
HowTo - Ordenação

Release 3.9.18

**Guido van Rossum
and the Python development team**

fevereiro 06, 2024

Python Software Foundation
Email: docs@python.org

Sumário

1	Básico de Ordenação	2
2	Funções Chave	2
3	Funções do Módulo Operator	3
4	Ascendente e Descendente	3
5	Estabilidade de Ordenação e Ordenações Complexas	4
6	A velha maneira utilizando Decorate-Sort-Undecorate	4
7	O método antigo utilizando o parâmetro <i>cmp</i>	5
8	Ímpares e extremidades	6

Autor Andrew Dalke e Raymond Hettinger

Versão 0.1

As listas em Python possuem um método embutido `list.sort()` que modifica a lista em si. Há também a função embutida `sorted()` que constrói uma nova lista ordenada à partir de um iterável.

Neste documento, exploramos várias técnicas para ordenar dados utilizando Python.

1 Básico de Ordenação

Uma simples ordenação ascendente é muito fácil: apenas chame a função `sorted()`. Isso retornará uma nova lista ordenada:

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

Você também pode utilizar o método `list.sort()`. Isso modifica a lista em si (e retorna `None` para evitar confusão). Usualmente este método é menos conveniente que a função `sorted()` - mas se você não precisará da lista original, esta maneira é levemente mais eficiente.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Outra diferença é que o método `list.sort()` é aplicável apenas às listas. Em contrapartida, a função `sorted()` aceita qualquer iterável.

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
[1, 2, 3, 4, 5]
```

2 Funções Chave

Tanto o método `list.sort()` quanto a função `sorted()` possuem um parâmetro `key` que especifica uma função (ou outro chamável) a ser chamada para cada elemento da lista antes de ser realizada a comparação.

Por exemplo, aqui há uma comparação case-insensitive de strings.

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

O valor do parâmetro `key` deve ser uma função (ou outro chamável) que recebe um único argumento e retorna uma chave à ser utilizada com o propósito de ordenação. Esta técnica é rápida porque a função chave é chamada exatamente uma vez para cada entrada de registro.

Uma padrão comum é ordenar objetos complexos utilizando algum índice do objeto como chave. Por exemplo:

```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2])    # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

A mesma técnica funciona com objetos que possuem atributos nomeados. Por exemplo:

```
>>> class Student:
...     def __init__(self, name, grade, age):
...         self.name = name
...         self.grade = grade
...         self.age = age
```

(continua na próxima página)

```
...     def __repr__(self):
...         return repr((self.name, self.grade, self.age))
```

```
>>> student_objects = [
...     Student('john', 'A', 15),
...     Student('jane', 'B', 12),
...     Student('dave', 'B', 10),
... ]
>>> sorted(student_objects, key=lambda student: student.age)    # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

3 Funções do Módulo Operator

O padrão de chave-função mostrado acima é muito comum, por isso, Python provê funções convenientes para tornar as funções de acesso mais fáceis e rápidas. O módulo `operator` tem as funções `itemgetter()`, `attrgetter()`, e `methodcaller()`

Usando estas funções, os exemplos acima se tornam mais simples e mais rápidos:

```
>>> from operator import itemgetter, attrgetter
```

```
>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

```
>>> sorted(student_objects, key=attrgetter('age'))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

As funções do módulo `operator` permite múltiplos níveis de ordenação. Por exemplo, ordenar por *grade* e então por *age*:

```
>>> sorted(student_tuples, key=itemgetter(1,2))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

```
>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

4 Ascendente e Descendente

Tanto o método `list.sort()` quanto a função `sorted()` aceitam um valor booleano para o parâmetro *reverse*. Essa flag é utilizada para ordenações descendentes. Por exemplo, para retornar os dados de estudantes pela ordem inversa de *age*:

```
>>> sorted(student_tuples, key=itemgetter(2), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

```
>>> sorted(student_objects, key=attrgetter('age'), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

5 Estabilidade de Ordenação e Ordenações Complexas

Ordenações são garantidas de serem *estáveis*. Isso significa que quando múltiplos registros possuem a mesma chave, eles terão sua ordem original preservada.

```
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> sorted(data, key=itemgetter(0))
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Observe como os dois registros de *blue* permanecem em sua ordem original de forma que *('blue', 1)* é garantido de preceder *('blue', 2)*.

Esta maravilhosa propriedade permite que você construa ordenações complexas em uma série de passos de ordenação. Por exemplo, para ordenar os registros de estudante por ordem descendente de *grade* e então ascendente de *age*, primeiro ordene *age* e depois ordene novamente utilizando *grade*:

```
>>> s = sorted(student_objects, key=attrgetter('age'))      # sort on secondary key
>>> sorted(s, key=attrgetter('grade'), reverse=True)       # now sort on primary key,
↪descending
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

Isso pode ser abstrato no caso das funções encapsuladoras que podem receber uma lista e uma tupla com o campos e então ordená-los em múltiplos passos.

```
>>> def multisort(xs, specs):
...     for key, reverse in reversed(specs):
...         xs.sort(key=attrgetter(key), reverse=reverse)
...     return xs
```

```
>>> multisort(list(student_objects), (('grade', True), ('age', False)))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

O algoritmo *Timsort* utilizado no Python realiza múltiplas ordenações de maneira eficiente, pois se aproveita de qualquer ordenação já presente no conjunto de dados.

