
HOW TO - Ordenar

Versión 3.10.18

Guido van Rossum
and the Python development team

julio 08, 2025

Python Software Foundation
Email: docs@python.org

Índice general

1	Conceptos básicos de ordenación	1
2	Funciones clave	2
3	Funciones del módulo <i>operator</i>	3
4	Ascendente y descendente	3
5	Estabilidad de ordenamiento y ordenamientos complejos	3
6	El método tradicional utilizando <i>Decorate-Sort-Undecorate</i>	4
7	El método tradicional utilizando el Parámetro <i>cmp</i>	5
8	Comentarios finales	6

Autor Andrew Dalke and Raymond Hettinger

Versión 0.1

Las listas de Python tienen un método incorporado `list.sort()` que modifica la lista in situ. También hay una función incorporada `sorted()` que crea una nueva lista ordenada a partir de un iterable.

En este documento exploramos las distintas técnicas para ordenar datos usando Python.

1 Conceptos básicos de ordenación

Una clasificación ascendente simple es muy fácil: simplemente llame a la función `sorted()`. Retorna una nueva lista ordenada:

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

También puede usar el método `list.sort()`. Modifica la lista in situ (y retorna `None` para evitar confusiones). Por lo general, es menos conveniente que `sorted()`, pero si no necesita la lista original, es un poco más eficiente.

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

Otra diferencia es que el método `list.sort()` solo aplica para las listas. En contraste, la función `sorted()` acepta cualquier iterable.

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

2 Funciones clave

Ambos `list.sort()` y `sorted()` tienen un parámetro `key` para especificar una función (u otra invocable) que se llamará en cada elemento de la lista antes de hacer comparaciones.

Por ejemplo, aquí hay una comparación de cadenas que no distingue entre mayúsculas y minúsculas:

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

El valor del parámetro `key` debe ser una función (u otra invocable) que tome un solo argumento y retorne una clave para usar con fines de clasificación. Esta técnica es rápida porque la función de la tecla se llama exactamente una vez para cada registro de entrada.

Un uso frecuente es ordenar objetos complejos utilizando algunos de los índices del objeto como claves. Por ejemplo:

```
>>> 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 misma técnica funciona para objetos con atributos nombrados. Por ejemplo:

```
>>> 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)]
```

3 Funciones del módulo *operator*

Las funciones clave utilizadas anteriormente son muy comunes, por lo que Python proporciona funciones para facilitar y agilizar el uso de las funciones de acceso. El módulo `operator` contiene las funciones `itemgetter()`, `attrgetter()`, y `methodcaller()`.

Usando esas funciones, los ejemplos anteriores se vuelven más simples y rápidos:

```
>>> 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)]
```

Las funciones del módulo *operator* permiten múltiples niveles de clasificación. Por ejemplo, para ordenar por *grade* y luego por *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)]
```

4 Ascendente y descendente

Ambos `list.sort()` y `sorted()` aceptan un parámetro *reverse* con un valor booleano. Esto se usa para marcar tipos descendentes. Por ejemplo, para obtener los datos de los estudiantes en orden inverso de *edad*:

```
>>> 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 Estabilidad de ordenamiento y ordenamientos complejos

Se garantiza que las clasificaciones serán *estables*. Eso significa que cuando varios registros tienen la misma clave, se conserva su orden original.

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

Observe cómo los dos registros para *blue* conservan su orden original de modo que se garantice que `('blue', 1)` preceda a `('blue', 2)`.

Esta maravillosa propiedad le permite construir ordenamientos complejos en varias etapas. Por ejemplo, para ordenar los datos de estudiantes en orden descendente por *grade* y luego ascendente por *age*, ordene primero por *age* y luego por *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)]
```

Esto se puede encapsular en una función que tome una lista y tuplas (atributo, orden) para ordenarlas por múltiples pases.

```
>>> 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)]
```

El algoritmo **Timsort** utilizado en Python realiza múltiples ordenamientos de manera eficiente porque puede aprovechar cualquier orden ya presente en el conjunto de datos.

