Capítulo 5: Estruturas de dados

Este capítulo descreve alguns pontos já abordados, porém com mais detalhes, e adiciona outros pontos.

Mais sobre listas

O tipo `list` possui mais métodos. Aqui estão todos os métodos disponíveis em um objeto lista:

list.append(x)

Adiciona um item ao fim da lista; equivale a `a[len(a):] = [x]`.

list.extend(L)

Prolonga a lista, adicionando no fim todos os elementos da lista `L` passada como argumento; equivalente a `a[len(a):] = L`.

list.insert(i, x)

Insere um item em uma posição especificada. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim `a.insert(0, x)` insere no início da lista, e `a.insert(len(a), x)` equivale a `a.append(x)`.

list.remove(x)

Remove o primeiro item encontrado na lista cujo valor é igual a *x*. Se não existir valor igual, uma exceção `ValueError` é levantada.

list.pop([i])

Remove o item na posição dada e o devolve. Se nenhum índice for especificado, `a.pop()` remove e devolve o último item na lista. (Os colchetes ao redor do *i* indicam que o parâmetro é opcional, não que você deva digitá-los daquela maneira. Você verá essa notação com frequência na Referência da Biblioteca Python.)

list.index(x)

Devolve o índice do primeiro item cujo valor é igual a *x*, gerando `ValueError` se este valor não existe

list.count(x)

Devolve o número de vezes que o valor *x* aparece na lista.

list.sort()

Ordena os itens na própria lista *in place*.

list.reverse()

Inverte a ordem dos elementos na lista *in place* (sem gerar uma nova lista).

Um exemplo que utiliza a maioria dos métodos::

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

(N.d.T. Note que os métodos que alteram a lista, inclusive `sort` e `reverse`, devolvem `None` para lembrar o programador de que modificam a própria lista, e não criam uma nova. O único método que altera a lista e devolve um valor é o `pop`)

Usando listas como pilhas

Os métodos de lista tornam muito fácil utilizar listas como pilhas, onde o item adicionado por último é o primeiro a ser recuperado (política "último a entrar, primeiro a sair"). Para adicionar um item ao topo da pilha, use `append`. Para recuperar um item do topo da pilha use `pop` sem nenhum índice. Por exemplo:

```
>>> pilha = [3, 4, 5]
>>> pilha.append(6)
>>> pilha.append(7)
>>> pilha
[3, 4, 5, 6, 7]
>>> pilha.pop()
7
>>> pilha
[3, 4, 5, 6]
>>> pilha.pop()
6
>>> pilha.pop()
5
>>> pilha
[3, 4]
```

Usando listas como filas

Você também pode usar uma lista como uma fila, onde o primeiro item adicionado é o primeiro a ser recuperado (política "primeiro a entrar, primeiro a sair"); porém, listas não são eficientes para esta finalidade. Embora *appends* e *pops* no final da lista sejam rápidos, fazer *inserts* ou *pops* no início da lista é lento (porque todos os demais elementos têm que ser deslocados).

Para implementar uma fila, use a classe `collections.deque` que foi projetada para permitir *appends* e *pops* eficientes nas duas extremidades. Por exemplo:

```
>>> from collections import deque
>>> fila = deque(["Eric", "John", "Michael"])
>>> fila.append("Terry")     # Terry chega
>>> fila.append("Graham")    # Graham chega
>>> fila.popleft()           # O primeiro a chegar parte
'Eric'
>>> fila.popleft()           # O segundo a chegar parte
'John'
>>> fila                     # O resto da fila, em ordem de chegada
deque(['Michael', 'Terry', 'Graham'])
```

(N.d.T. neste exemplo são usados nomes de membros do grupo *Monty Python*)

Ferramentas de programação funcional

Existem três funções embutidas que são muito úteis para processar listas: `filter`, `map`, e `reduce`.

`filter(funcao, sequencia)` devolve uma nova sequência formada pelos itens do segundo argumento para os quais `funcao(item)` é verdadeiro. Se a sequencia de entrada for string ou tupla, a saída será do mesmo tipo; caso contrário, o resultado será sempre uma lista. Por exemplo, para computar uma sequência de números não divisíveis por 2 ou 3:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

`map(funcao, sequencia)` aplica `funcao(item)` a cada item da sequência e devolve uma lista formada pelo resultado de cada aplicação. Por exemplo, para computar cubos:

```
>>> def cubo(x): return x*x*x
...
>>> map(cubo, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Mais de uma sequência pode ser passada; a função a ser aplicada deve aceitar tantos argumentos quantas sequências forem passadas, e é invocada com o item correspondente de cada sequência (ou `None`, se alguma sequência for menor que outra). Por exemplo:

