
Sorting Techniques

릴리스 3.12.3

Guido van Rossum and the Python development team

6월 05, 2024

Contents

1	정렬 기초	2
2	키 함수	2
3	Operator Module Functions and Partial Function Evaluation	3
4	오름차순과 내림차순	4
5	정렬 안정성과 복잡한 정렬	4
6	Decorate-Sort-Undecorate	5
7	Comparison Functions	5
8	Odds and Ends	6
9	Partial Sorts	6
	색인	7

저자

Andrew Dalke와 Raymond Hettinger

파이썬 리스트에는 리스트를 제자리에서 (in-place) 수정하는 내장 `list.sort()` 메서드가 있습니다. 또한, 이터러블로부터 새로운 정렬된 리스트를 만드는 `sorted()` 내장 함수가 있습니다.

이 문서에서는, 파이썬을 사용하여 데이터를 정렬하는 다양한 기술을 살펴봅니다.

1 정렬 기초

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]
```

`list.sort()` 메서드를 사용할 수도 있습니다. 리스트를 제자리에서 수정합니다(그리고 혼동을 피하고자 `None`을 반환합니다). 일반적으로 `sorted()` 보다 덜 편리합니다 - 하지만 원래 목록이 필요하지 않다면, 이것이 약간 더 효율적입니다.

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

또 다른 점은 `list.sort()` 메서드가 리스트에게만 정의된다는 것입니다. 이와 달리, `sorted()` 함수는 모든 이터러블을 받아들입니다.

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

2 키 함수

`list.sort()`와 `sorted()`는 모두 비교하기 전에 각 리스트 요소에 대해 호출할 함수(또는 다른 콜러블)를 지정하는 `key` 매개 변수를 가지고 있습니다.

예를 들어, 다음은 대소 문자를 구분하지 않는 문자열 비교입니다:

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

`key` 매개 변수의 값은 단일 인자를 취하고 정렬 목적으로 사용할 키를 반환하는 함수(또는 다른 콜러블)여야 합니다. 키 함수가 각 입력 레코드에 대해 정확히 한 번 호출되기 때문에 이 기법은 빠릅니다.

일반적인 패턴은 객체의 인덱스 중 일부를 키로 사용하여 복잡한 객체를 정렬하는 것입니다. 예를 들어:

```
>>> 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)]
```

같은 기법이 이름있는 어트리뷰트를 갖는 객체에서도 작동합니다. 예를 들어:

```
>>> 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))
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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.

이러한 함수를 사용하면, 위의 예제가 더 간단 해지고 빨라집니다:

```
>>> 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)]
```

operator 모듈 함수는 다중 수준의 정렬을 허용합니다. 예를 들어, 먼저 *grade*로 정렬한 다음 *age*로 정렬하려면, 이렇게 합니다:

```
>>> 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 오름차순과 내림차순

`list.sort()`와 `sorted()`는 모두 불리언 값을 갖는 *reverse* 매개 변수를 받아들입니다. 내림차순 정렬을 지정하는 데 사용됩니다. 예를 들어, 학생 데이터를 역 *age* 순서로 얻으려면, 이렇게 합니다:

```
>>> 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 정렬 안정성과 복잡한 정렬

정렬은 안정적임이 보장됩니다. 즉, 여러 레코드가 같은 키를 가질 때, 원래의 순서가 유지됩니다.

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

*blue*에 대한 두 레코드가 원래 순서를 유지해서 ('blue', 1)이 ('blue', 2)보다 앞에 나옴이 보장됨에 유의하십시오.

이 멋진 속성은 일련의 정렬 단계로 복잡한 정렬을 만들 수 있도록 합니다. 예를 들어, 학생 데이터를 *grade*의 내림차순으로 정렬한 다음, *age*의 오름차순으로 정렬하려면, 먼저 *age* 정렬을 수행한 다음 *grade*를 사용하여 다시 정렬합니다:

```
>>> 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)]
```

이것은 다중 패스로 정렬하기 위해 필드와 순서의 튜플 리스트를 받을 수 있는 래퍼 함수로 추상화할 수 있습니다.

```
>>> 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)]
```

파이썬에서 사용된 *Timsort* 알고리즘은 데이터 집합에 이미 존재하는 순서를 활용할 수 있으므로 효율적으로 여러 번의 정렬을 수행합니다.

6 Decorate-Sort-Undecorate

이 관용구는 그것의 세 단계를 따라 장식-정렬-복원(Decorate-Sort-Undecorate)이라고 합니다:

- 첫째, 초기 리스트가 정렬 순서를 제어하는 새로운 값으로 장식됩니다.
- 둘째, 장식된 리스트를 정렬합니다.
- 마지막으로, 장식을 제거하여, 새 순서로 초깃값만 포함하는 리스트를 만듭니다.

예를 들어, DSU 방식을 사용하여 *grade*로 학생 데이터를 정렬하려면 다음과 같이 합니다:

```
>>> 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)]
```

이 관용구는 튜플이 사전식으로 비교되기 때문에 작동합니다; 첫 번째 항목이 비교됩니다; 그들이 같으면 두 번째 항목이 비교되고, 이런 식으로 계속됩니다.

모든 경우에 장식된 리스트에 인덱스 *i*를 포함할 필요는 없지만, 두 가지 이점이 있습니다:

- 정렬이 안정적입니다 – 두 항목이 같은 키를 가지면, 그들의 순서가 정렬된 리스트에 유지됩니다.
- 장식된 튜플의 순서는 최대 처음 두 항목에 의해 결정되므로 원래 항목은 비교 가능할 필요가 없습니다. 그래서 예를 들어, 원래 리스트에는 직접 정렬될 수 없는 복소수가 포함될 수 있습니다.

이 관용구의 또 다른 이름은 펄 프로그래머들 사이에서 이것을 대중화한 Randal L. Schwartz의 이름을 딴 *Schwartzian 변환*입니다.

이제 파이썬 정렬이 키 함수를 제공하기 때문에, 이 기법은 자주 필요하지 않습니다.

7 Comparison Functions

Unlike key functions that return an absolute value for sorting, a comparison function computes the relative ordering for two inputs.

For example, a *balance scale* compares two samples giving a relative ordering: lighter, equal, or heavier. Likewise, a comparison function such as `cmp(a, b)` will return a negative value for less-than, zero if the inputs are equal, or a positive value for greater-than.

It is common to encounter comparison functions when translating algorithms from other languages. Also, some libraries provide comparison functions as part of their API. For example, `locale.strcoll()` is a comparison function.

To accommodate those situations, Python provides `functools.cmp_to_key` to wrap the comparison function to make it usable as a key function:

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

8 Odds and Ends

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function. This is necessary because “alphabetical” sort orderings can vary across cultures even if the underlying alphabet is the same.
- `reverse` 매개 변수는 여전히 정렬 안정성을 유지합니다 (그래서 같은 키를 갖는 레코드는 원래 순서를 유지합니다). 흥미롭게도, 그 효과는 내장 `reversed()` 함수를 두 번 사용하여 매개 변수 없이 흉내 낼 수 있습니다:

```
>>> 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.

- 키 함수는 정렬되는 객체에 직접 의존할 필요가 없습니다. 키 함수는 외부 자원에 액세스할 수도 있습니다. 예를 들어, 학생 성적이 디렉터리에 저장되어 있다면, 학생 이름의 별도 리스트를 정렬하는 데 사용할 수 있습니다:

```
>>> 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.

색인

P

Python 향상 제안
PEP 8, 6