo). Os espaços dentro de instruções e expressões quase sempre são ignorados (exceto em literais de string e em endentação). Os comentários são sempre ignorados: eles começam com o caractere # (não dentro de uma literal de string) e se estendem até o fim da linha corrente.

As strings de documentação são ignoradas, mas são salvas e exibidas por ferramentas. O Python suporta uma forma de comentário adicional chamada string de documentação (abreviadamente, *docstring*), a qual, ao contrário dos comentários `#`, é mantida. As strings de documentação são simplesmente strings que aparecem no início de arquivos de programa e de algumas instruções, e são automaticamente associadas a objetos. Seu contéudo é ignorado pelo Python, mas elas são anexadas automaticamente aos objetos em tempo de execução, e podem ser exibidas com ferramentas de documentação. As `docstrings` fazem parte da documentação mais ampla do Python e serão abordadas no final da Parte III.

Conforme você viu, não existem declarações de tipo variável no Python. Esse fato sozinho contribui para uma sintaxe de linguagem muito mais simples do que você pode estar acostumado. Mas para a maioria dos usuários iniciantes, a falta de chaves e pontos-e-vírgulas para marcar blocos e instruções parece ser a característica sintática mais insólita do Python; portanto, vamos explorar o que isso significa com mais detalhes aqui.

Delimitadores de bloco

O Python detecta os limites automaticamente pela *endentação* da linha – o espaço vazio à esquerda de seu código. Todas as instruções endentadas com a mesma distância à direita pertencem ao mesmo bloco de código. Em outras palavras, as instruções dentro de um bloco são alinhadas verticalmente. O bloco termina em uma linha menos endentada ou no final do arquivo, e os blocos mais profundamente aninhados são simplesmente mais endentados à direita do que as instruções que estão no bloco que os engloba.

Por exemplo, a Figura 9-1 demonstra a estrutura de bloco do código a seguir:

```
x = 1
if x:
    y =2
    if y:
        print 'bloco2'
    print 'bloco1'
print 'bloco0'
```

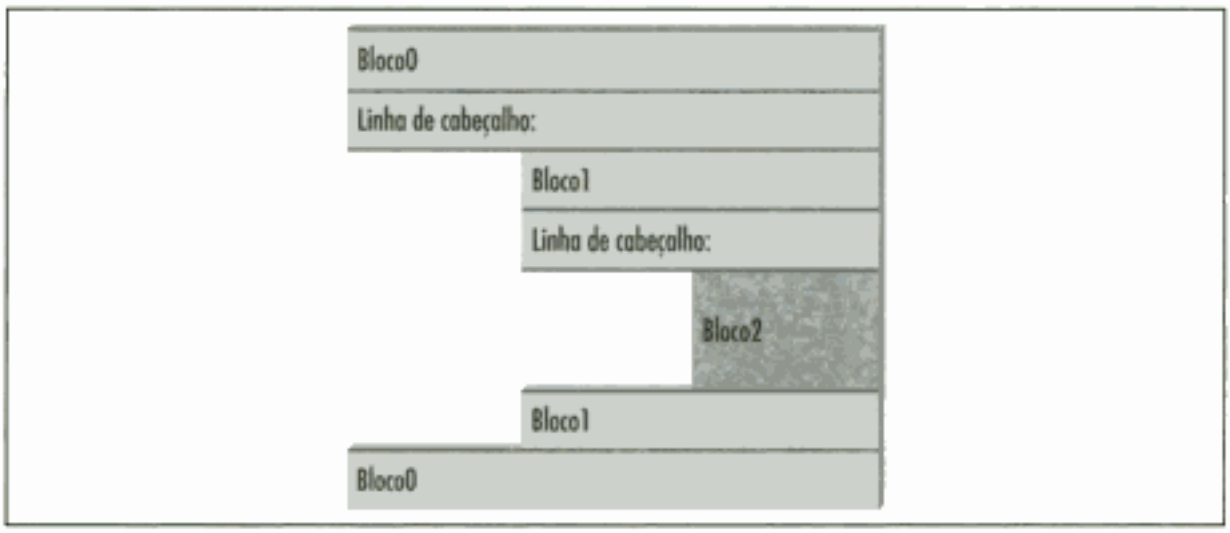

Figura 9-1 Blocos de código aninhados.

Esse código contém três blocos: o primeiro (o nível superior do arquivo) não é endentado, o segundo (dentro da instrução `if` mais externa) é endentado por quatro espaços e o terceiro (a instrução `print` sob a instrução `if` aninhada) é endentado por oito espaços.

Em geral, o código de nível superior (não aninhado) deve começar na linha 1. Os blocos aninhados podem começar em qualquer coluna; a endentação pode consistir em qualquer número de espaços e tabulações, contanto que seja a mesma para todas as instruções em determinado bloco. Isto é, o Python não se preocupa com a maneira como você endenta seu código; ele só se preocupa com o fato de que isso seja feito consistentemente. Tecnicamente, as tabulações valem por espaços suficientes para mover o número da coluna corrente para cima, em um múltiplo de 8, mas normalmente não é uma boa idéia misturar tabulações e espaços dentro de um bloco – use um ou outro.

A endentação à esquerda de seu código é o único lugar importante no Python onde o espaço em branco tem significado; na maioria dos outros contextos, o espaço pode ser escrito ou não. Entretanto, a endentação faz parte da sintaxe do Python e não é apenas uma sugestão estilística: todas as instruções dentro de determinado bloco devem ter a mesma endentação, senão o Python relatará um erro de sintaxe. Isso é assim de propósito – como você não precisa marcar explicitamente o início e o fim de um bloco de código aninhado, isso elimina parte da confusão sintática encontrada em outras linguagens.

Esse modelo de sintaxe também impõe a consistência da endentação, um componente fundamental da legibilidade em linguagens de programação estruturadas, como o Python. Às vezes, a sintaxe do Python é chamada de "o que você vê é o que obtém" das linguagens – a endentação de código informa aos leitores, inequivocamente, o que está associado com o que. A aparência consistente do Python torna o código mais fácil de manter.

