CAPÍTULO 05

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. Equivalente a `a[len(a):] = [x]`.

`list.`**`extend`**(*iterable*)

Prolonga a lista, adicionando no fim todos os elementos do argumento *iterable* passado como parâmetro. Equivalente 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 um item em uma dada posição na lista e o retorna. Se nenhum índice é especificado, `a.pop()` remove e devolve o último item da lista. (Os colchetes ao redor do *i* na demonstração do método indica que o parâmetro é opcional, e não que é necessário escrever estes colchetes ao chamar o método. Você verá este tipo de notação frequentemente na Biblioteca de Referência Python.)

`list.`**`clear`**()

Remove todos os itens de uma lista. Equivalente 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`**()

Devolve uma cópia rasa da lista. Equivalente a `a[:]`.

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

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi',
'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)   # Find next banana starting a
position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple',
'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple',
'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi',
'orange', 'pear']
>>> fruits.pop()
'pear'
```

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.

Outra coisa que você deve estar atento é que nem todos os dados podem ser ordenados ou comparados. Por exemplo, , `[None, 'hello', 10]` não podem ser ordenados, pois inteiros não podem ser comparados a strings e *None* não pode ser comparado a nenhum outro tipo. Além disso, existem alguns tipos de dados que não possuem uma relação de ordem 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. Por exemplo:

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

5.1.2. 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
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
```

```
>>> queue.popleft()                 # The first to arrive now
leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now
leaves
'John'
>>> queue                           # Remaining queue in order
of arrival
deque(['Michael', 'Terry', 'Graham'])
```

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 subsequência 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:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[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:

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

ou, de maneira equivalente:

```
squares = [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:

```
>>> 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 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]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit
']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is
raised
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two '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:

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

A compreensão de lista abaixo transpõe as linhas e colunas:

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

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

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

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

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
```

```
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[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:

```
>>> list(zip(*matrix))
[(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 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
```

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

Vimos que listas e strings têm muitas propriedades em comum, como indexação e operações de fatiamento. Elas são dois exemplos de *sequências* (veja Tipos sequências — list, tuple, range). 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*.

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

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... 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:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

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 *sequência de desempacotamento* 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.

5.4. 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 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:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange',
'banana'}
>>> print(basket)                      # show that duplicates
have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
```

```
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two
words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not
in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or
both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and
b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but
not both
{'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 `{}`.

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.

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()`.

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

Ao iterar sobre sequências, a posição 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 ao mesmo tempo, as entradas podem ser pareadas 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.
```

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.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for i in sorted(basket):
...     print(i)
...
apple
apple
banana
orange
orange
pear
```

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.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange',
'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

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

```
>>> import math
>>>  raw_data  =  [56.2,  float('NaN'),  51.7,  55.3,  52.5,
float('NaN'), 47.8]
>>> filtered_data = []
```

```
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7. 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. 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 *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:

```
(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)
```

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`.

ATIVIDADES

1. Faça um programa que leia um número indeterminado de valores, correspondentes a notas, encerrando a entrada de dados quando for informado um valor igual a -1 (que não deve ser armazenado). Após esta entrada de dados, faça:
    a. Mostre a quantidade de valores que foram lidos;
    b. Exiba todos os valores na ordem em que foram informados, um ao lado do outro;

c. Exiba todos os valores na ordem inversa à que foram informados, um abaixo do outro;
d. Calcule e mostre a soma dos valores;
e. Calcule e mostre a média dos valores;
f. Calcule e mostre a quantidade de valores acima da média calculada;
g. Calcule e mostre a quantidade de valores abaixo de sete;
h. Encerre o programa com uma mensagem;

2. Uma grande emissora de televisão quer fazer uma enquete entre os seus telespectadores para saber qual o melhor jogador após cada jogo. Para isto, faz-se necessário o desenvolvimento de um programa, que será utilizado pelas telefonistas, para a computação dos votos. Sua equipe foi contratada para desenvolver este programa, utilizando a linguagem de programação C++. Para computar cada voto, a telefonista digitará um número, entre 1 e 23, correspondente ao número da camisa do jogador. Um número de jogador igual zero, indica que a votação foi encerrada. Se um número inválido for digitado, o programa deve ignorá-lo, mostrando uma breve mensagem de aviso, e voltando a pedir outro número. Após o final da votação, o programa deverá exibir:
    a. O total de votos computados;
    b. Os númeos e respectivos votos de todos os jogadores que receberam votos;
    c. O percentual de votos de cada um destes jogadores;
    d. O número do jogador escolhido como o melhor jogador da partida, juntamente com o número de votos e o percentual de votos dados a ele.
        1. Observe que os votos inválidos e o zero final não devem ser computados como votos. O resultado aparece ordenado pelo número do jogador. O programa deve fazer uso de arrays. O programa deverá executar o cálculo do percentual de cada jogador através de uma função. Esta função receberá dois parâmetros: o número de votos de um jogador e o total de votos. A função calculará o percentual e retornará o valor calculado. Abaixo segue uma tela de

exemplo. O disposição das informações deve ser o mais próxima possível ao exemplo. Os dados são fictícios e podem mudar a cada execução do programa. Ao final, o programa deve ainda gravar os dados referentes ao resultado da votação em um arquivo texto no disco, obedecendo a mesma disposição apresentada na tela.

```
Enquete: Quem foi o melhor jogador?

Número do jogador (0=fim): 9
Número do jogador (0=fim): 10
Número do jogador (0=fim): 9
Número do jogador (0=fim): 10
Número do jogador (0=fim): 11
Número do jogador (0=fim): 10
Número do jogador (0=fim): 50
Informe um valor entre 1 e 23 ou 0 para sair!
Número do jogador (0=fim): 9
Número do jogador (0=fim): 9
Número do jogador (0=fim): 0

Resultado da votação:

Foram computados 8 votos.

Jogador Votos           %
9               4               50,0%
10              3               37,5%
11              1               12,5%
O melhor jogador foi o número 9, com 4 votos, correspondendo a 50%
do total de votos.
```