Sorting HOW TO

Release 2.7.15

Guido van Rossum
and the Python development team

May 21, 2018

Python Software Foundation
Email: docs@python.org

Contents

1 Sorting Basics 1

2 Key Functions 2

3 Operator Module Functions 3

4 Ascending and Descending 3

5 Sort Stability and Complex Sorts 3

6 The Old Way Using Decorate-Sort-Undecorate 4

7 The Old Way Using the *cmp* Parameter 5

8 Odd and Ends 6

Author Andrew Dalke and Raymond Hettinger

Release 0.1

Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.

1 Sorting Basics

A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```
>>> 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]
```

Another difference is that the `list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

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

2 Key Functions

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.

For example, here's a case-insensitive string comparison:

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

The value of the *key* parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

A common pattern is to sort complex objects using some of the object's indices as keys. For example:

```
>>> 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)]
```

The same technique works for objects with named attributes. For example:

```
>>> 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 Operator Module Functions

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.

Using those functions, the above examples become simpler and faster:

```
>>> 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)]
```

The operator module functions allow multiple levels of sorting. For example, to sort by *grade* then by *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 Ascending and Descending

Both `list.sort()` and `sorted()` accept a *reverse* parameter with a boolean value. This is used to flag descending sorts. For example, to get the student data in reverse *age* order:

```
>>> 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 Sort Stability and Complex Sorts

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)]
```

Notice how the two records for *blue* retain their original order so that `('blue', 1)` is guaranteed to precede `('blue', 2)`.

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending *grade* and then ascending *age*, do the *age* sort first and then sort again using *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)]
```

The Timsort algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset.

6 The Old Way Using Decorate-Sort-Undecorate

This idiom is called Decorate-Sort-Undecorate after its three steps:

- First, the initial list is decorated with new values that control the sort order.
- Second, the decorated list is sorted.
- Finally, the decorations are removed, creating a list that contains only the initial values in the new order.

For example, to sort the student data by *grade* using the DSU approach:

```
>>> 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)]
```

This idiom works because tuples are compared lexicographically; the first items are compared; if they are the same then the second items are compared, and so on.

It is not strictly necessary in all cases to include the index i in the decorated list, but including it gives two benefits:

- The sort is stable – if two items have the same key, their order will be preserved in the sorted list.
- The original items do not have to be comparable because the ordering of the decorated tuples will be determined by at most the first two items. So for example the original list could contain complex numbers which cannot be sorted directly.

Another name for this idiom is Schwartzian transform, after Randal L. Schwartz, who popularized it among Perl programmers.

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 The Old Way Using the *cmp* Parameter

Many constructs given in this HOWTO assume Python 2.4 or later. Before that, there was no `sorted()` builtin and `list.sort()` took no keyword arguments. Instead, all of the Py2.x versions supported a *cmp* parameter to handle user specified comparison functions.

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)
[1, 2, 3, 4, 5]
```

Or you can reverse the order of comparison with:

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

When porting code from Python 2.x to 3.x, the situation can arise when you have the user supplying a comparison function and you need to convert that to a key function. The following wrapper makes that easy to do:

```
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
```

To convert to a key function, just wrap the old comparison function:

```
>>> 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 Odd and Ends

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function.
- 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.

- Key functions need not depend directly on the objects being sorted. A key function can also access external resources. For instance, if the student grades are stored in a dictionary, they can be used to sort a separate list of student names:

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