Um código endentado consistentemente sempre satisfaz as regras do Python. Além disso, a maioria dos editores de textos (incluindo o IDLE) torna fácil seguir o modelo de endentação do Python, endentando o código automaticamente, enquanto você digita.

Delimitadores de instrução

Normalmente, as instruções terminam no fim da linha em que aparecem. Isso abrange a ampla maioria das instruções do Python que você escreverá. Contudo, quando as instruções são longas demais para caberem em uma única linha, algumas regras especiais podem ser usadas para fazer com que elas se estendam por várias linhas de continuação:

As instruções podem abranger várias linhas se você estiver continuando um par sintático aberto. Para instruções que são longas demais para caber em uma única linha, o Python permite que você continue a digitar a instrução na linha seguinte, caso esteja codificando algo incluído em pares `()`, `{}` ou `[]`. Por exemplo, expressões entre parênteses e literais de dicionário e lista podem abranger qualquer número de linhas. Sua instrução não termina até a linha na qual você digita a parte referente ao fechamento do par `()`, `}` ou `]`). A continuação de linhas pode começar em qualquer nível de endentação.

As instruções podem abranger várias linhas se terminarem com uma barra invertida. Este é um recurso um tanto desatualizado, mas se uma instrução precisa ocupar várias linhas, você também pode adicionar uma barra invertida (`\`) no final da linha anterior, para indicar que está continuando na linha seguinte. Mas como você também pode continuar adicionando parênteses em torno de construções longas, as barras invertidas quase nunca são necessárias.

Outras regras. Literais de string muito longas podem abranger linhas arbitrariamente. Na verdade, os blocos de string com aspas triplas, que conhecemos no Capítulo 5, são projetados para isso. Embora seja incomum, você também pode terminar instruções com um ponto-e-vírgula – às vezes isso é usado para espremer mais de uma instrução simples em uma única linha. Finalmente, comentários e linhas em branco podem aparecer em qualquer lugar.

Alguns casos especiais

Aqui temos um exemplo de como fica uma continuação de linha usando a regra dos pares abertos; podemos ocupar construções delimitadas com qualquer número de linhas:

```
L = ["Good",
     "Bad",
     "Ugly"]                    # Pares abertos podem abranger várias linhas.
```

Isso também funciona para tudo que estiver entre parênteses: expressões, argumentos de função, cabeçalhos de função (veja o Capítulo 12) etc. Se quiser usar barras invertidas para continuar, você pode, mas normalmente isso não é necessário:

```
if a == b and c == d and \
   d == e and f == g:
   print 'olde'                  # Barras invertidas permitem continuações.
```

Como qualquer expressão pode ser colocada entre parênteses, normalmente você pode simplesmente envolver algo entre parênteses sempre que precisar, para abranger várias linhas:

```
if (a == b and c == d and
    d == e and f == g):
    print 'new'                  # Mas parênteses normalmente também permitem.
```

Como um caso especial, o Python permite que você escreva mais de uma instrução não-composta (isto é, instruções sem outras aninhadas) na mesma linha, separadas por pontos-e-vírgulas. Alguns codificadores utilizam essa forma para economizar espaço em arquivos de programa, mas normalmente o código fica mais legível se você coloca uma instrução por linha na maior parte de seu trabalho:

```
x = 1; y = 2; print x                  # Mais de uma instrução simples
```

Finalmente, o Python permite que você mova o corpo de uma instrução composta para a linha de cabeçalho, desde que o corpo seja apenas uma instrução simples (não-composta). Você vai ver isso usado com mais freqüência em instruções `if` simples, com um único teste e uma única ação:

```
if 1: print 'hello'          # Instrução simples na linha de cabeçalho
```

Você pode combinar alguns desses casos especiais para escrever código difícil de ler, mas não recomendamos isso. Como regra geral, tente manter cada instrução em sua própria linha e endente tudo, exceto os blocos mais simples. Com seis meses de trabalho, você ficará feliz de ter feito isso.

TESTES DE VERDADE

Apresentamos as noções de comparação, igualdade e valores de verdade no Capítulo 7. Como a instrução `if` é a primeira que realmente utiliza resultados de teste, expandiremos algumas dessas idéias aqui. Em particular, os operadores booleanos do Python são um pouco diferentes de seus correlatos em linguagens como C. No Python:

- Verdadeiro significa qualquer número diferente de zero ou objeto não-vazio.
- Falso significa não verdadeiro: um número zero, um objeto vazio ou `None`.
- As comparações e testes de igualdade são aplicados recursivamente nas estruturas de dados.
- As comparações e testes de igualdade retornam `1` ou `0` (verdadeiro ou falso).
- Os operadores booleanos `and` e `or` retornam um objeto operando verdadeiro ou falso.

Em resumo, os operadores booleanos são usados para combinar os resultados de outros testes. Existem três operações de expressão booleanas no Python:

`X and Y`
: É verdadeiro se `X` e `Y` são verdadeiros

`X or Y`
: É verdadeiro se `X` ou `Y` é verdadeiro

`not X`
: É verdadeiro se `X` é falso (a expressão retorna `1` ou `0`)

Aqui, `X` e `Y` podem ser qualquer valor de verdade ou uma expressão que retorna um valor de verdade (por exemplo, um teste de igualdade, comparação de intervalo etc.). Os operadores booleanos são digitados como palavras no Python (em vez de `&&`, `||` e `!` da linguagem C). No Python, os operadores booleanos `and` e `or` retornam um *objeto* verdadeiro ou falso. Vamos ver alguns exemplos para saber como isso funciona:

```
>>> 2 < 3, 3 < 2                    # Menor que: retorna 1 ou 0
(1, 0)
```

Comparações de grandeza como essas retornam um valor inteiro `1` ou `0` como resultado do valor de verdade. Mas os operadores `and` e `or`, em vez disso, sempre retornam um objeto. Para testes do operador `or`, o Python avalia os objetos operando da esquerda para a direita e retorna o primeiro que for verdadeiro. Além disso, o Python pára no primeiro operando verdadeiro que encontra; normalmente, isso é chamado de *avaliação de curto-circuito*, pois a determinação de um resultado causa um curto-circuito (termina) no restante da expressão:

```
>>> 2 or 3, 3 or 2          # Retorna o operando da esquerda se for
                              verdadeiro.
(2, 3)                      # Senão, retorna o operando da direita
                              (verdadeiro ou falso).
>>> [] or 3
3
>>> [] or {}
{}
```