```
>>> seq = range(8)
>>> def somar(x, y): return x+y
...
>>> map(somar, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

Se `None` for passado no lugar da função, então será aplicada a função identidade (apenas devolve o argumento recebido). Se várias sequências forem passadas, a lista resultante terá tuplas formadas pelos elementos correspondentes de cada sequência. Isso se parece com a função `zip`, exceto que

`map` devolve uma lista com o comprimento da sequência mais longa que foi passada, preenchendo as lacunas com `None` quando necessário, e `zip` devolve uma lista com o comprimento da mais curta. Confira:

```
>>> map(None, range(5))
[0, 1, 2, 3, 4]
>>> map(None, range(5), range(3))
[(0, 0), (1, 1), (2, 2), (3, None), (4, None)]
>>> zip(range(5), range(3))
[(0, 0), (1, 1), (2, 2)]
>>>
```

A função `reduce(funcao, sequencia)` devolve um único valor construído a partir da sucessiva aplicação da função binária (N.d.T. que recebe dois argumentos) a todos os elementos da lista fornecida, começando pelos dois primeiros itens, depois aplicando a função ao primeiro resultado obtido e ao próximo item, e assim por diante. Por exemplo, para computar a soma dos inteiros de 1 a 10:

```
>>> def somar(x,y): return x+y
...
>>> reduce(somar, range(1, 11))
55
```

Se houver um único elemento na sequência fornecida, seu valor será devolvido. Se a sequência estiver vazia, uma exceção será levantada.

Um terceiro argumento pode ser passado para definir o valor inicial. Neste caso, redução de uma sequência vazia devolve o valor inicial. Do contrário, a redução se inicia aplicando a função ao valor inicial e ao primeiro elemento da sequência, e continuando a partir daí.

```
>>> def somatoria(seq):
...     def somar(x,y): return x+y
...     return reduce(somar, seq, 0)
...
>>> somatoria(range(1, 11))
55
>>> somatoria([])
0
```

Não use a função `somatória` deste exemplo; somar sequências de números é uma necessidade comum, e para isso Python tem a função embutida `sum`, que faz exatamente isto, e também aceita um valor inicial (opcional).

List comprehensions ou abrangências de listas

Uma *list comprehension* é uma maneira concisa de construir uma lista preenchida. (N.d.T. literalmente, *abrangência de lista* mas no Brasil o termo em inglês é muito usado; também se usa a abreviação *listcomp*)

Um uso comum é construir uma nova lista onde cada elemento é o resultado de alguma expressão aplicada a cada membro de outra sequência ou iterável, ou para construir uma subsequência cujos elementos satisfazem uma certa condição.

Por exemplo, suponha que queremos criar uma lista de quadrados, assim:

```
>>> quadrados = []
>>> for x in range(10):
...     quadrados.append(x**2)
...
>>> quadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Podemos obter o mesmo resultado desta forma:

```
quadrados = [x**2 for x in range(10)]
```

Isso equivale a `quadrados = map(lambda x: x**2, range(10))`, mas é mais conciso e legível.

Uma abrangência de lista é formada por um par de colchetes contendo uma expressão seguida de uma cláusula `for`, e então zero ou mais cláusulas `for` ou `if`. O resultado será uma lista resultante da avaliação da expressão no contexto das cláusulas `for` e `if`.

Por exemplo, esta listcomp combina os elementos de duas listas quando eles são diferentes:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Isto equivale a:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note como a ordem dos `for` e `if` é a mesma nos dois exemplos acima.

Se a expressão é uma tupla, ela deve ser inserida entre parênteses (ex., `(x, y)` no exemplo anterior).

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # criar uma lista com os valores dobrados
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtrar a lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplicar uma função a todos os elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # invocar um método em cada elemento
>>> frutas = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [arma.strip() for arma in frutas]
['banana', 'loganberry', 'passion fruit']
>>> # criar uma lista de duplas, ou tuplas de 2, como (numero, quadrado)
>>> [(x, x**2) for x in range(6)]
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # a tupla deve estar emtre parêntesis, do contrário ocorre um erro
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # achatar uma lista usando uma listcomp com dois 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A abrangência de lista é mais flexível do que `map` e pode conter expressões complexas e funções aninhadas, sem necessidade do uso de `lambda`:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Listcomps aninhadas

A expressão inicial de uma listcomp pode ser uma expressão arbitrária, inclusive outra listcomp.

Observe este exemplo de uma matriz 3x4 implementada como uma lista de 3 listas de comprimento 4:

```
>>> matriz = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

A abrangência de listas abaixo transpõe as linhas e colunas:

```
>>> [[linha[i] for linha in matriz] for i in range(len(matriz[0]))]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Como vimos na seção anterior, a listcomp aninhada é computada no contexto da cláusula `for` seguinte, portanto o exemplo acima equivale a:

```
>>> transposta = []
>>> for i in range(len(matriz[0])):
...     transposta.append([linha[i] for linha in matriz])
...
>>> transposta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

e isso, por sua vez, faz o mesmo que isto:

```
>>> transposta = []
>>> for i in range(len(matriz[0])):
...     # as próximas 3 linhas implementam a listcomp aninhada
...     linha_transposta = []
...     for linha in matriz:
...         linha_transposta.append(linha[i])
...     transposta.append(linha_transposta)
...
```

```
>>> transposta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
>>>
```

Na prática, você deve dar preferência a funções embutidas em vez de expressões complexas. A função `zip` resolve muito bem este caso de uso:

```
>>> zip(*matriz)
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Veja *Desempacotando listas de argumentos [Capítulo 4]* para entender o uso do asterisco neste exemplo.

O comando `del`

Existe uma maneira de remover um item de uma lista conhecendo seu índice, ao invés de seu valor: o comando `del`. Ele difere do método `list.pop`, que devolve o item removido. O comando `del` também pode ser utilizado para remover fatias (slices) da lista, ou mesmo limpar a lista toda (que fizemos antes atribuindo uma lista vazia à fatia `a[:]`). Por exemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` também pode ser usado para remover totalmente uma variável:

```
>>> del a
>>> a
Traceback (most recent call last):
  ...
NameError: name 'a' is not defined
```

Referenciar a variável `a` depois de sua remoção constitui erro (pelo menos até que seja feita uma nova atribuição para ela). Encontraremos outros usos para o comando `del` mais tarde.

Tuplas e sequências

Vimos que listas e strings têm muitas propriedades em comum, como indexação e operações de fatiamento (*slicing*). Elas são dois exemplos de *sequências* (veja Sequence Types). Como Python é uma linguagem em evolução, outros tipos de sequências podem ser adicionados. Existe ainda um outro tipo de sequência padrão na linguagem: a tupla (*tuple*).

Uma tupla consiste em uma sequência de valores separados por vírgulas:

```
>>> t = 12345, 54321, 'bom dia!'
```

```
>>> t[0]
12345
>>> t
(12345, 54321, 'bom dia!')
>>> # Tuplas podem ser aninhadas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'bom dia!'), (1, 2, 3, 4, 5))
```

Como você pode ver no trecho acima, na saída do console as tuplas são sempre envolvidas por parênteses, assim tuplas aninhadas podem ser lidas corretamente. Na criação, tuplas podem ser envolvidas ou não por parênteses, desde que o contexto não exija os parênteses (como no caso da tupla dentro a uma expressão maior).

Tuplas podem ser usadas de diversas formas: pares ordenados `(x, y)`, registros de funcionário extraídos uma base de dados, etc. Tuplas, assim como strings, são imutáveis: não é possível atribuir valores a itens individuais de uma tupla (você pode simular o mesmo efeito através de operações de fatiamento e concatenação; N.d.T. mas neste caso nunca estará modificando tuplas, apenas criando novas). Também é possível criar tuplas contendo objetos mutáveis, como listas.

Um problema especial é a criação de tuplas contendo 0 ou 1 itens: a sintaxe usa certos truques para acomodar estes casos. Tuplas vazias são construídas por uma par de parênteses vazios; uma tupla unitária é construída por um único valor e uma vírgula entre parênteses (não basta colocar um único valor entre parênteses). Feio, mas funciona:

```
>>> vazia = ()
>>> upla = 'hello',    # <-- note a vírgula no final
>>> len(vazia)
0
>>> len(upla)
1
>>> upla
('hello',)
```

O comando `t = 12345, 54321, 'hello!'` é um exemplo de *empacotamento de tupla* (*tuple packing*): os valores `12345`, `54321` e `'bom dia!'` são empacotados juntos em uma tupla. A operação inversa também é possível:

```
>>> x, y, z = t
```

Isto é chamado de desempacotamento de sequência (*sequence unpacking*), funciona para qualquer tipo de sequência do lado direito. Para funcionar, é necessário que a lista de variáveis do lado esquerdo tenha o mesmo comprimento da sequência à direita. Sendo assim, a atribuição múltipla é um caso de empacotamento de tupla e desempacotamento de sequência:

