5. Estruturas de dados

Esse capítulo descreve algumas coisas que você já aprendeu em detalhes e adiciona algumas coisas novas também.

5.1. Mais sobre listas

O tipo de dado lista tem ainda mais métodos. Aqui estão apresentados todos os métodos de objetos do tipo lista:

`list.`**`append`**`(`*`x`*`)`
Adiciona um item ao fim da lista. Similar a `a[len(a):] = [x]`.

`list.`**`extend`**`(`*`iterable`*`)`
Estende a lista, adicionando no fim todos os elementos do argumento iterável passado como parâmetro. Similar a `a[len(a):] = iterable`.

`list.`**`insert`**`(`*`i, x`*`)`
Insere um item em uma dada posição. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim `a.insert(0, x)` insere um elemento na frente da lista e `a.insert(len(a), x)` e 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 fornecida na lista e retorna. Se nenhum índice for especificado, `a.pop()` remove e retorna o último item da lista. Levanta um `IndexError` se a lista estiver vazia ou o índice estiver fora do intervalo da lista.

`list.`**`clear`**`()`
Remove todos os itens de uma lista. Similar a `del a[:]`.

`list.`**`index`**`(`*`x`*`[, `*`start`*`[, `*`end`*`]])`
Devolve o índice base-zero do primeiro item cujo valor é igual a *x*, levantando `ValueError` se este valor não existe.

Os argumentos opcionais *start* e *end* são interpretados como nas notações de fatiamento e são usados para limitar a busca para uma subsequência específica da lista. O índice retornado é calculado relativo ao começo da sequência inteira e não referente ao argumento *start*.

`list.`**`count`**`(`*`x`*`)`
Devolve o número de vezes em que *x* aparece na lista.

`list.`**`sort`**`(*, `*`key=None, reverse=False`*`)`
Ordena os itens na lista (os argumentos podem ser usados para personalizar a ordenação, veja a função `sorted()` para maiores explicações).

`list.`**`reverse`**`()`
Inverte a ordem dos elementos na lista.

`list.`**`copy`**`()`

Um exemplo que usa a maior parte dos métodos das listas:

```
>>> frutas = ['laranja', 'maçã', 'pera', 'banana', 'kiwi', 'maçã', 'banana']
>>> frutas.count('maçã')
2
>>> frutas.count('tangerina')
0
>>> frutas.index('banana')
3
>>> frutas.index('banana', 4) # Encontra a próxima banana iniciando da posição 4
6
>>> frutas.reverse()
>>> frutas
['banana', 'maçã', 'kiwi', 'banana', 'pera', 'maçã', 'laranja']
>>> frutas.append('uva')
>>> frutas
['banana', 'maçã', 'kiwi', 'banana', 'pera', 'maçã', 'laranja', 'uva']
>>> frutas.sort()
>>> frutas
['maçã', 'maçã', 'banana', 'banana', 'uva', 'kiwi', 'laranja', 'pera']
>>> frutas.pop()
'pera'
```

Você pode ter percebido que métodos como `insert`, `remove` ou `sort`, que apenas modificam a lista, não têm valor de retorno impresso – eles retornam o `None` padrão. [1] Isto é um princípio de design para todas as estruturas de dados mutáveis em Python.

Outro aspecto que você pode notar é que nem todos os dados podem ser classificados ou comparados. Por exemplo, `[None, 'hello', 10]` não é ordenável porque os inteiros não podem ser comparados a strings e `None` não pode ser comparado a outros tipos. Além disso, há alguns tipos que não têm uma relação de ordenação definida. Por exemplo, `3+4j < 5+7j` não é uma comparação válida.

5.1.1. 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 explícito. 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]
```

5.1.2. Usando listas como filas

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(["Erik", "João", "Miguel"])
>>> fila.append("Tiago")            # Tiago chega
>>> fila.append("George")           # George chega
>>> fila.popleft()                  # O primeiro a chegar agora sai
'Erik'
>>> fila.popleft()                  # O segundo a chegar agora sai
'João'
>>> fila                            # A fila restante na ordem de chegada
deque(['Miguel', 'Tiago', 'George'])
```

5.1.3. Compreensões de lista

Compreensões de lista fornece uma maneira concisa de criar uma lista. Aplicações comuns são criar novas listas onde cada elemento é o resultado de alguma operação aplicada a cada elemento de outra sequência ou iterável, ou criar uma sub-sequência de elementos que satisfaçam uma certa condição. (N.d.T. o termo original em inglês é *list comprehensions*, muito utilizado no Brasil; também se usa a abreviação *listcomp*).