Na primeira linha anterior, os dois operandos são verdadeiros (`2,3`); portanto, o Python sempre pára e retorna o que está à esquerda. Nos outros dois testes, o operando da esquerda é falso; portanto, o Python simplesmente avalia e retorna o objeto da direita (que terá um valor verdadeiro ou falso, se for testado). Além disso, as operações de `and` param assim que o resultado for conhecido; neste caso, o Python avalia os operandos da esquerda para a direita e pára no primeiro objeto falso:

```
>>> 2 and 3, 3 and 2        # Retorna o operando da esquerda se for falso.
(3, 2)                      # Senão, retorna o operando da direita
                              (verdadeiro ou falso).
>>> [] and {}
[]
>>> 3 and []
[]
```

Aqui, os dois operandos são verdadeiros na primeira linha; portanto, o Python avalia os dois lados e retorna o objeto da direita. No segundo teste, o operando da esquerda é falso (`[]`); portanto, o Python pára e o retorna como resultado do teste. No último teste, o lado esquerdo é verdadeiro (`3`); portanto, o Python avalia e retorna o objeto da direita (que por acaso é falso `[]`).

O resultado final de tudo isso é o mesmo em C e na maioria das outras linguagens – você recebe um valor que é logicamente verdadeiro ou falso, se testado em uma instrução `if` ou `while`. Entretanto, no Python, os valores booleanos retornam o objeto da esquerda ou da direita e não um flag inteiro.

Uma nota final: conforme descrito no Capítulo 7, o Python 2.3 inclui um novo tipo booleano, chamado `bool`, que internamente é uma subclasse do tipo inteiro `int`, com valores `True` e `False`. Esses dois casos são, na verdade, apenas versões personalizadas dos valores inteiros `1` e `0`, os quais produzem as palavras `True` e `False` quando impressos ou convertidos em strings de alguma outra forma. A única vez em que você geralmente notará essa mudança será quando ver saídas booleanas impressas como `True` e `False`. Mais informações sobre subclasses de tipo aparecem no Capítulo 23.

Por que isto é relevante: valores booleanos

Uma maneira comum de usar o comportamento exclusivo dos operadores booleanos do Python é na seleção de um conjunto de objetos com um operador `or`. Uma instrução:

```
X = A or B or C or None
```

configura `X` com o primeiro objeto não-vazio (ou seja, verdadeiro) entre `A`, `B` e `C`, ou `None`, se todos forem vazios. Isso se mostra um paradigma de desenvolvimento muito comum no Python: para selecionar um objeto não-vazio em um conjunto de tamanho fixo, basta enfileirá-los em uma expressão `or`.

Também é importante entender a avaliação de curto-circuito, pois as expressões à direita de um operador booleano poderiam chamar funções que fazem muito trabalho ou têm efeitos colaterais que não aconteceriam se a regra do curto-circuito surtisse efeito:

```
if f1() or f2(): ...
```

Aqui, se `f1` retornar um valor verdadeiro (ou não-vazio), o Python nunca executará `f2`. Para garantir que as duas funções sejam executadas, chame-as antes do operador `or`:

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

Você verá outra aplicação desse comportamento no Capítulo 14: por causa da maneira como os valores booleanos funcionam, a expressão `((A and B) or C)` pode ser usada para simular uma instrução `if/else` – ou quase. Observe também que, como todos os objetos são inerentemente verdadeiros ou falsos, no Python é comum e mais fácil testar um objeto diretamente `(if X:)`, em vez de compará-lo com um valor vazio `(if X!= '':)`. Para uma string, os dois testes são equivalentes.

10

Loops while e for

Neste capítulo, conheceremos as duas principais construções de *loops* do Python – instruções que reproduzem uma ação repetidamente. A primeira delas, o loop `while`, fornece uma maneira de desenvolver loops gerais. A segunda, o loop `for`, é projetada para percorrer os itens de um objeto de seqüência e executar um bloco de código para cada item.

Existem outros tipos de operações de loop no Python, mas as duas instruções abordadas aqui representam a principal sintaxe fornecida para escrever ações repetidas. Também estudaremos aqui algumas instruções incomuns, como `break` e `continue`, pois elas são usadas dentro de loops.

LOOPS WHILE

A instrução `while` do Python é sua construção de iteração mais geral. Em termos simples, ela executa repetidamente um bloco de instruções endentadas, contanto que um teste no início continue avaliando um valor verdadeiro. Quando o teste se torna falso, o controle continua após todas as instruções presentes no bloco `while`; o miolo nunca é executado se o teste é falso desde o início.

A instrução `while` é uma das duas instruções de loop (junto com `for`). Ela é chamada de loop porque o controle continua voltando ao início da instrução, até que o teste se torne falso. O resultado é que o miolo do loop é executado repetidamente, enquanto o teste que está no início for verdadeiro. Além das instruções, o Python também fornece várias ferramentas que fazem loop (iteram) implicitamente: as funções `map`, `reduce` e `filter`, o teste de participação como membro `in`, compreensões de lista e muito mais. Vamos explorar a maior parte delas no Capítulo 14.

Formato geral

Em sua forma mais complexa, a instrução `while` consiste em uma linha de cabeçalho com uma expressão de teste, um miolo com uma ou mais instruções endentadas e uma parte `else` opcional, que é executada se o controle sai do loop sem passar por uma instrução `break`.

