
排序指南

发布 2.7.18

Guido van Rossum
and the Python development team

五月 20, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 基本排序	2
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	4
6 使用装饰-排序-去装饰的旧方法	4
7 使用 <i>cmp</i> 参数的旧方法	5
8 其它	6

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()` 函数即可。它会返回一个新的已排序列表。

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

You can also use the `list.sort()` method of a list. It modifies the list in-place (and returns `None` to avoid confusion). Usually it's less convenient than `sorted()` - but if you don't need the original list, it's slightly more efficient.

```
>>> 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

Starting with Python 2.4, both `list.sort()` and `sorted()` added a *key* parameter to specify a function to be called on each list element prior to making comparisons.

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 形参的值应该是一个函数，它接受一个参数并返回一个用于排序的键。这种技巧速度很快，因为对于每个输入记录只会调用一次 *key* 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

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 `operator.itemgetter()`, `operator.attrgetter()`, and starting in Python 2.5 an `operator.methodcaller()` function.

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)]
```

The `operator.methodcaller()` function makes method calls with fixed parameters for each object being sorted. For example, the `str.count()` method could be used to compute message priority by counting the number of exclamation marks in a message:

```
>>> from operator import methodcaller
>>> messages = ['critical!!!', 'hurry!', 'standby', 'immediate!!!']
>>> sorted(messages, key=methodcaller('count', '!'))
['standby', 'hurry!', 'immediate!!!', 'critical!!!']
```

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

Starting with Python 2.2, sorts are guaranteed to be *stable*. That means that when multiple records have the same key, their original order is preserved.

```
>>> 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)]
```

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 使用装饰-排序-去装饰的旧方法

这个三个步骤被称为 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*。

For large lists and lists where the comparison information is expensive to calculate, and Python versions before 2.4, DSU is likely to be the fastest way to sort the list. For 2.4 and later, key functions provide the same functionality.

7 使用 *cmp* 参数的旧方法

本 HOWTO 中给出的许多结构都假定为 Python 2.4 或更高版本。在此之前，没有内置 `sorted()`，`list.sort()` 也没有关键字参数。相反，所有 Py2.x 版本都支持 *cmp* 参数来处理用户指定的比较函数。

In Python 3, the *cmp* parameter was removed entirely (as part of a larger effort to simplify and unify the language, eliminating the conflict between rich comparisons and the `__cmp__()` magic method).

In Python 2, `sort()` allowed an optional function which can be called for doing the comparisons. That function should take two arguments to be compared and then return a negative value for less-than, return zero if they are equal, or return a positive value for greater-than. For example, we can do:

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

或者你可反转比较的顺序:

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

将代码从 Python 2.x 移植到 3.x 时，如果用户提供比较功能并且需要将其转换为键函数，则会出现这种情况。以下包装器使这很容易:

```
def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'
    class K(object):
        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):
            return mycmp(self.obj, other.obj) != 0
    return K
```

要转换为键函数，只需包装旧的比较函数:

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

In Python 2.7, the `functools.cmp_to_key()` function was added to the `functools` module.

8 其它

- 对于区域相关的排序，请使用 `locale.strxfrm()` 作为键函数，或者 `locale.strcoll()` 作为比较函数。
- The *reverse* parameter still maintains sort stability (so that records with equal keys retain their original order). Interestingly, that effect can be simulated without the parameter by using the builtin `reversed()` function twice:

```
>>> 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)]
```

- To create a standard sort order for a class, just add the appropriate rich comparison methods:

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

For general purpose comparisons, the recommended approach is to define all six rich comparison operators. The `functools.total_ordering()` class decorator makes this easy to implement.

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