```
>>> a, b = b, a  # troca os valores de a e b
```

Existe uma certa assimetria aqui: empacotamento de múltiplos valores sempre cria tuplas, mas o desempacotamento funciona para qualquer sequência.

Sets (conjuntos)

Python também inclui um tipo de dados para conjuntos, chamado `set`. Um conjunto é uma coleção desordenada de elementos, sem elementos repetidos. Usos comuns para sets incluem a verificação eficiente da existência de objetos e a eliminação de itens duplicados. Conjuntos também suportam operações matemáticas como união, interseção, diferença e diferença simétrica.

Uma pequena demonstração:

```
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> frutas = set(cesta)   # criar um conjunto sem duplicatas
>>> frutas
set(['abacaxi', 'uva', 'laranja', 'banana'])
>>> 'laranja' in frutas   # testar se um elemento existe é muito rápido
True
>>> 'capim' in frutas
False

>>> # Demonstrar operaçes de conjunto em letras únicas de duas palavras
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras unicas em a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letras em a mas não em b
set(['r', 'd', 'b'])
>>> a | b                             # letras em a ou em b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letras tanto em a como em b
set(['a', 'c'])
>>> a ^ b                             # letras em a ou b mas não em ambos
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

N.d.T. A sintaxe de sets do Python 3.1 foi portada para o Python 2.7, tornando possível escrever `{10, 20, 30}` para definir `set([10, 20, 30])`. O conjunto vazio tem que ser escrito como `set()` ou `set([])`, pois `{}` sempre representou um dicionário vazio, como veremos a seguir. Também existe uma sintaxe para *set comprehensions*, que nos permite escrever `{x*10 for x in [1, 2, 3]}` para construir `{10, 20, 30}`.

Dicionários

Outra estrutura de dados muito útil embutida em Python é o *dicionário*, cujo tipo é `dict` (ver Mapping Types — dict). Dicionários são também chamados de “memória associativa” ou “vetor associativo” em outras linguagens. Diferente de sequências que são indexadas por inteiros, dicionários são indexados por chaves (*keys*), que podem ser de qualquer tipo imutável (como strings e inteiros). Tuplas também podem ser chaves se contiverem apenas strings, inteiros ou outras tuplas. Se a tupla contiver, direta ou indiretamente, qualquer valor mutável, não poderá ser chave. Listas não podem ser usadas como chaves porque são podem ser modificadas *in place* pela atribuição em índices ou fatias, e por métodos como `append` e `extend`.

Um bom modelo mental é imaginar um dicionário como um conjunto não ordenado de pares chave-valor, onde as chaves são únicas em uma dada instância do dicionário. Dicionários são

delimitados por chaves: `{}`, e contém uma lista de pares *chave:valor* separada por vírgulas. Dessa forma também será exibido o conteúdo de um dicionário no console do Python. O dicionário vazio é `{}`.

As principais operações em um dicionário são armazenar e recuperar valores a partir de chaves. Também é possível remover um par *chave:valor* com o comando `del`. Se você armazenar um valor utilizando uma chave já presente, o antigo valor será substituído pelo novo. Se tentar recuperar um valor usando uma chave inexistente, será gerado um erro.

O método `keys` do dicionário devolve a lista de todas as chaves presentes no dicionário, em ordem arbitrária (se desejar ordená-las basta aplicar o a função `sorted` à lista devolvida). Para verificar a existência de uma chave, use o operador `in`.

A seguir, um exemplo de uso do dicionário:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

O construtor `dict` produz dicionários diretamente a partir de uma lista de chaves-valores, armazenadas como duplas (tuplas de 2 elementos). Quando os pares formam um padrão, uma list comprehensions pode especificar a lista de chaves-valores de forma mais compacta.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use uma list comprehension
{2: 4, 4: 16, 6: 36}
```

N.d.T. A partir do Python 2.7 também existem *dict comprehensions* (abrangências de dicionário). Com esta sintaxe o último dict acima pode ser construído assim `{x: x**2 for x in (2, 4, 6)}`.

Mais adiante no tutorial aprenderemos sobre *expressões geradoras*, que são ainda mais adequados para fornecer os pares de chave-valor para o construtor `dict`.

Quando chaves são strings simples, é mais fácil especificar os pares usando argumentos nomeados no construtor:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

N.d.T. Naturalmente, por limitações da sintaxe, essa sugestão só vale se as chaves forem strings ASCII, sem acentos, conforme a regras para formação de identificadores do Python.

Técnicas de iteração

Ao percorrer um dicionário em um laço, a variável de interação receberá uma chave de cada vez:

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k in knights:
...     print k
...
gallahad
robin
```

Quando conveniente, a chave e o valor correspondente podem ser obtidos simultaneamente com o método `iteritems`.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Ao percorrer uma sequência qualquer, o índice da posição atual e o valor correspondente podem ser obtidos simultaneamente usando a função `enumerate`:

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Para percorrer duas ou mais sequências simultaneamente com o laço, os itens podem ser agrupados com a função `zip`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}?  It is {1}.'.format(q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

N.d.T. O exemplo acima reproduz um diálogo do filme *Monty Python - Em Busca do Cálice Sagrado*. Este trecho não pode ser traduzido, exceto por um decreto real.