O Python continua avaliando o teste do início e executando as instruções aninhadas na parte `while`, até que o teste retorne um valor falso:

```
while <teste>:          # Faz um loop em teste
    <instruções1>       # Miolo do loop
else:                   # else opcional
    <instruções2>       # Executadas se não saiu do loop com break
```

Exemplos

Para ilustrar, aqui estão alguns loops `while` simples em ação. O primeiro apenas imprime uma mensagem para sempre, aninhando uma instrução `print` em um loop `while`. Lembre-se de que um valor inteiro `1` significa verdadeiro; como o teste é sempre verdadeiro, o Python continua executando o miolo para sempre ou até que você interrompa sua execução. Normalmente, esse tipo de comportamento é chamado de *loop infinito*:

```
>>> while 1:
...     print 'Type Ctrl-C to stop me!'
```

O próximo exemplo fica fracionando o primeiro caractere de uma string, até que a string esteja vazia e, portanto, falsa. É comum testar um objeto diretamente dessa forma, em vez de usar o equivalente mais longo: `while x != '':`. Posteriormente neste capítulo, veremos outras maneiras de percorrer mais diretamente os itens de uma string com um loop `for`.

```
>>> x = 'spam'
>>> while x:
...     print x,
...     x = x[1:]     # Retira o primeiro caractere de x.
...
spam pam am m
```

O código a seguir conta do valor de `a` até (mas não incluindo) `b`. Posteriormente, veremos uma maneira mais fácil de fazer isso com as instruções `for` e `range` do Python.

```
>>> a=0; b=10
>>> while a < b:      # Uma maneira de codificar loops contadores
...     print a,
...     a += 1         # Ou a = a+1
...
0 1 2 3 4 5 6 7 8 9
```

BREAK, CONTINUE, PASS E A CLÁUSULA ELSE

Agora que já vimos nosso primeiro loop em Python, devemos apresentar duas instruções simples que só tem significado quando aninhadas dentro de loops – as instruções `break` e `continue`. Estudaremos também a cláusula de loop `else`, pois ela é entrelaçada com `break`, e a instrução de lugar reservado vazio, `pass`. No Python:

`break`
Sai do loop mais próximo que a envolve (após a instrução de loop inteira)

`continue`
Pula para o início do loop mais próximo que a envolve (para a linha de cabeçalho do loop)

`pass`
Não faz absolutamente nada; trata-se de um lugar reservado de instrução, vazio

Bloco `else` *do loop*
É executado se, e somente se, saímos do loop normalmente – sem atingir uma instrução `break`

Formato de loop geral

Levando em conta as instruções `break` e `continue`, o formato geral do loop `while` é o seguinte:

```
while <teste1>:
    <instruções1>
    if <teste2>: break          # Sai do loop agora, pula a cláusula else.
    if <teste3>: continue       # Vai para o início do loop agora.
else:
    <instruções2>               # Se não atingimos uma instrução 'break'
```

As instruções `break` e `continue` podem aparecer em qualquer lugar dentro do miolo do loop `while` (e `for`), mas normalmente são escritas de forma mais aninhada em um teste `if`, para entrarem em ação em resposta a algum tipo de condição.

Exemplos

Vamos ver alguns exemplos simples para saber como essas instruções aparecem juntas na prática. A instrução `pass` é usada quando a sintaxe exige uma instrução, mas você não tem nada de útil a fazer. Ela é usada freqüentemente para escrever um miolo vazio de uma instrução composta. Por exemplo, se você quiser escrever um loop infinito que não faça nada sempre que for percorrido, utilize a instrução `pass`:

```
while 1: pass   # Digite Ctrl-C para interromper!
```

Como o miolo é apenas uma instrução vazia, o Python fica preso nesse loop.* De forma aproximada, a instrução `pass` está para as instruções assim como `None` está para os objetos – um nada explícito. Note que o miolo do loop `while` está na mesma linha do cabeçalho, após os dois-pontos. Assim como na instrução `if`, isso só funciona se o miolo não for uma instrução composta.

A instrução `continue` é um salto imediato para o início de um loop. Às vezes, ela permite que você evite o aninhamento de instruções. O próximo exemplo usa `continue` para pular números ímpares. Esse código imprime todos os números pares menores do que `10` e maiores ou iguais a 0. Lembre-se de que 0 significa falso e `%` é o operador de resto de divisão; portanto, esse loop conta regressivamente até zero, pulando os números que não são múltiplos de dois (ele imprime 8 6 4 2 0):

```
x = 10
while x:
    x = x-1                     # Ou x-= 1
    if x % 2 != 0: continue     # Ímpar?--pula a impressão
    print x,
```

Como a instrução `continue` pula para o início do loop, você não precisa aninhar a instrução `print` dentro de um teste `if`; a instrução `print` só é alcançada se a instrução `continue` não é executada. Se isso parece familiar com uma instrução "goto" de outras linguagens, está certo. O Python não tem nenhuma instrução `goto`, mas como a instrução `continue` permite que você salte em um programa, muitos dos alertas sobre a legibilidade e a manutenção, que você pode

* Esse código não faz nada, para sempre. Provavelmente, esse não é o programa em Python mais útil já escrito, a não ser que você queira testar um medidor de CPU ou aquecer seu computador laptop em um dia frio de inverno. Francamente, contudo, não conseguimos imaginar um exemplo melhor com a instrução `pass`. Veremos outros lugares onde ela faz sentido, posteriormente no livro (por exemplo, no Capítulo 22, para definir classes vazias que implementam objetos que se comportam como "estruturas" e "registros" em outras linguagens).

ter ouvido sobre a instrução `goto`, se aplicam. Provavelmente, a instrução `continue` deve ser pouco usada, especialmente quando você está começando a aprender sobre Python. O exemplo anterior poderia ser mais claro se a instrução `print` estivesse aninhada sob a instrução `if`, por exemplo:

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:              # Par?--imprime
        print x,
```

A instrução `break` é uma saída imediata do loop. Como o código que está depois dela nunca é atingido, às vezes a instrução `break` também pode evitar o aninhamento. Por exemplo, aqui está um loop interativo simples, o qual insere dados com `raw_input` e termina quando o usuário digita "stop" para o nome solicitado:

```
>>> while 1:
...     name = raw_input('Enter name:')
...     if name == 'stop': break
...     age  = raw_input('Enter age: ')
...     print 'Hello', name, '=>', int(age) ** 2
...
Enter name:mel
Enter age: 40
Hello mel => 1600
Enter name:bob
Enter age: 30
Hello bob => 900
Enter name:stop
```

Observe como esse código converte a entrada de `age` em um inteiro, antes de elevá-la à segunda potência, com `int; raw_input` retorna a entrada do usuário como uma string. No Capítulo 22, você verá que ela também pode lançar uma exceção no final do arquivo (por exemplo, se o usuário digitar `Ctrl-Z` ou `Ctrl-D`). Se isso importa, englobe `raw_input` em instruções `try`.

Quando combinada com a cláusula `else`, a instrução `break` freqüentemente pode eliminar os flags de status de pesquisa usados em outras linguagens. Por exemplo, o trecho de código a seguir determina se um número inteiro positivo `y` é primo, procurando fatores maiores do que `1`:

```
x = y / 2                           # Para algum y > 1
while x > 1:
    if y % x == 0:                  # Resto
        print y, 'has factor', x
        break                       # Pula a cláusula else
    x = x-1
else:                               # Saída normal
    print y, 'is prime'
```

Em vez de configurar um flag para ser testado quando o loop termina, insira uma instrução `break` onde um fator é encontrado. Desse modo, a cláusula `else` pode supor que será executada somente se nenhum fator for encontrado. Se você não atingir a instrução `break`, o número é primo.*

* Mais ou menos. Números menores do que `2` não são considerados primos pela definição matemática estrita. Sendo realmente rigorosos, esse código também falha para números negativos e em ponto flutuante, além de ser arruinado pela alteração da futura divisão "real" `/`, mencionada no Capítulo 4. Se você quiser experimentar esse código, veja o exercício no final da Parte 4, que o engloba em uma função.

A cláusula `else` também será executada se o miolo do loop nunca for executado, pois você também não executa uma instrução `break` nesse caso. Em um loop `while`, isso acontece se o teste no cabeçalho é falso logo no início. No exemplo anterior, você ainda recebe a mensagem `"is prime"` (é primo) se `x` for inicialmente menor ou igual a `1` (por exemplo, se `y` for `2`).

Mais informações sobre a cláusula else

Como a cláusula `else` é única no Python, à primeira vista ela tende a confundir alguns iniciantes. Em termos mais gerais, a cláusula `else` fornece uma sintaxe explícita para um cenário de desenvolvimento comum – ela é uma estrutura de desenvolvimento que permite a você capturar a "outra" maneira de sair de um loop, sem configurar nem verificar flags ou condições.

Suponha, por exemplo, que você esteja escrevendo um loop para procurar um valor em uma lista e precise saber se o valor foi encontrado após sair do loop. Você poderia escrever essa tarefa da seguinte maneira:

```
found = 0
while x and not found:
    if match(x[0]):              # O valor está no início?
        print 'Ni'
        found = 1
    else:
        x = x[1:]                # Fraciona o início e repete.
if not found:
    print 'not found'
```

Aqui, inicializamos, configuramos e, posteriormente, testamos um flag, para saber se a pesquisa foi bem-sucedida. Esse é um código Python válido e funciona, mas é exatamente o tipo de estrutura para a qual a cláusula foi feita para manipular. Aqui está um código equivalente com a cláusula `else`:

```
while x:                         # Sai quando x está vazio.
    if match(x[0]):
        print 'Ni'
        break                    # Sai, vai para outro lugar.
    x = x[1:]
else:
    print 'Not found            # Aqui, somente se esgotou x.
```

Aqui, o flag desapareceu e substituímos o teste `if` no fim do loop por uma instrução `else` (alinhada verticalmente com a palavra `while`, pela endentação). Como a instrução `break`, dentro da parte principal da instrução `while`, sai do loop e rodeia a instrução `else`, isso serve como uma maneira mais estruturada de capturar o caso de falha da pesquisa.

Alguns leitores podem notar que a cláusula `else` do exemplo anterior poderia ser substituída por um teste de `x` vazio após o loop (por exemplo, `if not x:`). Embora isso seja verdade nesse exemplo, a cláusula `else` fornece uma sintaxe explícita para esse padrão de codificação (é mais obviamente uma cláusula de falha de pesquisa aqui) e, em alguns casos, esse teste de vazio explícito pode não se aplicar. Além disso, a cláusula `else` se torna ainda mais útil quando usada em conjunto com o loop `for`, pois a iteração da seqüência não está sob seu controle.

LOOPS for

O loop `for` é um iterador de seqüência genérico no Python: ele pode percorrer os itens de qualquer objeto seqüência ordenada. O loop `for` funciona em strings, listas, tuplas e em novos objetos que criaremos posteriormente com classes.

Formato geral

O loop `for` do Python começa com uma linha de cabeçalho que especifica um destino (ou destinos) de atribuição, junto com um objeto que você queira percorrer. O cabeçalho é seguido por um bloco de instruções endentadas que você queira repetir:

```
for <destino> in <objeto>:      # Atribui itens do objeto ao destino.
    <instruções>                # Miolo do loop repetido: usa o destino
else:
    <instruções>                # Se não atingirmos uma instrução 'break'
```

Quando o Python executa um loop `for`, ele atribui os itens do objeto seqüência ao *destino*, um por um, e executa o miolo do loop para cada um. Normalmente, o miolo do loop usa o

Por que isto é relevante: simulando loops while da linguagem C

A seção sobre instruções de expressão afirmou que o Python não permite que instruções, como as atribuições, apareçam em lugares onde ela espera uma expressão. Isso significa que um padrão de desenvolvimento comum da linguagem C não funcionaria no Python:

```
while ((x = next()) != NULL) {...processa x...}
```

As atribuições da linguagem C retornam o valor atribuído; as atribuições do Python são apenas instruções e não expressões. Isso elimina uma notória classe de erros da linguagem C (você não pode digitar = acidentalmente no Python, quando quiser escrever ==). Mas se você precisa de um comportamento semelhante, existem pelo menos três maneiras de obter o mesmo efeito nos loops `while` do Python, sem incoporar atribuições em testes de loop. Você pode mover a atribuição para o miolo do loop com uma instrução `break`:

```
while 1:
    x = next()
    if not x: break
    ...processa x...
```

mover a atribuição para o loop com testes:

```
x = 1
while x:
    x = next()
    if x:
        ...processa x...
```

ou mover a primeira atribuição para fora do loop:

```
x = next()
while x:
    ...processa x...
    x = next()
```

Desses três padrões de desenvolvimento, o primeiro pode ser considerado, por alguns, como o menos estruturado, mas também parece ser o mais simples e mais comumente utilizado. Um loop `for` simples do Python também pode substituir alguns loops da linguagem C.

destino da atribuição para se referir ao item corrente na seqüência, como se fosse um cursor percorrendo uma seqüência.

O nome usado como destino da atribuição em uma linha de cabeçalho de loop `for` normalmente é uma variável (possivelmente nova), no escopo onde o loop `for` é escrito. Não há nada de muito especial nisso; ela pode até ser alterada dentro do miolo do loop, mas será configurada automaticamente como o próximo item da seqüência, quando o controle retornar novamente para o início do loop. Após o loop, essa variável normalmente ainda se refere ao último item visitado, que é o último item da seqüência, a menos que se saia do loop com uma instrução `break`.

O loop `for` também aceita um bloco `else` opcional, que funciona exatamente como acontece nos loops `while`; ele é executado se o loop termina sem encontrar uma instrução `break` (isto é, se todos os itens da seqüência foram visitados). Na verdade, as instruções `break` e `continue`, apresentadas anteriormente, funcionam da mesma forma no loop `for` e no loop `while`. O formato completo do loop `for` pode ser descrito da seguinte maneira:

```
for <destino> in <objeto>:      # Atribui itens do objeto ao destino.
    <instruções>
    if <teste>: break           # Sai do loop agora, pula a cláusula else.
    if <teste>: continue        # Vai para o início do loop agora.
else:
    <instruções>                # Se não atingimos uma instrução 'break'
```

Exemplos

Vamos digitar alguns loops `for` interativamente. No primeiro exemplo, o nome `x` é atribuído a cada um dos três itens da lista por sua vez, da esquerda para a direita, e a instrução `print` é executada para cada um. Dentro da instrução `print` (o miolo do loop), o nome `x` refere-se ao item corrente na lista:

```
>>> for x in ["spam", "eggs", "ham"]:
...     print x,
...
spam eggs ham
```

Os próximos dois exemplos calculam a soma e o produto de todos os itens de uma lista. No próximo capítulo, veremos as funções internas que aplicam operações como + e * a itens de uma lista automaticamente, mas normalmente é fácil usar apenas um loop `for`:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in[1, 2, 3, 4]:prod *= item
...
>>> prod
24
```

Os loops `for` também funcionam em strings e tuplas – qualquer seqüência funciona em um loop `for`:

```
>>> S, T = "lumberjack", ("and", "I'm", "okay")

>>> for x in S: print x,
...
l u m b e r j a c k

>>> for x in T: print x,
...
and I'm okay
```

Se você estiver fazendo uma iteração por uma seqüência de tuplas, o destino do loop pode ser, na verdade, uma *tupla* de destinos. Esse é apenas outro caso de atribuição de desempacotamento de tupla em funcionamento; lembre-se de que o loop `for` atribui itens da seqüência ao destino e a atribuição funciona de maneira igual em todos os lugares:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:              # Atribuição de tupla em funcionamento
...     print a, b
...
1 2
3 4
5 6
```

Aqui, na primeira passagem pelo loop, é como escrever `(a,b) = (1,2)`. Na segunda, `(a,b)` é atribuído a `(3,4)` e assim por diante. Esse não é um caso especial; qualquer destino de atribuição funciona sintaticamente após a palavra `for`.

Agora, vamos ver algo um pouco mais sofisticado. O próximo exemplo ilustra a cláusula `else` em um loop `for` e o aninhamento de instruções. Dada uma lista de objetos (itens) e uma lista de chaves (tests), este código pesquisa cada chave na lista de objetos e relata o resultado da pesquisa:

```
>>> items = ["aaa", 111, (4, 5), 2.01      # Um conjunto de objetos
>>> tests = [(4, 5), 3.14]                 # Chaves para pesquisar
>>>
>>> for key in tests:                      # Para todas as chaves
...     for item in item:                  # Para todos os itens
...         if item == key:                # Verifica a correspondência.
...             print key, "was found"
...             break
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

Como a instrução `if` aninhada executa uma instrução `break` quando uma correspondência é encontrada, a cláusula `else` do loop pode supor que a pesquisa falhou. Observe o aninhamento aqui: quando esse código é executado, existem dois loops ocorrendo ao mesmo tempo. O loop externo percorre a lista de chaves e o loop interno percorre a lista de itens para cada chave. O aninhamento da cláusula `else` é fundamental; ela está endentada no mesmo nível da linha de cabeçalho do loop `for` interno; portanto, é associada ao loop interno (e não ao `if` ou ao `for` externo).

A propósito, esse exemplo será mais fácil de escrever se empregarmos o operador `in` para testar a participação como membro. Como o operador `in` percorre uma lista implicitamente, em busca de uma correspondência, ele substitui o loop interno:

```
>>> for key in tests:          # Para todas as chaves
...     if key in items:       # Deixa a Python procurar uma correspondência.
...         print key, "was found"
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

Em geral, é uma boa idéia deixar o Python fazer o máximo de trabalho possível, por questões de brevidade e desempenho. O próximo exemplo executa uma tarefa típica de estrutura de dados com um loop `for` – coletar itens comuns em duas seqüências (strings). É mais ou menos uma rotina de interseção de conjuntos simples; após a execução do loop, `res` refere-se a uma lista que contém todos os itens encontrados em `seq1` e em `seq2`:

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>> res = []                   # Começa vazio.
>>> for x in seq1:             # Percorre a primeira seqüência.
...     if x in seq2:          # Item comum?
...         res.append(x)      # Adiciona no final do resultado.
...
>>> res
['s', 'a', 'm']
```

Infelizmente, esse código está equipado para funcionar apenas com duas variáveis específicas: `seq1` e `seq2`. Seria ótimo se esse loop pudesse ser generalizado de alguma forma, em uma ferramenta que você pudesse usar mais de uma vez. Conforme você verá, essa idéia simples nos leva às funções, o assunto da Parte 4.

