
排序技法

發行 3.14.0a2

Guido van Rossum and the Python development team

11 月 23, 2024

Python Software Foundation
Email: docs@python.org

Contents

| | |
|---|---|
| 1 基礎排序 | 1 |
| 2 鍵函式 (key functions) | 2 |
| 3 Operator 模組函式以及部份函式 (partial function) 評估 | 3 |
| 4 升序與降序 | 3 |
| 5 排序穩定性與合併排序 | 3 |
| 6 裝飾-排序-移除裝飾 (decorate-sort-undecorate) | 4 |
| 7 比較函式 (comparison functions) | 4 |
| 8 Strategies For Unorderable Types and Values | 5 |
| 9 雜項說明 | 5 |
| 10 部份排序 | 6 |
| 索引 | 7 |

作者

Andrew Dalke 和 Raymond Hettinger

Python 的串列有一個內建的 `list.sort()` 方法可以原地 (in-place) 排序該串列，也有一個內建的 `sorted()` 函式可以排序可迭代物件 (iterable) 並建立一個新的排序好的串列。

在這份文件內，我們探索使用 Python 排序資料的各種方法。

1 基礎排序

單純的升序排序很容易做到：只要呼叫 `sorted()` 函式，它會回傳一個新的串列：

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

你也可以使用 `list.sort()` 方法，它會原地排序串列（`return` 傳 `None` 以避免混淆）。它通常會比 `sorted()` 來得不方便——但如果你不需要保留原始串列的話，它會稍微有效率一點。

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

另一個差別是 `list.sort()` 方法只有定義在串列上，而 `sorted()` 函式可以接受任何可迭代物件。

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

2 鍵函式 (key functions)

The `list.sort()` method and the functions `sorted()`, `min()`, `max()`, `heapq.nsmallest()`, and `heapq.nlargest()` have a *key* parameter to specify a function (or other callable) to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison using `str.casefold()`:

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

參數 *key* 的值必須是一個函式（或其它可呼叫物件），且這個函式接受單一引數 `return` 傳一個用來排序的鍵。因為對每個輸入來，鍵函式只會被呼叫一次，所以這個做法是快速的。

一個常見的模式是在排序複雜物件的時候使用一部分物件的索引值當作鍵，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)]
```

相同的做法也適用在有命名屬性的物件，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)]
```

具有命名屬性的物件可以如上方的方式使用一個常規的類 `return` 建立，或是他們可以是 `dataclass` 的實例或是一個 `named tuple`。

3 Operator 模組函式以及部份函式 (partial function) 評估

上述的鍵函式模式非常常見，所以 Python 提供了方便的函式讓物件存取更簡單且快速。operator 模組有 itemgetter()、attrgetter() 及 methodcaller() 函式可以使用。

使用這些函式讓上面的範例變得更簡單且快速：

```
>>> 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 模組的函式允許多層的排序，例如先用 *grade* 排序再用 *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)]
```

模組 functools 提供了另一個操作鍵函式的好用工具。partial() 函式可以減少多引數函式的引數數目，使其更適合作為鍵函式使用。

```
>>> 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 升序與降序

list.sort() 和 sorted() 都有一個 boolean 參數 *reverse* 用來表示是否要降序排序。例如將學生資料依據 *age* 做降序排序：

```
>>> 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 排序穩定性與合併排序

排序保證是穩定的，意思是當有多筆資料有相同的鍵，它們會維持原來的順序。

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

可以注意到有兩筆資料的鍵都是 *blue*，它們會維持本來的順序，即 ('blue', 1) 保證在 ('blue', 2) 前面。

這個美妙的特性讓你可以用一連串的排序來作出「合排」。例如對學生資料用 *grade* 做降序排序再用 *age* 做升序排序，你可以先用 *age* 排序一遍再用 *grade* 排序一遍：

```
>>> 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)]
```

這可以抽出一個包裝函式 (wrapper function)，接受一個串列及多個欄位及升降序的元組引數，來對這個串列排序多遍。

```
>>> 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 使用的 *Timsort* 演算法，因能利用資料集已經有的順序，可以有效率地做多次排序。

6 裝飾-排序-移除裝飾 (decorate-sort-undecorate)

這個用語的來源是因它做了以下三件事情：

- 首先，原始串列會裝飾 (decorated) 上新的值用來控制排序的順序。
- 接下來，排序裝飾過的串列。
- 最後，裝飾會被移除，以新的順序生成一個只包含原始值的串列。

例如用上面的方式來以 *grade* 排序學生資料：

