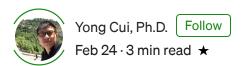
You have 1 free member-only story left this month. See the benefits of Medium membership

Advanced Sorting in Python Using Lambdas and Custom Functions

Sort lists of elements of non-basic data types (e.g., int, str)



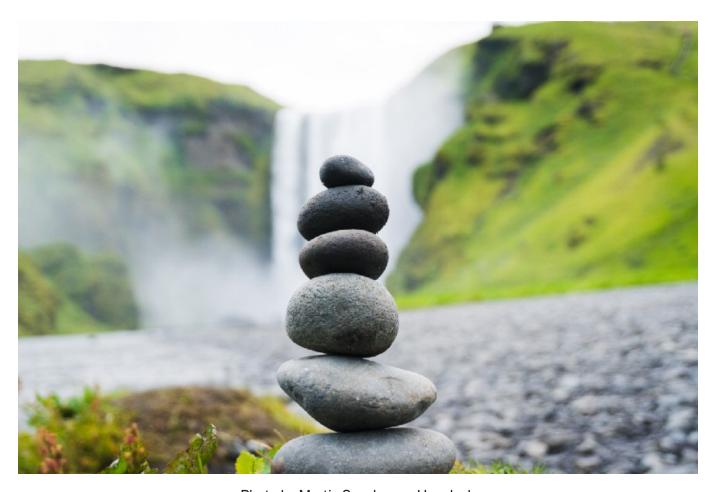


Photo by Martin Sanchez on Unsplash

When it comes to sorting, the most-used Python functions are sorted() and sort().

Although there are some differences in their usage, memory usage, and operation speed, as discussed in a previous <u>Medium article</u>, both functions can be used to sort a <code>list</code>. Below is a simple example.

Basic Sorting Examples

In the above example, there are a few things to highlight for those who are less familiar with the <code>sorted()</code> and <code>sort()</code> functions.

- The sort() function is actually an instance method of the list data type, such that the usage is list.sort().
- The sorting operation by <code>sort()</code> is in place and returns <code>None</code>. Thus, if you check the type by running <code>type(numbers.sort())</code>, the output will be <code>NoneType</code>.
- Unlike the <code>sort()</code> function, the <code>sorted()</code> function has better flexibility by allowing the argument to be a <code>list</code> or any other <code>iterable</code> objects, like the <code>tuple</code> shown in the example.
- For the <code>sorted()</code> function, the return value is a <code>list</code> even if we pass in a non-list data like a <code>tuple</code>.
- The default sorting order is ascending. If we specify that reverse=True, the sorting order will be descending.

Essentially, both <code>sorted()</code> and <code>sort()</code> functions are comparable when a list is to be sorted with the <code>sorted()</code> function has better flexibility. Therefore, for the following demonstration, we'll just use the <code>sorted()</code> function.

Sorting With Key

Beyond the basic sorting operations on int and str, what can we do if we want to sort a list of dictionaries, like the one called activities below?

```
2
         return {'order': order, 'day': day, 'activity': activity}
 3
     mon_activity = daily_activity(0, 'Mon', 'Baseball')
 4
 5
     tue_activity = daily_activity(1, 'Tue', 'Swim')
     wed_activity = daily_activity(2, 'Wed', 'Soccer')
 6
    thu_activity0 = daily_activity(3, 'Thu', 'Basketball')
 7
     thu_activity1 = daily_activity(4, 'Thu', 'Dance')
     thu_activity2 = daily_activity(5, 'Thu', 'Football')
     activities = [mon_activity, tue_activity, wed_activity, thu_activity0, thu_activity1, thu_activi
10
python_sorting_custom_data.py hosted with ♥ by GitHub
                                                                                              view raw
```

List of Dictionaries

If we sort the variable activities, the following error will occur, as the interpreter doesn't know how to compare instances of dict. What should we do then?

```
TypeError: '<' not supported between instances of 'dict' and 'dict'
```

Lambdas come to the rescue! If you recall, a lambda in Python is just an anonymous function that has one or more arguments, but only one expression.

