Guide pour le tri

Version 2.7.16

Guido van Rossum
and the Python development team

octobre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

Author Andrew Dalke et Raymond Hettinger

Release 0.1

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]
```

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]
```

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

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.

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

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

La valeur du paramètre *key* devrait être une fonction 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),
```

(suite sur la page suivante)

(suite de la page précédente)

```
...     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 Fonctions du module *operator*

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.

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 `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 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

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)]
```

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)]
```

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

6 La méthode traditionnelle utilisant 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 tuples 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 on la même clef, leur ordre sera préservé dans la liste triée.
- Les objets d'origine ne sont pas nécessairement comparables car l'ordre des tuples 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.

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 La méthode traditionnelle d'utiliser le paramètre *cmp*

Plusieurs construction données dans ce guide se basent sur Python 2.4 ou plus. Avant cela, il n'y avait pas la fonction `sorted()` et la méthode `list.sort()` ne prenait pas d'arguments nommés. À la place, toutes les versions Python 2.x utilisaient un paramètre *cmp* pour prendre en charge les fonctions de comparaisons définies par les utilisateurs.

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) # doctest: +SKIP
[1, 2, 3, 4, 5]
```

Ou nous pouvons inverser l'ordre de comparaison avec :

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

En portant du code depuis Python 2.X vers 3.x, des problèmes peuvent survenir quand des utilisateurs fournissent une fonction de comparaison et qu'il faut convertir cette fonction en une fonction-clef. La fonction d'encapsulation suivante rend cela plus facile à faire :

```
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
```

Pour convertir une fonction clef, ils suffit d'encapsuler l'ancienne fonction de comparaison :

```
>>> 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 Curiosités et conclusion

— Pour du tri de texte localisé, utilisez `locale.strxfrm()` en tant que fonction clef ou `locale.strcoll()` comme fonction de comparaison.

— 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.

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