如何排序

發🄵 *3.11.11*

Guido van Rossum and the Python development team

12 月 07, 2024

Python Software Foundation
Email: docs@python.org

Contents

作者
Andrew Dalke 和 Raymond Hettinger

發🄵版本
0.1

内置列表方法 `list.sort()` 原地修改列表，而内置函数 `sorted()` 由可迭代对象新建有序列表。

在此文件，我們使用 Python 進行各種方式排序資料

1 基礎排序

普通的升序排序非常容易：只需调用 sorted() 函数。它返回新有序列表：

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

亦可用 list.sort() 方法。它原地修改原列表（并返回 None 以避免混淆）。往往不如 sorted() 方便——但若不需原列表，用它会略高效些。

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

另一个区别是 list.sort() 方法只为列表定义，而 sorted() 函数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 键函数

list.sort() 和 sorted() 皆有 *key* 形参用以指定在比较前要对每个列表元素调用的函数（或其它可调用对象）。

例如，这是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 形参的值需为一元函数（或其它可调用对象），其返回值用于排序。这很快，因为键函数只需在输入的每个记录上调用恰好一次。

常见的模式是用对象的某一些索引作为键对复杂对象排序。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)]
```

同样的方法对于有具名属性的对象也适用。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 = [
```

(繼續下一頁)

(繼續上一頁)

```
...     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。

用了那些函数之后，前面的示例变得更简单，运行起来也更快：

```
>>> 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)]
```

运算符模块的函数可以用来作多级排序。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🄵與降🄵

`list.sort()` 和 `sorted()` 接受布尔形参 *reverse* 用于标记降序排序。例如，将学生数据按 *age* 倒序排序：

```
>>> 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 排序稳定性与复杂排序

排序保证 稳定：等键记录保持原始顺序。

```
>>> 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* 的两个记录是如何保序的：`('blue', 1)` 保证先于 `('blue', 2)`。

这个了不起的特性使得借助一系列排序步骤构建出复杂排序成为可能。例如，要按 *grade* 降序后 *age* 升序排序学生数据，只需先用 *age* 排序再用 *grade* 排序即可：

```
>>> 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)]
```

可抽象为包装函数，依据接收的一些字段序的元组对接收的列表做多趟排序。

```
>>> 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 中曾用的 Timsort 算法借助数据集中任何已有的有序性来高效进行多种排序。

6 装饰-排序-去装饰

装饰-排序-去装饰 (Decorate-Sort-Undecorate) 得名于它的三个步骤：

- 首先，用控制排序顺序的新值装饰初始列表。
- 其次，排序装饰后的列表。
- 最后，去除装饰即得按新顺序排列的初始值的列表。

例如，用 DSU 方法按 *grade* 排序学生数据：

```
>>> 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)]
```

这方法语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 比较函数

与返回一个用于排序的绝对值的键函数不同，比较函数是计算两个输入的相对排序。

例如，一个 天平 会比较两个样本并给出一个相对排序：较轻、相等或较重。类似地，一个比较函数如 `cmp(a, b)` 将返回一个负值表示小于，零表示相等，或是一个正值表示大于。

当从其他语言转写算法时经常会遇到比较函数。此外，某些库也提供了比较函数作为其 API 的组成部分。例如，`locale.strcoll()` 就是一个比较函数。

为了适应这些情况，Python 提供了 `functools.cmp_to_key` 用来包装比较函数使其可以作为键函数来使用:

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

8 杂项说明

- 对于可感知语言区域的排序，请使用 `locale.strxfrm()` 作为键函数或使用 `locale.strcoll()` 作为比较函数。因为在不同语言中即便字母表相同“字母”排列顺序也可能不同所以这样做是必要的。
- *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)]
```

- 排序例程在两个对象之间进行比较时使用 <。因此，通过定义一个 `__lt__()` 方法，就可以轻松地为类添加标准排序顺序:

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

  不过，请注意如果没有 `__lt__()`，则 < 可以退回到使用 `__gt__()` (参见 `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']
```