排序的技术

作者: Andrew Dalke 与 Raymond Hettinger

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。

在本文档中，我们将探索使用 Python 对数据进行排序的各种技术。

排序的基础知识

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]
```

键函数

`list.sort()` 方法以及 `sorted()`, `min()`, `max()`, `heapq.nsmallest()` 和 `heapq.nlargest()` 等函数都有一个 *key* 形参用以指定要在进行比较之前对每个列表元素调用的函数（或其它可调用对象）。

例如，下面是使用 `str.casefold()` 进行不区分大小写的字符串比较：

```
>>> sorted("This is a test string from Andrew".split(), key=str.casefold)
['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])   # 按年龄排序
[('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 = [
...     Student('john', 'A', 15),
...     Student('jane', 'B', 12),
...     Student('dave', 'B', 10),
... ]
>>> sorted(student_objects, key=lambda student: student.age)   # 按年龄排序
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

有具名属性的对象可像上面这样用一个常规的类来创建，亦可是 `dataclass` 实例或 named tuple。

运算符模块的函数与函数的偏求值

上述 key function 模式相当常见，为了让访问器函数更加好写好用，Python 提供了一些便捷函数。`operator` 模块里有 `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)]
```

另一个有助于创建键函数的工具位于 `functools` 模块。`partial()` 函数可以降低多元函数的 元数 使之适合做键函数。

```
>>> 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']
```

升序与降序

`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)]
```

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'))     # 根据次要键（年龄）排序
>>> sorted(s, key=attrgetter('grade'), reverse=True)       # 现在根据主要键（成绩）
[('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。

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_c
>>> decorated.sort()
>>> [student for grade, i, student in decorated]               # 取消装饰
[('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。

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))  # 基于地区的排序规则
```

不可排序类型和值的策略

在排序时可能出现多种涉及类型和值的问题。 下面是一些有助于解决问题的策略：

- 在排序之前将不可比较的输入类型转换为字符串。

```
>>> data = ['twelve', '11', 10]
>>> sorted(map(str, data))
['10', '11', 'twelve']
```

需要这样做是因为大多数跨类型比较都会引发 `TypeError`。

- 在排序之前移除特殊的值：

```
>>> from math import isnan
>>> from itertools import filterfalse
>>> data = [3.3, float('nan'), 1.1, 2.2]
>>> sorted(filterfalse(isnan, data))
[1.1, 2.2, 3.3]
```

这是必要的，因为 IEEE-754标准 规定，“每一个NaN都应该与任何事物进行无序比较，包括它自身。”

同样，`None` 也可以从数据集中剥离:

```
>>> data = [3.3, None, 1.1, 2.2]
>>> sorted(x for x in data if x is not None)
[1.1, 2.2, 3.3]
```

这是必需的，因为``None``与其他类型不具有可比性。

- 在排序之前将映射类型转换为已排序的项列表：

```
>>> data = [{'a': 1}, {'b': 2}]
>>> sorted(data, key=lambda d: sorted(d.items()))
[{'a': 1}, {'b': 2}]
```

这是必需的，因为字典到字典的比较会引发 `TypeError`。

- 在排序之前将集合类型转换为排序列表：

```
>>> data = [{'a', 'b', 'c'}, {'b', 'c', 'd'}]
>>> sorted(map(sorted, data))
[['a', 'b', 'c'], ['b', 'c', 'd']]
```

这是必需的，因为集合类型中包含的元素没有确定的顺序。例如,``list({'a', 'b'})``可以产生``['a', 'b']`` 或 `['b', 'a']`。

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 `<` 在 `__lt__()` 未被实现时可以回退为使用 `__gt__()` (请参阅 `object.__lt__()` 了解相关机制的细节)。 为避免意外，**PEP 8** 建议实现所有的六个比较方法。 `total_ordering()` 装饰器被提供用来令此任务更为容易。

- 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']
```

部分排序

有些应用程序只需要对部分数据进行排序。 标准库提供了几种工具可以执行比完整排序更轻量的任务:

- `min()` 和 `max()` 可分别返回最小和最大值。 这两个函数只需逐一检查输入数据而几乎不需要任何额外的内存。
- `heapq.nsmallest()` 和 `heapq.nlargest()` 可分别返回 *n* 个最小和最大的值。 这两个函数每次只需逐一检查数据并仅需在内存中保留 *n* 个元素。 对于相对于输入总数来说较小的 *n* 值来说，这两个函数将进行远少于完整排序的比较。
- `heapq.heappush()` 和 `heapq.heappop()` 会创建并维护一组部分排序的数据其中最小的元素将处在 `0` 位置上。 这两个函数很适合实现常用于任务调度的优先级队列。