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]
```

Note que isto cria (ou sobrescreve) uma variável chamada `x` que ainda existe após o término do laço. Podemos calcular a lista dos quadrados sem qualquer efeito colateral usando:

```
quadrados = list(map(lambda x: x**2, range(10)))
```

ou, de maneira equivalente:

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

que é mais conciso e legível.

Um compreensão de lista consiste de 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 nova lista resultante da avaliação da expressão no contexto das cláusulas `for` e `if`. Por exemplo, essa compreensão combina os elementos de duas listas se eles forem 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)]
```

e é equivalente a:

```
>>> combos = []
>>> for x in [1,2,3]:
```

```
...                 combos.append((x, y))
...
>>> combos
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note como a ordem das instruções `for` e `if` é a mesma em ambos os trechos.

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

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # cria uma nova lista com os valores dobrados
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtra a lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplica uma função para todos os elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # chama um método em cada elemento
>>> frutafresca = ['  banana', '  baga-de-logan ', 'maracujá  ']
>>> [arma.strip() for arma in frutafresca]
['banana', 'baga-de-logan', 'maracujá']
>>> # cria uma lista de tuplas de 2 elementos como (número, 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 entre parênteses, do contrário um erro é levantado
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # achatamento de uma lista usando uma compreensão de lista 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]
```

Compreensões de lista podem conter expressões complexas e funções aninhadas:

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

5.1.4. Compreensões de lista aninhadas

A expressão inicial em uma compreensão de lista pode ser qualquer expressão arbitrária, incluindo outra compreensão de lista.

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 compreensão de lista abaixo transpõe as linhas e colunas:

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

```
>>> transposta = []
>>> for i in range(4):
...     transposta.append([row[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(4):
...     # as 3 linhas a seguir implementam uma compreensão de lista 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 instruções complexas. A função `zip()` resolve muito bem este caso de uso:

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

Veja Desempacotando listas de argumentos para entender o uso do asterisco neste exemplo.

5.2. A instrução `del`

Existe uma maneira de remover um item de uma lista usando seu índice no lugar do seu valor: a instrução `del`. Ele difere do método `pop()` que devolve um valor. A instrução `del` pode também ser utilizada para remover fatias de uma lista ou limpar a lista inteira (que fizemos antes por atribuição de 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
```

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 a instrução `del` mais tarde.

5.3. Tuplas e Sequências

pos de sequências podem ser adicionados. Existe ainda um outro tipo de sequência padrão na linguagem: a *tupla*.

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

```
>>> t = 12345, 54321, 'olá!'
>>> t[0]
12345
>>> t
(12345, 54321, 'olá!')
>>> # Tuplas pode ser aninhadas:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'olá!'), (1, 2, 3, 4, 5))
>>> # Tuplas são imutáveis:
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # mas elas podem conter objetos mutáveis:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

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 de uma expressão maior). Não é possível atribuir itens individuais de uma tupla, contudo é possível criar tuplas que contenham objetos mutáveis, como listas.

Apesar de tuplas serem similares a listas, elas são frequentemente utilizadas em situações diferentes e com propósitos distintos. Tuplas são imutáveis, e usualmente contém uma sequência heterogênea de elementos que são acessados via desempacotamento (ver a seguir nessa seção) ou índice (ou mesmo por um atributo no caso de `namedtuples`). Listas são mutáveis, e seus elementos geralmente são homogêneos e são acessados iterando sobre a lista.

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 um 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. Por exemplo:

```
>>> vazio = ()
>>> singleton = 'olá', # <!- note a linha ao final vazia >>> len(vazio)
0
>>> len(singleton)
1
>>> singleton
('olá',)
```

A instrução `t = 12345, 54321, 'bom dia!'` é um exemplo de *empacotamento de tupla*: os valores `12345`, `54321` e `'bom dia!'` são empacotados em uma tupla. A operação inversa também é possível:

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

Isso é chamado, apropriadamente, de *desempacotamento de sequência* e funciona para qualquer sequência no lado direito. O desempacotamento de sequência requer que haja tantas variáveis no lado esquerdo do sinal de igual, quanto existem de elementos na sequência. Observe que a atribuição múltipla é, na verdade, apenas uma combinação de empacotamento de tupla e desempacotamento de sequência.

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 conjuntos 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.

Chaves ou a função `set()` podem ser usados para criar conjuntos. Note: para criar um conjunto vazio você precisa usar `set()`, não `{}`; este último cria um dicionário vazio, uma estrutura de dados que discutiremos na próxima seção.

Uma pequena demonstração:

```
>>> cesta = {'maçã', 'laranja', 'maçã', 'pera', 'laranja', 'banana'}
>>> print(cesta)                      # mostra que itens duplicados foram removidos
{'laranja', 'banana', 'pera', 'maçã'}
>>> 'laranja' in cesta                # teste de pertinência rápido
True
>>> 'crabgrass' in cesta
False