6 A velha maneira utilizando Decorate-Sort-Undecorate

Esse item idiomático, chamado de Decorate-Sort-Undecorate, é realizado em três passos:

- Primeiro, a lista inicial é decorada com novos valores que controlarão a ordem em que ocorrerá a ordenação
- Segundo, a lista decorada é ordenada.
- Finalmente, os valores decorados são removidos, criando uma lista que contém apenas os valores iniciais na nova ordenação.

Por exemplo, para ordenar os dados dos estudantes por *grade* usando a abordagem DSU:

```
>>> decorated = [(student.grade, i, student) for i, student in enumerate(student_
↪objects)]
>>> decorated.sort()
>>> [student for grade, i, student in decorated]          # undecorate
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

Esse padrão idiomático funciona porque tuplas são comparadas lexicograficamente; os primeiros itens são comparados; se eles são semelhantes, então os segundos itens são comparados e assim sucessivamente.

Não é estritamente necessário incluir o índice *i* em todos os casos de listas decoradas, mas fazer assim traz dois benefícios:

- A ordenação é estável - se dois itens tem a mesma chave, suas ordens serão preservadas na lista ordenada
- Os itens originais não precisarão ser comparados porque a ordenação de tuplas decoradas será determinada por no máximo os primeiros dois itens. Então, por exemplo, a lista original poderia conter números complexos que não poderão ser ordenados diretamente.

Outro nome para este padrão idiomático é [Schwartzian transform](#) de Randal L. Schwartz, que popularizou isto entre os programadores Perl.

Agora que a ordenação do Python prevê funções-chave, essa técnica não se faz mais necessária.

7 O método antigo utilizando o parâmetro *cmp*

Muitos construtores apresentados neste HOWTO assumem o uso do Python 2.4 ou superior. Antes disso, não havia a função embutida `sorted()` e o método `list.sort()` não recebia os argumentos nomeados. Apesar disso, todas as versões do Py2.x suportam o parâmetro *cmp* para lidar com a função de comparação especificada pelo usuário.

No Py3.0, o parâmetro *cmp* foi removido totalmente (como parte de um grande esforço para simplificar e unificar a linguagem, eliminando o conflito entre comparações enriquecidas e o método mágico `__cmp__()`)

No Py2.x, ordenação permite uma função opcional que pode ser invocada para fazer comparações. Esta função pode receber dois argumentos para serem comparados e retorna um valor negativo para o menor-que, retorna zero se eles são iguais, ou retorna um valor positivo para o maior-que. Por exemplo, podemos fazer:

```
>>> def numeric_compare(x, y):
...     return x - y
>>> sorted([5, 2, 4, 1, 3], cmp=numeric_compare)
[1, 2, 3, 4, 5]
```

Ou podemos inverter a ordem de comparação com:

```
>>> def reverse_numeric(x, y):
...     return y - x
>>> sorted([5, 2, 4, 1, 3], cmp=reverse_numeric)
[5, 4, 3, 2, 1]
```

Quando migrando o código do Python 2.x para o 3.x, pode surgir a situação em que há o usuário suprimindo a função de comparação e é necessário converter isso em uma função chave. O seguinte empacotamento torna isso fácil de fazer:

```
def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'
    class K:
        def __init__(self, obj, *args):
            self.obj = obj
        def __lt__(self, other):
            return mycmp(self.obj, other.obj) < 0
        def __gt__(self, other):
            return mycmp(self.obj, other.obj) > 0
        def __eq__(self, other):
            return mycmp(self.obj, other.obj) == 0
        def __le__(self, other):
            return mycmp(self.obj, other.obj) <= 0
        def __ge__(self, other):
            return mycmp(self.obj, other.obj) >= 0
        def __ne__(self, other):
```

(continua na próxima página)

```

        return mycmp(self.obj, other.obj) != 0
    return K

```

Para converter a função chave, apenas empacote a velha função de comparação:

```

>>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
[5, 4, 3, 2, 1]

```

No Python 3.2, a função `functools.cmp_to_key()` foi adicionada ao módulo `functools` na biblioteca padrão.

8 Ímpares e extremidades

- Para ordenação com reconhecimento de localidade, use `locale.strxfrm()` para uma função chave ou `locale.strcoll()` para uma função de comparação.
- O parâmetro `reverse` ainda mantém a estabilidade da ordenação (para que os registros com chaves iguais mantenham a ordem original). Curiosamente, esse efeito pode ser simulado sem o parâmetro usando a função embutida `reversed()` duas vezes:

```

>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> standard_way = sorted(data, key=itemgetter(0), reverse=True)
>>> double_reversed = list(reversed(sorted(reversed(data), key=itemgetter(0))))
>>> assert standard_way == double_reversed
>>> standard_way
[('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]

```

- The sort routines use `<` when making comparisons between two objects. So, it is easy to add a standard sort order to a class by defining an `__lt__()` method:

```

>>> Student.__lt__ = lambda self, other: self.age < other.age
>>> sorted(student_objects)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

```

However, note that `<` can fall back to using `__gt__()` if `__lt__()` is not implemented (see `object.__lt__()`).

- As funções principais não precisam depender diretamente dos objetos que estão sendo ordenados. Uma função chave também pode acessar recursos externos. Por exemplo, se as notas dos alunos estiverem armazenadas em um dicionário, elas poderão ser usadas para ordenar uma lista separada de nomes de alunos:

```

>>> students = ['dave', 'john', 'jane']
>>> newgrades = {'john': 'F', 'jane': 'A', 'dave': 'C'}
>>> sorted(students, key=newgrades.__getitem__)
['jane', 'dave', 'john']

```