ソートのテクニック

リリース *3.14.0a3*

Guido van Rossum and the Python development team

12 月 21, 2024

目次

著者

Andrew Dalke and Raymond Hettinger

Python のリストにはリストをインプレースに変更する、組み込みメソッド `list.sort()` があります。他にもイテラブルからソートしたリストを作成する組み込み関数 `sorted()` があります。

このドキュメントでは Python を使った様々なソートのテクニックを探索します。

1 ソートの基本

単純な昇順のソートはとても簡単です: `sorted()` 関数を呼ぶだけです。そうすれば、新たにソートされたリストが返されます:

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

`list.sort()` メソッドを呼びだしても同じことができます。この方法はリストをインプレースに変更します (そして sorted との混乱を避けるため `None` を返します)。多くの場合、こちらの方法は `sorted()` と比べると不便です - ただし、元々のリストが不要な場合には、わずかですがより効率的です。

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

違いは他にもあります、`list.sort()` メソッドはリストにのみ定義されています。一方 `sorted()` 関数は任意のイテラブルを受け付けます。

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

2 Key 関数

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']
```

key パラメータの値は関数または呼び出し可能オブジェクトであって、単一の引数をとり、ソートに利用されるキー値を返すものでなければいけません。この制約によりソートを高速に行えます、キー関数は各入力レコードに対してきっちり一回だけ呼び出されるからです。

よくある利用パターンはいくつかの要素から成る対象をインデクスのどれかをキーとしてソートすることです。例えば:

```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
```

(次のページに続く)

(前のページからの続き)

```
... ]
>>> sorted(student_tuples, key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

同じテクニックは名前づけされた属性 (named attributes) を使うことでオブジェクトに対しても動作します。例えば:

```
>>> 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)]
```

名前づけされた属性を持つオブジェクトは上で示したとおり通常のクラスから作れますし、`dataclass` や named tuple のインスタンスでもよいです。

3 operator モジュールの関数や partial 関数による評価

上で示した key function パターンは非常に一般的なので、Python はアクセサ関数を簡単に速く作れる便利な関数を提供しています。`operator` モジュールに `itemgetter()`、`attrgetter()`、および `methodcaller()` 関数があります。

これらの関数を利用すると、上の例はもっと簡単で高速になります:

```
>>> 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 モジュールの関数は複数の段階でのソートを可能にします。例えば、*grade* でソートしてさらに *age* でソートする場合:

```
>>> 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)]
```

functools モジュールには他にもキー関数を作るための便利なツールがあります。partial() 関数は複数引数の関数の アリティ を減らし、キー関数に適したものにできます。

```
>>> 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 昇順と降順

list.sort() と sorted() の両方とも *reverse* パラメータを 真偽値として受け付けます。このパラメータは降順ソートを行うかどうかの フラグとして利用されます。例えば、学生のデータを *age* の逆順で得たい場合:

```
>>> 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 ソートの安定性と複合的なソート

ソートは、安定 (stable) であることが保証されています。これはレコードの中に同じキーがある場合、元々の順序が維持されるということを意味します。

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

二つの *blue* のレコードが元々の順序を維持して、('blue', 1) が ('blue', 2) の前にあることに注意してください。

この素晴しい性質によって複数のソートを段階的に組み合わせることができます。例えば、学生データを

grade の降順にソートし、さらに *age* の昇順にソートしたい場合には、まず *age* でソートし、次に *grade* でもう一度ソートします:

```
>>> 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)]
```

この処理は、リストおよびフィールド名とソート順序のタプルを複数受け取れるラッパー関数へ抽象化できます。

```
>>> 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 では Timsort アルゴリズムが利用されていて、効率良く複数のソートを行うことができます、これは現在のデータセット中のあらゆる順序をそのまま利用できるからです。

6 デコレート - ソート - アンデコレート (DSU)

このイディオムは以下の 3 つのステップにちなんでデコレート-ソート-アンデコレート (Decorate-Sort-Undecorate) と呼ばれています:

- まず、元となるリストをソートしたい順序を制御する新しい値でデコレートします。
- 次に、デコレートしたリストをソートします。
- 最後に、デコレートを取り除き、新しい順序で元々の値のみを持つリストを作ります。

