
Sorting Techniques

Yayın 3.13.0rc2

Guido van Rossum and the Python development team

Eylül 25, 2024

Python Software Foundation
Email: docs@python.org

İçindekiler

1	Sıralama Temelleri	2
2	Anahtar Fonksiyonları	2
3	Operator Module Functions and Partial Function Evaluation	3
4	Yükselen ve Alçalan	4
5	Sıralama Kararlılığı ve Karmaşık Sıralamalar	4
6	Süsle-Sırala-Boz	4
7	Karşılaştırma Fonksiyonları	5
8	Tuhafliklar ve Sonlar	5
9	Partial Sorts	6
	Dizin	7

Yazar

Andrew Dalke and Raymond Hettinger

Python listeleri, listeyi yerinde değiştiren yerleşik bir `list.sort()` yöntemine sahiptir. Ayrıca, bir yinelenebilir-den yeni bir sıralanmış liste oluşturan bir `sorted()` yerleşik işlevi de vardır.

Bu belgede, Python kullanarak verileri sıralamak için çeşitli teknikleri keşfediyor olacağız.

1 Sıralama Temelleri

Basit bir artan sıralama yaratmak çok kolaydır: `sorted()` fonksiyonunu çağırmanız yeterlidir. Bu fonksiyon, yeni bir sıralanmış liste döndürür:

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

Ayrıca `list.sort()` yöntemini de kullanabilirsiniz. Listeyi yerinde modifiye eder (ve karışıklığı önlemek için `None` döndürür). Genellikle `sorted()` yönteminden daha az kullanışlıdır - ancak orijinal listeye ihtiyacınız yoksa, biraz daha verimlidir.

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

Diğer bir fark ise `list.sort()` metodunun sadece listeler için tanımlanmış olmasıdır. Buna karşılık, `sorted()` fonksiyonu herhangi bir yinelenebilir kabul eder.

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

2 Anahtar Fonksiyonları

Hem `list.sort()` hem de `sorted()`, karşılaştırma yapmadan önce listenin her ögesi üzerinde çağrılacak bir işlevi (veya başka bir çağrılabilir) özellikle belirtmek için bir `key` parametresine sahiptir.

Örneğin, büyük/küçük harfe duyarlı olmayan bir dize karşılaştırması bu şekilde yapılmaktadır:

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

`key` (anahtar) parametresinin değeri, tek bir argüman alan ve sıralama amacıyla kullanılacak bir anahtarı döndürecek bir fonksiyon (veya başka bir çağrılabilir) olmalıdır. Bu teknik, hızlı çalışır çünkü anahtar işlevi her girdi (input) kaydı için tam olarak bir kez çağrılır.

Yaygın bir model, nesnenin bazı indislerini anahtar olarak kullanarak karmaşık nesneleri sıralamaktır. Örneğin:

```
>>> 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)]
```

Aynı teknik, adlandırılmış niteliklere sahip nesneler için de geçerlidir. Örneğin:

```
>>> 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),
```

(sonraki sayfaya devam)

```

...     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.

Bu fonksiyonların kullanımı sonucunda, yukarıdaki örnekler daha basit ve hızlı hale gelir:

```

>>> 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)]

```

Operatör modülü fonksiyonları birden fazla seviyede sıralama yapılmasına izin verir. Örneğin, *sınıf* ve ardından *yaş*'a göre sıralamak için:

```

>>> 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 Yükselen ve Alçalan

Hem `list.sort()` hem de `sorted()` boolean değerli bir *reverse* parametresi kabul eder. Bu, azalan sıralamaları işaretlemek için kullanılır. Örneğin, öğrenci verilerini ters olarak *yaş* sırasına göre elde etmek için:

```
>>> 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 Sıralama Kararlılığı ve Karmaşık Sıralamalar

Sıralamaların *kararlı* olması kesindir. Bunun anlamı ise, birden fazla kayıt aynı anahtara sahip olduğunda, orijinal sıralamanın korunacağıdır.

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

blue için iki kaydın orijinal sıralarını nasıl koruduğuna dikkat edin, böylece `('blue', 1)` kaydının `('blue', 2)` kaydından önce gelmesi garanti edilir.

Bu harika özellik, birkaç sıralama adımı sonucunda karmaşık sıralamalar oluşturmanıza olanak tanır. Örneğin, öğrenci verilerini azalan *sınıf* ve ardından artan *yaş* ile sıralamak için, önce *yaş* sıralamasını yapın ve ardından *sınıf* kullanarak tekrar sıralayın:

```
>>> 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)]
```

Bu, bir listeyi ve alan çiftlerini alıp bunları birden fazla geçişte sıralayabilen bir sarmalayıcı fonksiyon oluşturacak şekilde soyutlanabilir.