6 El método tradicional utilizando *Decorate-Sort-Undecorate*

Este patrón de implementación, llamado DSU (por sus siglas en inglés *Decorate-Sort-Undecorate*), se realiza en tres pasos:

- Primero, la lista inicial está «decorada» con nuevos valores que controlarán el orden en que se realizará el pedido.
- En segundo lugar, se ordena la lista decorada.
- Finalmente, los valores decorados se eliminan, creando una lista que contiene solo los valores iniciales en el nuevo orden.

Por ejemplo, para ordenar los datos de los estudiantes por *grade* utilizando el enfoque 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)]
```

Esta técnica funciona porque las tuplas se comparan en orden lexicográfico; se comparan los primeros objetos; si hay objetos idénticos, se compara el siguiente objeto, y así sucesivamente.

No es estrictamente necesario en todos los casos incluir el índice *i* en la lista decorada, pero incluirlo ofrece dos ventajas:

- El orden es estable: si dos elementos tienen la misma clave, su orden se conservará en la lista ordenada.
- Los elementos originales no tienen que ser comparables porque el orden de las tuplas decoradas estará determinado por, como máximo, los dos primeros elementos. Entonces, por ejemplo, la lista original podría contener números complejos que no se pueden ordenar directamente.

Otro nombre para esta técnica es **Transformación Schwartziana**, después de que Randal L. Schwartz la popularizara entre los programadores de Perl.

Ahora que la clasificación de Python proporciona funciones clave, esta técnica ya no se usa con frecuencia.

7 El método tradicional utilizando el Parámetro *cmp*

Muchos constructores presentados en este CÓMO asumen el uso de Python 2.4 o superior. Antes de eso, no había una función `sorted()` incorporada y el método `list.sort()` no tomaba los argumentos nombrados. A pesar de esto, todas las versiones de Py2.x admiten el parámetro *cmp* para manejar la función de comparación especificada por el usuario.

En Py3.0, el parámetro *cmp* se eliminó por completo (como parte de un mayor esfuerzo para simplificar y unificar el lenguaje, eliminando el conflicto entre las comparaciones enriquecidas y el método mágico `__cmp__()`).

En Py2.x, se permitió una función opcional a la que se puede llamar para hacer las comparaciones. Esa función debe tomar dos argumentos para comparar y luego retornar un valor negativo para menor que, retornar cero si son iguales o retornar un valor positivo para mayor que. Por ejemplo, podemos hacer:

```
>>> def numeric_compare(x, y):
...     return x - y
>>> sorted([5, 2, 4, 1, 3], cmp=numeric_compare)
[1, 2, 3, 4, 5]
```

O puede revertir el orden de comparación con:

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

Al migrar código de Python 2.x a 3.x, puede surgir la situación de que el usuario proporciona una función de comparación y necesita convertirla en una función clave. La siguiente envoltura lo hace fácil de hacer:

```
def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'
    class K:
        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
```

Para convertir a una función clave, simplemente ajuste la antigua función de comparación:

```
>>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
[5, 4, 3, 2, 1]
```

En Python 3.2, la función `functools.cmp_to_key()` se agregó al módulo `functools` en la biblioteca estándar.

8 Comentarios finales

- Para una ordenación local, use `locale.strxfrm()` para una función clave o `locale.strcoll()` para una función de comparación.
- El parámetro *reverse* aún mantiene estabilidad de ordenamiento (de modo que los registros con claves iguales conservan el orden original). Curiosamente, ese efecto se puede simular sin el parámetro utilizando la función incorporada `reversed()` dos veces:

```
>>> 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__()`).

- Las funciones clave no necesitan depender directamente de los objetos que se ordenan. Una función clave también puede acceder a recursos externos. Por ejemplo, si las calificaciones de los estudiantes se almacenan en un diccionario, se pueden usar para ordenar una lista separada de nombres de estudiantes:

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