例えば、DSU アプローチを利用して学生データを *grade* でソートする場合:

```
>>> 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)]
```

このイディオムはタプルが辞書編集的に比較されるため正しく動作します; 最初の要素が比較され、同じ場合には第二の要素が比較され、以下も同様に動きます。

デコレートしたリストのインデクス *i* は全ての場合で含まれる必要はありませんが、そうすることで二つの利点があります:

- ソートが安定になります -- もし二つの要素が同じキーを持つ場合、それらの順序がソートされたリストでも維持されます。
- 元々の要素が比較可能な要素を持つとは限りません、なぜならデコレートされたタプルの順序は多くの場合、最初の二つの要素で決定されるからです。例として元のリストは直接比較できない複素数を含むことができます。

このイディオムの別名に Schwartzian transform があります。これは Perl プログラマの間で有名な Randal L. Schwartz にちなんでいます。

いまや Python のソートは key 関数による方法を提供しているので、このテクニックは不要でしょう。

7 比較関数

ソートのための絶対的な値を返すキー関数と違って、2 つの入力を受け取り、その相対的な順序を計算するような関数を比較関数といいます。

例えば、天秤 は 2 つのサンプルを比較し、より軽いか、同じか、より重いかの相対的な順序を与えます。同じように、`cmp(a, b)` などの比較関数はより小さいときは負の値を、入力が等しい場合は 0 を、より大きい場合は正の値を返します。

他のプログラミング言語で書かれたアルゴリズムを Python に書き直すときに比較関数を目にすることがよくあります。また、その API の一部として比較関数を提供しているライブラリもあります。例えば、`locale.strcoll()` は比較関数です。

このような場合に対処するため、Python は比較関数をラップしてキー関数として使えるようにする関数 `functools.cmp_to_key` を提供しています。

```
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]
```

(次のページに続く)

(前のページからの続き)

```
>>> 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 残りのはしばし

- ロケールに対応したソートを行うには、キー関数に `locale.strxfrm()` を使うか、比較関数に `locale.strcoll()` を使ってください。これが必要なのは、同じアルファベットを使っていたとしても、文化が違えば ” アルファベット順” の意味するものが変わることがあるからです。

- *reverse* パラメータはソートの安定性を保ちます (ですから、レコードのキーが等しい場合元々の順序が維持されます)。面白いことにこの影響はパラメータ無しで `reversed()` 関数を二回使うことで模倣することができます:

```
>>> 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
```

(次のページに続く)

(前のページからの続き)

```
>>> standard_way
[('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
```

- ソート関数は、2 つのオブジェクトを比較する際 < を用います。したがって、クラスに標準のソート順序を追加することは `__lt__()` メソッドを定義することで達成できます。

```
>>> Student.__lt__ = lambda self, other: self.age < other.age
>>> sorted(student_objects)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

  ただし、< は `__lt__()` が実装されていなければ `__gt__()` を使って代替することに注意してください (この技法について詳しくは `object.__lt__()` を参照してください)。驚きを防ぐため、**PEP 8** は 6 つの比較メソッドを全て実装することを推奨しています。この作業を簡単にするため、`total_ordering()` デコレータが提供されています。

- key 関数はソートするオブジェクトに依存する必要はありません。key 関数は外部リソースにアクセスすることもできます。例えば学生の成績が辞書に保存されている場合、それを利用して別の学生の名前のリストをソートすることができます:

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

10 部分的なソート

アプリケーションには、データの一部のみが整列されていればよいものがあります。標準ライブラリは完全なソートより少ない作業をするいくつかのツールを提供します。

- `min()` と `max()` はそれぞれ最小値と最大値を返します。これらの関数は入力データを一度だけ通過し、補助的なメモリをほとんど要求しません。

- `heapq.nsmallest()` と `heapq.nlargest()` はそれぞれ最小と最大の値 n 個を返します。これらの関数はデータを一度だけ通過し、一度に n 要素だけを記憶します。入力の数に対して小さい n の値では、これらの関数は完全なソートより比較の回数がはるかに少なくなります。

- `heapq.heappush()` と `heapq.heappop()` は最小の要素がつねに位置 0 になる、部分的にソートされたデータの配列を作り維持します。これらの関数はタスクスケジューリングに一般的に使われる優先度キューを実装するのに適しています。

索引