
排序指南

发布 3.12.0a0

Guido van Rossum and the Python development team

十月 24, 2022

Python Software Foundation
Email: docs@python.org

Contents

1 基本排序	1
2 关键函数	2
3 Operator 模块函数	3
4 升序和降序	3
5 排序稳定性和排序复杂度	3
6 Decorate-Sort-Undecorate	4
7 Comparison Functions	4
8 Odds and Ends	5

作者 Andrew Dalke 和 Raymond Hettinger

发布版本 0.1

Python 列表有一个内置的 `list.sort()` 方法可以直接修改列表。还有一个 `sorted()` 内置函数，它会从一个可迭代对象构建一个新的排序列表。

在本文档中，我们将探索使用 Python 对数据进行排序的各种技术。

1 基本排序

简单的升序排序非常简单：只需调用 `sorted()` 函数。它返回一个新的排序后列表：

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

你也可以使用 `list.sort()` 方法，它会直接修改原列表（并返回 `None` 以避免混淆），通常来说它不如 `sorted()` 方便——但如果你不需要原列表，它会更有效率。

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

另外一个区别是，`list.sort()` 方法只是为列表定义的，而 `sorted()` 函数可以接受任何可迭代对象。

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

2 关键函数

`list.sort()` 和 `sorted()` 都有一个 `key` 形参用来指定在进行比较前要在每个列表元素上调用的函数 (或其他可调用对象)。

例如，下面是一个不区分大小写的字符串比较：

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

`key` 形参的值应该是个函数 (或其他可调用对象)，它接受一个参数并返回一个用于排序的键。这种机制速度很快，因为对于每个输入记录只会调用一次键函数。

一种常见的模式是使用对象的一些索引作为键对复杂对象进行排序。例如：

```
>>> 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)]
```

同样的技术也适用于具有命名属性的对象。例如：

```
>>> 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),
...     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 Operator 模块函数

上面显示的键函数模式非常常见, 因此 Python 提供了便利功能, 使访问器功能更容易, 更快捷。operator 模块有 itemgetter()、attrgetter() 和 methodcaller() 函数。

使用这些函数, 上述示例变得更简单, 更快捷:

```
>>> 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)]
```

Operator 模块功能允许多级排序。例如, 按 *grade* 排序, 然后按 *age* 排序:

```
>>> 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 升序和降序

list.sort() 和 sorted() 接受布尔值的 *reverse* 参数。这用于标记降序排序。例如, 要以反向 *age* 顺序获取学生数据:

```
>>> 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 排序稳定性和排序复杂度

排序保证是 *稳定* 的。这意味着当多个记录具有相同的键值时, 将保留其原始顺序。

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

注意 *blue* 的两个记录如何保留它们的原始顺序, 以便 ('blue', 1) 保证在 ('blue', 2) 之前。

这个美妙的属性允许你在一系列排序步骤中构建复杂的排序。例如, 要按 *grade* 降序然后 *age* 升序对学生数据进行排序, 请先 *age* 排序, 然后再使用 *grade* 排序:

```
>>> 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)]
```

这可以被抽象为一个包装函数, 该函数能接受一个列表以及字段和顺序的元组, 以对它们进行多重排序。

```
>>> 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)]
```

Python 中使用的 [Timsort](#) 算法可以有效地进行多种排序，因为它可以利用数据集中已存在的任何排序。

6 Decorate-Sort-Undecorate

这三个步骤被称为 Decorate-Sort-Undecorate：

- 首先，初始列表使用控制排序顺序的新值进行修饰。
- 然后，装饰列表已排序。
- 最后，删除装饰，创建一个仅包含新排序中初始值的列表。

例如，要使用 DSU 方法按 *grade* 对学生数据进行排序：

```
>>> 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)]
```

这方法有效是因为元组按字典顺序进行比较，先比较第一项；如果它们相同则比较第二个项目，依此类推。

不一定在所有情况下都要在装饰列表中包含索引 *i*，但包含它有两个好处：

- 排序是稳定的——如果两个项具有相同的键，它们的顺序将保留在排序列表中。
- 原始项目不必具有可比性，因为装饰元组的排序最多由前两项决定。因此，例如原始列表可能包含无法直接排序的复数。

这个方法的另一个名字是 Randal L. Schwartz 在 Perl 程序员中推广的 [Schwartzian transform](#)。

既然 Python 排序提供了键函数，那么通常不需要这种技术。

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))
```

8 Odds and Ends

- 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.
- `reverse` 参数仍然保持排序稳定性（因此具有相等键的记录保留原始顺序）。有趣的是，通过使用内置的 `reversed()` 函数两次，可以在没有参数的情况下模拟该效果：

```
>>> 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)]
```

- 在对两个对象进行比较时，排序例程使用 `<`。因此，通过定义一个 `__lt__()` 方法，很容易为一个类添加一个标准的排序顺序。

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

然而，请注意，如果 `__gt__()` 没有实现，`<` 可以退回到使用 `__lt__()`（见 `object.__lt__()`）。

- 键函数不需要直接依赖于被排序的对象。键函数还可以访问外部资源。例如，如果学生成绩存储在字典中，则可以使用它们对单独的学生姓名列表进行排序：

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