

---

# Sorting Techniques

*Versión 3.14.0rc3*

**Guido van Rossum and the Python development team**

octubre 01, 2025

Python Software Foundation  
Email: docs@python.org

## Índice general

<b>1</b>	<b>Conceptos básicos de ordenación</b>	<b>1</b>
<b>2</b>	<b>Funciones clave</b>	<b>2</b>
<b>3</b>	<b>Operator Module Functions and Partial Function Evaluation</b>	<b>3</b>
<b>4</b>	<b>Ascendente y descendente</b>	<b>3</b>
<b>5</b>	<b>Estabilidad de ordenamiento y ordenamientos complejos</b>	<b>3</b>
<b>6</b>	<b>Decorate-Sort-Undecorate</b>	<b>4</b>
<b>7</b>	<b>Funciones de comparación</b>	<b>5</b>
<b>8</b>	<b>Strategies For Unorderable Types and Values</b>	<b>5</b>
<b>9</b>	<b>Curiosidades</b>	<b>6</b>
<b>10</b>	<b>Partial Sorts</b>	<b>6</b>
	<b>Índice</b>	<b>8</b>

---

### Autor

Andrew Dalke and Raymond Hettinger

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

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']
```

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)]
```

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.

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)]
```

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 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 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 Funciones de comparación

A diferencia de las funciones clave que devuelven un valor absoluto para la ordenación, una función de comparación calcula la ordenación relativa para dos entradas.

Por ejemplo, una [escala de balance](#) compara dos muestras dando un orden relativo: más ligero, igual o más pesado. Del mismo modo, una función de comparación como `cmp(a, b)` devolverá un valor negativo para menor que, cero si las entradas son iguales, o un valor positivo para mayor que.

Es habitual encontrar funciones de comparación al traducir algoritmos de otros lenguajes. Además, algunas bibliotecas proporcionan funciones de comparación como parte de su API. Por ejemplo, `locale.strcoll()` es una función de comparación.

Para adaptarse a estas situaciones, Python proporciona `functools.cmp_to_key` para envolver la función de comparación y hacerla utilizable como una función clave:

```
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 Curiosidades

- Para ordenar teniendo en cuenta la localización, utilice `locale.strxfrm()` para una función clave o `locale.strcoll()` para una función de comparación. Esto es necesario porque la ordenación «alfabética» puede variar entre culturas aunque el alfabeto subyacente sea el mismo.
- 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__()` 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.

- 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']
```

## 10 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.

## Índice

### P

Python Enhancement Proposals

PEP 8, 6