
Sorting Techniques

Version 3.13.3

Guido van Rossum and the Python development team

avril 27, 2025

Python Software Foundation
Email : docs@python.org

Table des matières

1	Les bases du tri	1
2	Fonctions clef	2
3	Operator Module Functions and Partial Function Evaluation	3
4	Ascendant et descendant	3
5	Stabilité des tris et tris complexes	4
6	Decorate-Sort-Undecorate	4
7	Comparison Functions	5
8	Et n'oublions pas	5
9	Partial Sorts	6
	Index	7

Auteur

Andrew Dalke et Raymond Hettinger

Les listes Python ont une méthode native `list.sort()` qui modifie les listes elles-mêmes. Il y a également une fonction native `sorted()` qui construit une nouvelle liste triée depuis un itérable.

Dans ce document, nous explorons différentes techniques pour trier les données en Python.

1 Les bases du tri

Un tri ascendant simple est très facile : il suffit d'appeler la fonction `sorted()`. Elle renvoie une nouvelle liste triée :

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

Vous pouvez aussi utiliser la méthode `list.sort()`. Elle modifie la liste elle-même (et renvoie `None` pour éviter les confusions). Habituellement, cette méthode est moins pratique que la fonction `sorted()` -- mais si vous n'avez pas besoin de la liste originale, cette technique est légèrement plus efficace.

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

Une autre différence est que la méthode `list.sort()` est seulement définie pour les listes. Au contraire, la fonction `sorted()` accepte n'importe quel itérable.

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

2 Fonctions clef

`list.sort()` et `sorted()` ont un paramètre `key` afin de spécifier une fonction (ou autre callable) qui peut être appelée sur chaque élément de la liste avant d'effectuer des comparaisons.

Par exemple, voici une comparaison de texte insensible à la casse :

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

La valeur du paramètre `key` doit être une fonction (ou autre callable) qui prend un seul argument et renvoie une clef à utiliser à des fins de tri. Cette technique est rapide car la fonction clef est appelée exactement une seule fois pour chaque enregistrement en entrée.

Un usage fréquent est de faire un tri sur des objets complexes en utilisant les indices des objets en tant que clef. Par exemple :

```
>>> 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)]
```

La même technique marche pour des objets avec des attributs nommés. Par exemple :

```
>>> 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)]
```

Objects with named attributes can be made by a regular class as shown above, or they can be instances of `dataclass` or a named tuple.

3 Operator Module Functions and Partial Function Evaluation

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 `itemgetter()`, `attrgetter()`, and a `methodcaller()` function.

En utilisant ces fonctions, les exemples au dessus deviennent plus simples et plus rapides :

```
>>> 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)]
```

Les fonctions du module *operator* permettent plusieurs niveaux de tri. Par exemple, pour trier par *grade* puis par *age* :

```
>>> 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 `functools` module provides another helpful tool for making key-functions. The `partial()` function can reduce the *arity* of a multi-argument function making it suitable for use as a key-function.

```
>>> 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 Ascendant et descendant

`list.sort()` et `sorted()` acceptent un paramètre nommé *reverse* avec une valeur booléenne. C'est utilisé pour déterminer l'ordre descendant des tris. Par exemple, pour avoir les données des étudiants dans l'ordre inverse par *age* :

```
>>> 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 Stabilité des tris et tris complexes

Les tris sont garantis *stables*. Cela signifie que lorsque plusieurs enregistrements ont la même clef, leur ordre original est préservé.

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

Notez comme les deux enregistrements pour *blue* gardent leur ordre original et que par conséquent il est garanti que ('blue', 1) précède ('blue', 2).

Cette propriété fantastique vous permet de construire des tris complexes dans des tris en plusieurs étapes. Par exemple, afin de sortir les données des étudiants en ordre descendant par *grade* puis en ordre ascendant par *age*, effectuez un tri par *age* en premier puis un second tri par *grade* :

```
>>> 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)]
```

Ceci peut être encapsulé dans une fonction qui prend une liste et des n-uplets (attribut, ordre) pour les trier en plusieurs passes.

```
>>> 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)]
```

L'algorithme *Timsort* utilisé dans Python effectue de multiples tris efficacement parce qu'il peut tirer avantage de l'ordre existant dans un jeu de données.

6 Decorate-Sort-Undecorate

Cette technique est appelée Decorate-Sort-Undecorate et se base sur trois étapes :

- Premièrement, la liste de départ est décorée avec les nouvelles valeurs qui contrôlent l'ordre du tri.
- En second lieu, la liste décorée est triée.
- Enfin, la décoration est supprimée, créant ainsi une liste qui contient seulement la valeur initiale dans le nouvel ordre.

Par exemple, pour trier les données étudiant par *grade* en utilisant l'approche DSU :

```
>>> 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)]
```

Cette technique marche parce que les *n*-uplets sont comparés par ordre lexicographique ; les premiers objets sont comparés ; si il y a des objets identiques, alors l'objet suivant est comparé, et ainsi de suite.

Il n'est pas strictement nécessaire dans tous les cas d'inclure l'indice *i* dans la liste décorée, mais l'inclure donne deux avantages :

- Le tri est stable -- si deux objets ont la même clef, leur ordre sera préservé dans la liste triée.
- Les objets d'origine ne sont pas nécessairement comparables car l'ordre des *n*-uplets décorés sera déterminé par au plus les deux premiers objets. Donc par exemple la liste originale pourrait contenir des nombres complexes qui pourraient ne pas être triés directement.

Un autre nom pour cette technique est [Schwartzian transform](#), après que Randal L. Schwartz l'ait popularisé chez les développeurs Perl.

Maintenant que le tri Python fournit des fonctions-clef, cette technique n'est plus souvent utilisée.

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)) # locale-aware sort order
```

8 Et n'oublions pas

- 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.
- Le paramètre *reverse* maintient toujours un tri stable (de telle sorte que les enregistrements avec des clef égales gardent le même ordre). Notez que cet effet peut être simulé sans le paramètre en utilisant la fonction native `reversed()` deux fois :

```
>>> 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)]
```

- The sort routines use `<` when making comparisons between two objects. So, it is easy to add a standard sort order to a class by defining an `__lt__()` method :

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

However, note that `<` can fall back to using `__gt__()` if `__lt__()` is not implemented (see `object.__lt__()` for details on the mechanics). To avoid surprises, [PEP 8](#) recommends that all six comparison methods be implemented. The `total_ordering()` decorator is provided to make that task easier.

- Les fonctions clef n'ont pas besoin de dépendre directement des objets triés. Une fonction clef peut aussi accéder à des ressources externes. En l'occurrence, si les grades des étudiants sont stockés dans un dictionnaire, ils peuvent être utilisés pour trier une liste différentes de noms d'étudiants :

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

9 Partial Sorts

Some applications require only some of the data to be ordered. The standard library provides several tools that do less work than a full sort :

- `min()` and `max()` return the smallest and largest values, respectively. These functions make a single pass over the input data and require almost no auxiliary memory.
- `heapq.nsmallest()` and `heapq.nlargest()` return the n smallest and largest values, respectively. These functions make a single pass over the data keeping only n elements in memory at a time. For values of n that are small relative to the number of inputs, these functions make far fewer comparisons than a full sort.
- `heapq.heappush()` and `heapq.heappop()` create and maintain a partially sorted arrangement of data that keeps the smallest element at position 0. These functions are suitable for implementing priority queues which are commonly used for task scheduling.

Index

P

Python Enhancement Proposals
PEP 8, [5](#)