>>> # Demonstra operações de conjunto em letras únicas de duas palavras
>>>
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # letras únicas em a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                             # letras em a, mas não em b
{'r', 'd', 'b'}
>>> a | b                             # letras em a ou em b ou ambos
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                             # letras em ambos a e b
{'a', 'c'}
>>> a ^ b                             # letras em a ou b, mas não em ambos
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Da mesma forma que compreensão de listas, compreensões de conjunto também são suportadas:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. Dicionários

Outra estrutura de dados muito útil embutida em Python é o *dicionário*, cujo tipo é `dict` (ver Tipo mapeamento — 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 podem ser modificadas *internamente* 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 `{}`.

será substituído pelo novo. Se tentar recuperar um valor usando uma chave inexistente, será gerado um erro.

Executar `list(d)` em um dicionário devolve a lista de todas as chaves presentes no dicionário, na ordem de inserção (se desejar ordená-las basta usar a função `sorted(d)`). 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
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

O construtor `dict()` produz dicionários diretamente de sequências de pares chave-valor:

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

Além disso, as compreensões de dicionários podem ser usadas para criar dicionários a partir de expressões arbitrárias de chave e valor:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

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, 'guido': 4127, 'jack': 4098}
```

5.6. Técnicas de iteração

Ao iterar sobre dicionários, a chave e o valor correspondente podem ser obtidos simultaneamente usando o método `items()`.

```
>>> cavaleiros = {'gallahad': 'o puro', 'robin': 'o bravo'}
>>> for k, v in cavaleiros.items():
...     print(k, v)
...
gallahad o puro
robin o bravo
```

Ao iterar sobre sequências, a posição e o valor correspondente podem ser obtidos simultaneamente usando a função `enumerate()`.

```
...
0 jogo
1 da
2 velha
```

Para percorrer duas ou mais sequências ao mesmo tempo, as entradas podem ser pareadas com a função `zip()`.

```
>>> perguntas = ['Nome', 'Missão', 'Cor favorita']
>>> respostas = ['Lancelot', 'o santo graal', 'azul']
>>> for q, a in zip(perguntas, respostas):
...     print('{0}? É {1}.'.format(q, a))
...
Nome? É Lancelot.
Missão? É o santo graal.
Cor favorita? É azul.
```

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

```
>>> for i in reversed(range(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 = ['maçã', 'laranja', 'maçã', 'pera', 'laranja', 'banana']
>>> for i in sorted(cesta):
...     print(i)
...
maçã
maçã
banana
laranja
laranja
pera
```

Usar `set()` em uma sequência elimina elementos duplicados. O uso de `sorted()` em combinação com `set()` sobre uma sequência é uma maneira idiomática de fazer um loop sobre elementos exclusivos da sequência na ordem de classificação.

```
>>> cesta = ['maçã', 'laranja', 'maçã', 'pera', 'laranja', 'banana']
>>> for i in sorted(cesta):
...     print(i)
...
maçã
maçã
banana
laranja
laranja
pera
```

Às vezes é tentador alterar uma lista enquanto você itera sobre ela; porém, costuma ser mais simples e seguro criar uma nova lista.

```
>>> dados_filtrados = []
>>> for valor in dados_brutos:
...     if not math.isnan(value):
...         dados_filtrados.append(valor)
...
>>> dados_filtrados
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. Mais sobre condições

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

Os operadores de comparação `in` e `not in` fazem testes de inclusão que determinam se um valor está (ou não está) em um contêiner. Os operadores `is` e `is not` comparam se dois objetos são realmente o mesmo objeto. Todos os operadores de comparação possuem a mesma prioridade, que é menor 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 *curto-circuito*: seus argumentos são avaliados da esquerda para a direita, e a avaliação encerra 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 curto-circuito 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 no Python, ao contrário de C, a atribuição dentro de expressões deve ser feita explicitamente com o operador morsa `:=`. Isso evita uma classe comum de problemas encontrados nos programas C: digitar `=` em uma expressão quando `==` era o planejado.

5.8. 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 codificação Unicode para definir a ordenação. Alguns exemplos de comparações entre sequências do mesmo tipo:

```
'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)
```

Note que comparar objetos de tipos diferentes com `<` ou `>` é permitido desde que os objetos possuam os métodos de comparação apropriados. Por exemplo, tipos numéricos mistos são comparados de acordo com os seus valores numéricos, portanto 0 é igual a 0.0, etc. Em caso contrário, ao invés de fornecer uma ordenação arbitrária, o interpretador levantará um `TypeError`.

Notas de rodapé

[1] Outras linguagens podem retornar o objeto modificado, o que permite encadeamento de métodos, como `d->insert("a")->remove("b")->sort();`.