VARIAÇÕES DE LOOP

O loop `for` subordina a maioria dos loops estilo contador. Geralmente ele é mais simples de escrever e mais rápido para executar do que um loop `while`, de modo que é a primeira ferramenta que você deve procurar quando precisar percorrer uma seqüência. Mas também existem situações em que você precisará iterar de uma maneira mais especializada. Por exemplo, e se você precisar visitar todo segundo e terceiro item de uma lista ou alterar a lista no processo? E quanto a percorrer mais de uma seqüência em paralelo no mesmo loop `for`?

Você sempre pode escrever tais iterações exclusivas com um loop while e indexação manual, mas o Python fornece duas funções internas que permitem especializar a iteração em um loop `for`:

- A função interna `range` retorna uma lista de inteiros sucessivamente mais altos, que podem ser usados como índices em um loop `for`.*
- A função interna `zip` retorna uma lista de tuplas de itens paralelos, que podem ser usadas para percorrer várias seqüências em um loop `for`.

Vamos ver cada uma dessas funções internas por sua vez.

* O Python também fornece uma função interna chamada `xrange` que gera um índice por vez, em vez de armazená-los simultaneamente em uma lista, como faz a função `range`. Não há nenhuma vantagem na velocidade de `xrange`, mas ela é útil como uma otimização de espaço, caso você precise gerar um número muito grande de valores.

Loops contadores: range

A função `range` é independente dos loops `for`. Embora seja mais freqüentemente usada para gerar índices em um loop `for`, você pode usá-la sempre que precise de uma lista de inteiros:

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

Por que isto é relevante: varredores de arquivo

Em geral, os loops são úteis em qualquer lugar onde você precise repetir ou processar algo mais de uma vez. Como os arquivos contêm vários caracteres e linhas, eles são um dos usos mais típicos para os loops. Para carregar todo o conteúdo de um arquivo em uma string, de uma só vez, você simplesmente chama a instrução `read`:

```
file = open('test.txt', 'r')
print file.read()
```

Mas para carregar um arquivo por partes, é comum escrever um loop `while` com instruções `break` no final do arquivo ou um loop `for`. Para ler por caracteres:

```
file = open('test.txt')
while 1:
    char = file.read(1)                 # Lê por caractere.
    if not char: break
    print char,

for char in open('test.txt').read():
    print char
```

O loop `for` aqui também processa cada caractere, mas carrega o arquivo na memória, todo de uma vez. Para ler por linhas ou blocos com um loop `while`:

```
file = open('test.txt')
while 1:
    line = file.readline()              # Lê linha por linha.
    if not line: break
    print line,

file = open('test.txt', 'rb')
while 1:
    chunk = file.read(10)               # Lê trechos de byte.
    if not chunk: break
    print chunk,
```

Contudo, para ler arquivos de texto linha por linha, o loop `for` tende a ser mais fácil de escrever e mais rápido para executar:

```
for line in open('test.txt').readlines(): print line
for line in open('test.txt').xreadlines():print line
for line in open('test.txt'): print line
```

`readlines` carrega um arquivo todo de uma vez em uma lista de string de linha; `xreadlines`, em vez disso, carrega as linhas de acordo com a demanda, para não encher a memória no caso de arquivos grandes. O último exemplo aqui conta com os novos *iteradores* de arquivo para obter o equivalente de `xreadlines` (os iteradores serão abordados no Capítulo 14). A partir do Python 2.2, o nome `open`, em todos os exemplos anteriores também, pode ser substituído pelo nome `file`. Consulte o manual da biblioteca para ver mais informações sobre as chamadas usadas aqui. Como regra geral, quanto maior o tamanho dos dados que você lê em cada passo, mais rapidamente seu programa é executado.

Com um argumento, a função `range` gera uma lista com inteiros de zero até (mas não incluindo) o valor do argumento. Se você passar dois argumentos, o primeiro será considerado como o limite inferior. Um terceiro argumento opcional pode fornecer um *passo*; se for usado, o Python soma o passo a cada inteiro sucessivo no resultado (o padrão dos passos é um). Os intervalos também podem ser negativos, em ordem descendente, caso você queira:

```
>>> range(-5, 5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4,]

>>> range(5, -5, -1)
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Embora todos esses resultados de intervalo possam ser úteis por si só, eles tendem a ser mais úteis dentro de loops `for`. Por exemplo, eles fornecem uma maneira simples de repetir uma ação por um número específico de vezes. Para imprimir três linhas, por exemplo, use um intervalo para gerar o número de inteiros apropriado:

```
>>> for i in range(3):
...     print i, 'Pythons'
...
0 Pythons
1 Pythons
2 Pythons
```

A função `range` também é comumente usada para fazer iteração em uma seqüência, indiretamente. A maneira mais fácil e mais rápida de percorrer uma seqüência exaustivamente sempre é com um loop `for` simples. O Python trata da maior parte dos detalhes para você:

```
>>> X = 'spam'
>>> for item in X: print item,          # Iteração simples
...
s p a m
```

Observe a vírgula no final da instrução `print` aqui, para suprimir o avanço de linha padrão (cada impressão continua adicionando na linha de saída corrente). Internamente, o loop `for` trata dos detalhes da iteração automaticamente. Se você precisar realmente assumir explicitamente o controle da lógica de indexação, pode fazer isso com um loop `while`:

```
>>> i = 0
>>> while i < len(X):                   # iteração do loop while
...     print X[i],; i += 1
...
s p a m
```

Você também pode fazer indexação manual com um loop `for`, se usar a função `range` para gerar uma lista de índices para fazer a iteração:

```
>>> X
'spam'
>>> len(X)                              # Comprimento da string
4
>>> range(len(X))                       # Todos os deslocamentos válidos em X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): printX[i],  # Indexação de loop for manual
...
s p a m
```

O exemplo aqui está percorrendo uma lista de *deslocamentos* em `X` e não os *itens* reais de `X`. Precisamos indexar de volta em `X`, dentro do loop, para buscar cada item.

Varreduras não exaustivas: range

O último exemplo da seção anterior funciona, mas provavelmente é executado de forma mais lenta do que deveria. A não ser que você tenha um requisito de indexação especial, é sempre melhor usar a forma de loop `for` simples no Python – use o loop `for`, em vez do loop `while` quando possível, e não conte com chamadas de `range` em loops `for`, exceto como,último recurso.

Entretanto, o mesmo padrão de desenvolvimento usado naquele exemplo anterior também nos permite fazer tipos de varreduras mais especializadas:

```
>>> S = 'abcdefghijk'
>>> range(0, len(S), 2)
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print S[i],
...
a c e g i k
```

Aqui, visitamos cada *segundo* item na string `S`, percorrendo a lista gerada por `range`. Para visitar cada terceiro item, mude o terceiro argumento de `range` para `3` e assim por diante. Na verdade, a função `range` usada dessa maneira permite que você pule itens em loops, enquanto ainda mantém a simplicidade do loop `for`. Consulte também o novo terceiro limite de fracionamento opcional do Python 2.3, na seção "Indexação e fracionamento" do Capítulo 5. Na versão 2.3, um efeito semelhante pode ser obtido com:

```
for x in S[::2]: print x
```

Alterando listas: range

Outro lugar comum em que você pode usar a função `range` e o loop `for` combinados é em loops que alteram uma lista enquanto ela está sendo percorrida. O exemplo a seguir precisa de um índice para poder atribuir um valor atualizado a cada posição, à medida que avançamos:

```
>>> L = [1, 2, 3, 4, 5]
>>>
>>> for i in range(len(L)):          # Soma um a cada item em L
...     L[i] += 1                    # Ou L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

Aqui, não há nenhuma maneira de fazer o mesmo com um estilo de loop `for x in L:`, pois tal loop itera pelos itens reais e não por posições de uma lista. O loop `while` equivalente exige um pouco mais de trabalho de nossa parte:*

* Uma expressão de *compreensão de lista*, da forma `[x+1 for x in L]`, também faria um trabalho semelhante aqui, se bem que sem alterar a lista original no local (poderíamos atribuir o resultado do novo objeto lista da expressão de volta a `L`, mas isso não atualizaria quaisquer outras referências à lista original). Consulte o Capítulo 14 para ver mais informações sobre compreensões de lista.

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

Varreduras paralelas: zip e map

O truque da função `range` percorre seqüências com um loop `for` de maneira exaustiva. A função interna `zip` nos permite usar loops `for` para visitar várias seqüências em *paralelo*. Na operação básica, a função `zip` pega uma ou mais seqüências e retorna uma lista de tuplas que dispõem em pares os itens paralelos extraídos de seus argumentos. Por exemplo, suponha que estejamos trabalhando com duas listas:

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

Para combinar os itens dessas listas, podemos usar a função `zip`:

```
>>> zip(L1,L2)
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Esse resultado pode ser útil em outros contextos. Contudo, quando aliado ao loop `for`, ele suporta iterações paralelas:

```
>>> for (x,y) in zip(L1, L2):
...     print x, y, '--', x+y
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Aqui, percorremos o resultado da chamada de zip – os pares de itens extraídos das duas listas. Esse loop `for` usa atribuição de tupla novamente, para desempacotar cada tupla no resultado da função `zip` (na primeira vez, é como se executássemos `(x,y) =(1,5)`). O efeito é que percorremos `L1` e `L2` em nosso loop. Poderíamos obter um efeito semelhante com um loop while que manipulasse a indexação manualmente, mas teríamos mais coisas para digitar e poderia ser mais lento do que a estratégia `for/zip`.

A função `zip` é mais geral do que esse exemplo sugere. Por exemplo, ela aceita qualquer tipo de seqüência e mais de dois argumentos:

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> zip(T1,T2,T3)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

A função `zip` trunca as tuplas resultantes no comprimento da seqüência mais curta, quando os comprimentos dos argumentos diferem:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
```

```
>>> zip(S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

A relacionada e mais antiga função interna `map` cria pares de itens de seqüências de maneira semelhante, mas preenche as seqüências mais curtas com `None`, caso os comprimentos dos argumentos sejam diferentes:

```
>>> map(None, S1, S2)
[('a','x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

Na verdade, o exemplo está usando uma forma degenerada da função interna `map`. Normalmente, a função `map` recebe uma função, um ou mais argumentos de seqüência e reúne os resultados da chamada da função com itens paralelos extraídos das seqüências. Quando o argumento da função é `None` (como aqui), ela simplesmente cria pares de itens, como a função `zip`. A função `map` e ferramentas baseadas em função semelhantes serão abordadas no Capítulo 14.

Construção de dicionário com zip

Os dicionários sempre podem ser criados escrevendo um literal de dicionário ou pela atribuição de chaves com o passar do tempo:

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'toast': 5, 'eggs': 3, 'spam': 1}

>>> D1 = {}
>>> D1['spam'] = 1
>>> D1['eggs'] = 3
>>> D1['toast'] = 5
```

Contudo, o que fazer se seu programa recebe chaves e valores de dicionário em *listas* em tempo de execução, após você ter escrito seu script?

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

Uma solução para ir das listas para um dicionário é usar a função `zip` nas listas e percorrê-las em paralelo com um loop `for`:

```
>>> zip(keys,vals)
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'toast': 5, 'eggs':3, 'spam': 1}
```

Contudo, verifica-se que, no Python 2.2, você pode pular completamente o loop `for` e simplesmente passar as listas de chaves/valores da função `zip` para a chamada de construtor interno `dict`:

```
>>> keys = ['spam','eggs', 'toast']
>>> vals = [1, 3, 5]
>>> D3 = dict(zip(keys,vals))
>>> D3
{'toast': 5, 'eggs':3, 'spam': 1}
```

O nome interno `dict` é, na verdade, um nome de tipo no Python. Às vezes, chamá-lo é algo como uma conversão de lista para dicionário, mas na verdade é um pedido de construção de objeto (mais informações sobre nomes de tipo aparecem no Capítulo 23). Além disso, no Capítulo 14, conheceremos um conceito relacionado, porém mais rico: a *compreensão de lista*, que constrói listas em uma única expressão.