Sorting Techniques

Release 3.13.0b3

Guido van Rossum and the Python development team

julho 07, 2024

Python Software Foundation
Email: docs@python.org

Sumário

Autor
Andrew Dalke e Raymond Hettinger

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 ordenação ascendente simples é muito fácil: apenas chame a função `sorted()`. Retorna 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.casefold)
['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
...     def __repr__(self):
...         return repr((self.name, self.grade, self.age))

>>> student_objects = [
...     Student('john', 'A', 15),
```

(continua na próxima página)

(continuação da página anterior)

```
...     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)]
```

Objects with named attributes can be made by a regular class as shown above, or they can be instances of `dataclass` or a named tuple.

3 Operator Module Functions and Partial Function Evaluation

The key function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The `operator` module has `itemgetter()`, `attrgetter()`, and a `methodcaller()` function.

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)]
```

The `functools` module provides another helpful tool for making key-functions. The `partial()` function can reduce the arity of a multi-argument function making it suitable for use as a key-function.

```
>>> from functools import partial
>>> from unicodedata import normalize

>>> names = 'Zoë Åbjørn Núñez Élana Zeke Abe Nubia Eloise'.split()

>>> sorted(names, key=partial(normalize, 'NFD'))
['Abe', 'Åbjørn', 'Eloise', 'Élana', 'Nubia', 'Núñez', 'Zeke', 'Zoë']

>>> sorted(names, key=partial(normalize, 'NFC'))
['Abe', 'Eloise', 'Nubia', 'Núñez', 'Zeke', 'Zoë', 'Åbjørn', 'Élana']
```

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 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 Comparison Functions

Unlike key functions that return an absolute value for sorting, a comparison function computes the relative ordering for two inputs.

For example, a balance scale compares two samples giving a relative ordering: lighter, equal, or heavier. Likewise, a comparison function such as `cmp(a, b)` will return a negative value for less-than, zero if the inputs are equal, or a positive value for greater-than.

It is common to encounter comparison functions when translating algorithms from other languages. Also, some libraries provide comparison functions as part of their API. For example, `locale.strcoll()` is a comparison function.

To accommodate those situations, Python provides `functools.cmp_to_key` to wrap the comparison function to make it usable as a key function:

```
sorted(words, key=cmp_to_key(strcoll))  # locale-aware sort order
```

8 Curiosidades e conclusões

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function. This is necessary because "alphabetical" sort orderings can vary across cultures even if the underlying alphabet is the same.
- 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__()` for details on the mechanics). To avoid surprises, **PEP 8** recommends that all six comparison methods be implemented. The `total_ordering()` decorator is provided to make that task easier.

- 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']
```

9 Partial Sorts

Some applications require only some of the data to be ordered. The standard library provides several tools that do less work than a full sort:

- `min()` and `max()` return the smallest and largest values, respectively. These functions make a single pass over the input data and require almost no auxiliary memory.
- `heapq.nsmallest()` and `heapq.nlargest()` return the *n* smallest and largest values, respectively. These functions make a single pass over the data keeping only *n* elements in memory at a time. For values of *n* that are small relative to the number of inputs, these functions make far fewer comparisons than a full sort.
- `heapq.heappush()` and `heapq.heappop()` create and maintain a partially sorted arrangement of data that keeps the smallest element at position `0`. These functions are suitable for implementing priority queues which are commonly used for task scheduling.

Índice

P