In the example below, we create a lambda and assign it to the variable add_ten . As a lambda is a function, the assigned variable add_ten is a function, and thus, we call it as a regular function using the parentheses.

```
1  Syntax:
2  lambda arguments : expression
3  Example:
4  add_ten = lambda a : a + 10
5  print(type(add_ten))
6  # Prints <class 'function'>
7  print(add_ten(5))
8  # Prints 15
lambda.py hosted with \( \subseteq \text{by GitHub} \)
view raw
```

We have some basic ideas of how a lambda works, and we can now see how we can use it to sort the dictionaries.

```
sorted_activities = sorted(activities, key=lambda x: x['order'], reverse=True)
print(sorted_activities)
```

```
# Prints the followling lines

[{'order': 5, 'day': 'Thu', 'activity': 'Football'},

{'order': 4, 'day': 'Thu', 'activity': 'Dance'},

{'order': 3, 'day': 'Thu', 'activity': 'Basketball'},

{'order': 2, 'day': 'Wed', 'activity': 'Soccer'},

{'order': 1, 'day': 'Tue', 'activity': 'Swim'},

{'order': 0, 'day': 'Mon', 'activity': 'Baseball'}]

python_sorting_dict_lambda.py hosted with $\infty$ by GitHub
view raw
```

Sort a list of dictionaries

As shown in the above example, we create a lambda and pass it in as the key argument. Now, the interpreter doesn't complain that the list of dict s can't be sorted. Hooray!

Actually, we can write a lambda that is a little more complicated if we want the activities to be sorted by more than one keys.

```
sorted_activities = sorted(activities, key=lambda x: (x['day'], x['activity']), reverse=True)
     print(sorted_activities)
 3
     # Prints the following lines
4
 5
     [{'order': 2, 'day': 'Wed', 'activity': 'Soccer'},
     {'order': 1, 'day': 'Tue', 'activity': 'Swim'},
7
     {'order': 5, 'day': 'Thu', 'activity': 'Football'},
8
     {'order': 4, 'day': 'Thu', 'activity': 'Dance'},
9
      {'order': 3, 'day': 'Thu', 'activity': 'Basketball'},
10
      {'order': 0, 'day': 'Mon', 'activity': 'Baseball'}]
python sorting dict lambda1.py hosted with \bigcirc by GitHub
                                                                                               view raw
```

Sort by day and activity

As shown above, the list is now sorted by day and activity. The same effect can be achieved by using the itemgetter() function as shown below.

Actually, the operator module has additional operations, such as attrgetter(), which is handy in sorting custom objects.

```
from operator import itemgetter

sorted_activities = sorted(activities, key=itemgetter('day',
'activity'), reverse=True)
```

From the above example, you may have noticed that for the key argument, we can simply pass in a function, which gives us more flexibility in how we can customize the sorting. Here's an example of this.

```
def activity_sorting(activity):
 1
         return activity['activity']
2
 3
     sorted_activities = sorted(activities, key=activity_sorting)
4
     print(sorted_activities)
5
6
7
    # Prints the following lines
     [{'order': 0, 'day': 'Mon', 'activity': 'Baseball'},
     {'order': 3, 'day': 'Thu', 'activity': 'Basketball'},
9
     {'order': 4, 'day': 'Thu', 'activity': 'Dance'},
10
     {'order': 5, 'day': 'Thu', 'activity': 'Football'},
11
      {'order': 2, 'day': 'Wed', 'activity': 'Soccer'},
12
13
      {'order': 1, 'day': 'Tue', 'activity': 'Swim'}]
python_sorting_dict_fun.py hosted with ♥ by GitHub
                                                                                              view raw
```

Sort by a custom function

We have a function called <code>activity_sorting</code>, which is used as the <code>key</code> for the <code>sorted()</code> function. From the output, you can tell that the <code>activities</code> variable is now sorted by the <code>activity</code>.

Conclusion

This tutorial has introduced how we can use lambdas and custom functions to sort a list of dictionaries in Python. These approaches can also be applied to lists of custom objects.

If the sorting key involves just one expression, you may want to consider using lambdas without writing a full function outside the sorted() or sort() function.

Thanks to Zack Shapiro.

Programming Python Data Science Python3 Functional Programming

Get the Medium app