```
>>> 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)]
```

這個方式會有效是因元組是依照字典順序 (lexicographically) 來比較，先比較第一個項目，如果一樣再比較第二個項目，依此類推。

在所有情況下都把索引 *i* 加入已裝飾的串列不是對需要的，但這樣做會有兩個好處：

- 排序會是穩定的 -- 如果兩個項目有相同的鍵，它們在排序好的串列中會保持原來的順序。
- 原始項目不需要是可以比較的，因最多只會用到前兩個項目就能固定裝飾過的元組的順序。例如原始串列可以包含不能直接用來排序的函數。

這個用語的另一個名字是 *Schwartzian transform*，是由於 Randal L. Schwartz 讓這個方法在 Perl 程式設計師間普及。

而因 Python 的排序提供了鍵函式，已經不太需要用到這個方法了。

7 比較函式 (comparison functions)

不像鍵函式回傳一個用來排序的值，比較函式計算兩個輸入間的相對順序。

例如 *天秤* 比較兩邊樣本給出相對的順序：較輕、相同或較重。同樣地，像是 `cmp(a, b)` 這樣的比較函式會回傳負數代表小於、0 代表輸入相同或正數代表大於。

當從其它語言翻譯演算法的時候常看到比較函式。有些函式庫也會提供比較函式作其 API 的一部份，例如 `locale.strcoll()` 就是一個比較函式。

為了滿足這些情境，Python 提供 `functools.cmp_to_key` 來包裝比較函式，讓其可以當作鍵函式來使用：

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

8 Strategies For Unorderable Types and Values

A number of type and value issues can arise when sorting. Here are some strategies that can help:

- Convert non-comparable input types to strings prior to sorting:

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

This is needed because most cross-type comparisons raise a `TypeError`.

- Remove special values prior to sorting:

```
>>> 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]
```

This is needed because the [IEEE-754 standard](#) specifies that, "Every NaN shall compare unordered with everything, including itself."

Likewise, `None` can be stripped from datasets as well:

```
>>> 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]
```

This is needed because `None` is not comparable to other types.

- Convert mapping types into sorted item lists before sorting:

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

This is needed because dict-to-dict comparisons raise a `TypeError`.

- Convert set types into sorted lists before sorting:

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

This is needed because the elements contained in set types do not have a deterministic order. For example, `list({'a', 'b'})` may produce either `['a', 'b']` or `['b', 'a']`.

9 雜項說明

- 要處理能理解本地語系 (locale aware) 的排序可以使用 `locale.strxfrm()` 當作鍵函式, 或 `locale.strcoll()` 當作比較函式。這樣做是必要的, 因^[1]在不同文化中就算是相同的字母, 按「字母順序」排序的結果也各不相同。
- `reverse` 參數依然會維持排序穩定性 (即有相同鍵的資料會保持原來順序)。有趣的是, 不加這個參數也可以模擬這個效果, 只要使用^[2]建的 `reversed()` 函式兩次:

```
>>> 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)]
```

- 排序時會使用 `<` 來比較兩個物件，因此要在類 `__lt__()` 加入排序順序比較規則，只要透過定義 `__lt__()` 方法：

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

然而，請注意，當 `__lt__()` 沒有被實做時，`<` 可以回退 (fall back) 使用 `__gt__()`（有關技術上的細節資訊請看 `object.__lt__()`）。為了避免意外發生，**PEP 8** 建議所有六種的比較函式都需要被實作。裝飾器 `total_ordering()` 被提供用來讓任務更簡單。

- 鍵函式不需要直接依賴用來排序的物件。鍵函式也可以存取外部資源，例如如果學生成績儲存在字典，它可以用來排序一個單獨的學生姓名串列：

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

10 部份排序

有些應用程式只需要排序部份的資料。基礎函式庫提供相較做完整排序更輕鬆的多項工具：

- `min()` 以及 `max()` 會分別回傳最小值及最大值。這些函式會將輸入資料進行一次傳遞且幾乎無須輔助記憶體。
- `heapq.nsmallest()` 以及 `heapq.nlargest()` 會分別回傳 n 個最小值及最大值。這些函式會將資料進行一次傳遞且一次只會保留 n 個元素在記憶體中。對於相對於輸入數量較小的 n 個值，這些函數進行的比較比完整排序要少得多。
- `heapq.heappush()` 和 `heapq.heappop()` 會建立並維持一個部份排序的資料排列，將最小的元素保持在位置 0。這些函式適合用來實作常用於任務排程的優先佇列。

索引

P

Python Enhancement Proposals
PEP 8, 6