Para percorrer uma sequência em ordem inversa, chame a função `reversed` com a sequência na ordem original.

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
9
7
```

```
5
3
1
```

Para percorrer uma sequência de maneira ordenada, use a função `sorted`, que retorna uma lista ordenada com os itens, mantendo a sequência original inalterada.

```
>>> cesta = ['uva', 'laranja', 'uva', 'abacaxi', 'laranja', 'banana']
>>> for fruta in sorted(cesta):
...     print fruta
...
abacaxi
banana
laranja
laranja
uva
uva
>>> for fruta in sorted(set(cesta)):  # sem duplicações
...     print fruta
...
abacaxi
banana
laranja
uva
>>>
```

Mais sobre condições

As condições de controle usadas em `while` e `if` podem conter quaisquer operadores, não apenas comparações.

Os operadores de comparação `in` e `not in` verificam se um valor ocorre (ou não ocorre) em uma dada sequência. Os operadores `is` e `is not` comparam se dois objetos são na verdade o mesmo objeto; isto só é relevante no contexto de objetos mutáveis, como listas. Todos os operadores de comparação possuem a mesma precedência, que é menor do que a prioridade de todos os operadores numéricos.

Comparações podem ser encadeadas: Por exemplo `a < b == c` testa se `a` é menor que `b` e também se `b` é igual a `c`.

Comparações podem ser combinadas através de operadores booleanos `and` e `or`, e o resultado de uma comparação (ou de qualquer outra expressão), pode ter seu valor booleano negado através de `not`. Estes possuem menor prioridade que os demais operadores de comparação. Entre eles, `not` é o de maior prioridade e `or` o de menor. Dessa forma, a condição `A and not B or C` é equivalente a `(A and (not B)) or C`. Naturalmente, parênteses podem ser usados para expressar o agrupamento desejado.

Os operadores booleanos `and` e `or` são operadores *short-circuit*: seus argumentos são avaliados da esquerda para a direita, e a avaliação para quando o resultado é determinado. Por exemplo, se `A` e `C` são expressões verdadeiras, mas `B` é falsa, então `A and B and C` não chega a avaliar a expressão `C`. Em geral, quando usado sobre valores genéricos e não como booleanos, o valor do resultado de um operador atalho é o último valor avaliado na expressão.

É possível atribuir o resultado de uma comparação ou outra expressão booleana para uma variável. Por exemplo:

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Observe que em Python, diferente de C, atribuição não pode ocorrer dentro de uma expressão. Programadores C podem resmungar, mas isso evita toda uma classe de problemas frequentemente encontrados em programas C: digitar `=` numa expressão quando a intenção era `==`.

Comparando sequências e outros tipos

Objetos sequência podem ser comparados com outros objetos sequência, desde que o tipo das sequências seja o mesmo. A comparação utiliza a ordem *lexicográfica*: primeiramente os dois primeiros itens são comparados, e se diferirem isto determinará o resultado da comparação, caso contrário os próximos dois itens serão comparados, e assim por diante até que se tenha exaurido alguma das sequências. Se em uma comparação de itens, os mesmos forem também sequências (aninhadas), então é disparada recursivamente outra comparação lexicográfica. Se todos os itens da sequência forem iguais, então as sequências são ditas iguais. Se uma das sequências é uma subsequência da outra, então a subsequência é a menor. A comparação lexicográfica de strings utiliza ASCII para definir a ordenação. Alguns exemplos de comparações entre sequências do mesmo tipo:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

É permitido comparar objetos de diferentes tipos. O resultado é determinístico, porém, arbitrário: os tipos são ordenados pelos seus nomes. Então, uma `list` é sempre menor do que uma `str`, uma `str` é sempre menor do que uma `tuple`, etc. [1] Tipos numéricos misturados são comparados de acordo com seus valores numéricos, logo 0 é igual a 0.0, etc.

Notas

1 As regras para comparação de objetos de tipos diferentes não são definitivas; elas podem variar em futuras versões da linguagem.