```
>>> 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)]
```

Python'da kullanılan *Timsort* algoritması, bir veri kümesinde zaten mevcut olan herhangi bir sıralamadan yararlanabildiği için çoklu sıralamayı verimli bir şekilde yapar.

6 Süsle-Sırala-Boz

Süsle-Sırala-Boz deyimini, içerdiği üç adımdan ilham alınarak oluşturulmuştur:

- İlk olarak, ilk liste sıralama düzenini kontrol eden yeni değerlerle süslenir (dekore edilir).
- İkinci olarak, dekore edilmiş liste sıralanır.
- Son olarak, süslemeler kaldırılır ve yeni sırada yalnızca ilk değerleri içeren bir liste oluşturulur.

Örneğin, DSU yaklaşımını kullanarak öğrenci verilerini *sınıf* bazında sıralamak için:

```
>>> 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)]
```

Bu deyim, veri çiftleri leksikografik (sözlükbilimsel) olarak karşılaştırıldığı için işe yarar: İlk öğeler karşılaştırılır, aynıysa ikinci öğeler karşılaştırılır ve bu böyle devam eder.

i indeksini dekore edilmiş listeye dahil etmek her durumda gerekli değildir, ancak dahil etmek iki fayda sağlar:

- Sıralama sabittir – iki öğe aynı anahtara sahipse, sıralanmış listede sıraları korunacaktır.
- Orijinal öğelerin karşılaştırılabilir olması gerekmez çünkü dekore edilmiş çiftlerin sıralaması en fazla ilk iki öğe tarafından belirlenecektir. Örneğin orijinal liste doğrudan sıralanamayan karmaşık sayılar içerebilir.

Bu deyimın bir diğer adı da, deyimı Perl programcıları arasında popüler hale getiren Randal L. Schwartz’a atfen *Schwartzian transform*’dur.

Artık Python sıralama anahtar fonksiyonları sağladığından, bu tekniğe pek sık ihtiyaç duyulmamaktadır.

7 Karşılaştırma Fonksiyonları

Sıralama için mutlak bir değer döndüren anahtar işlevlerinin aksine, karşılaştırma işlevi iki girdi için göreceli bir sıralamayı hesaplar.

Örneğin, bir *balance scale* iki örneği karşılaştırarak göreceli bir sıralama verir: daha hafif, eşit veya daha ağır. Benzer şekilde, `cmp(a, b)` karşılaştırma fonksiyonu; girdiler eşitse sıfır, küçükse negatif, büyükse pozitif bir değer döndürür.

Algoritmaları diğer dillerden çevirirken karşılaştırma fonksiyonlarıyla karşılaşmak yaygındır. Ayrıca, bazı kütüphaneler API’lerinin bir parçası olarak karşılaştırma fonksiyonları sağlar. Örneğin, `locale.strcoll()` bir karşılaştırma fonksiyonudur.

Bu durumlara uyum sağlamak için Python, karşılaştırma fonksiyonunu bir anahtar fonksiyon olarak kullanılabilir hale getirmek için `functools.cmp_to_key` aracını sağlar:

```
sorted(words, key=cmp_to_key(strcoll)) # locale-aware sort order
```

8 Tuhafliklar ve Sonlar

- Yerel ayarlara duyarlı sıralama yapmak istiyorsanız, anahtar işlevleri için `locale.strxfrm()` ve karşılaştırma işlevleri için `locale.strcoll()` kullanabilirsiniz. Bu gereklidir çünkü “alfabetik” sıralamalar, temel alfabe aynı olsa bile kültürler arasında farklılık gösterebilir.
- *reverse* parametresi sıralamalardaki kararlılığı aynı şekilde korur (böylece eşit anahtarlara sahip kayıtlar orijinal sırasını korumuş olur). İlginç bir şekilde bu etki, parametre olmadan yerleşik `reversed()` fonksiyonu iki kez kullanılarak da simüle edilebilir:

```
>>> 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.

- Anahtar işlevlerinin doğrudan sıralanan nesnelere bağlı olması gerekmez. Bir anahtar işlevi, harici kaynaklara da erişebilir. Örneğin, öğrenci notları bir sözlükte saklanıyorsa, öğrenci adlarından oluşan ayrı bir listenin sıralanmasında da kullanılabilirler:

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

9 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.

Dizin

P

Python Geliştirme Önerileri
PEP 8,6