ソート HOW TO

リリース *2.7.17*

Guido van Rossum
and the Python development team

12 月 24, 2019

Python Software Foundation
Email: docs@python.org

目 次

著者 Andrew Dalke and Raymond Hettinger

リリース 0.1

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。

第1章　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]
```

リストの `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]
```

第2章　Key 関数

Python 2.4 から、`list.sort()` と `sorted()` には *key* パラメータが追加されました、これは比較を行う前にリストの各要素に対して呼び出される関数を指定するパラメータです。

例えば、大文字小文字を区別しない文字列比較の例:

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

key パラメータは単一の引数をとり、ソートに利用される key を返さ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),
... ]
>>> 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)]
```

第3章　operator モジュール関数

上述した key 関数のパターンはとても一般的です、そのため、Python は高速で扱いやすいアクセサ関数を提供しています。 operator モジュールには `operator.itemgetter()`, `operator.attrgetter()`, そして Python 2.5 から利用できるようになった `operator.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)]
```

`operator.methodcaller()` 関数は、オブジェクトが比較される際に呼ばれる固定パラメータでのメソッド呼び出しを作ります。例えば、エクスクラメーションマークの数でメッセージの重要度を計算するのに `str.count()` を使えるでしょう:

```
>>> from operator import methodcaller
>>> messages = ['critical!!!', 'hurry!', 'standby', 'immediate!!']
>>> sorted(messages, key=methodcaller('count', '!'))
['standby', 'hurry!', 'immediate!!', 'critical!!!']
```

第4章　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)]
```

第5章　ソートの安定性と複合的なソート

Python 2.2 からソートは、 stable であることが保証されるようになりました。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* のレコートが元々の順序を維持して、`('blue', 1)` が `('blue', 2)` の前にあること注意してください。

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)]
```

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。

第6章　デコレート-ソート-アンデコレートを利用した古いやり方

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。

巨大なリストや比較の情報を得る計算が高くつくリストに対するソートや Python のバージョンが 2.4 より前の場合には、 DSU はリストをソートするのに最速な方法のようです。 2.4 以降では、key 関数が同じ機能を提供します。

第7章　*cmp* パラメータを利用した古い方法

この HOWTO の内容の多くは Python 2.4 以降を仮定しています。それ以前では組み込み関数 `sorted()` と `list.sort()` はキーワード引数をとりませんでした。その代わりに Py2.x バージョンの全ては、ユーザが比較関数を指定するための *cmp* パラメーターをサポートしました。

Python 3 では *cmp* パラメータは完全に削除されました (ぜいたくな比較と `__cmp__()` マジックメソッドの衝突を除き、言語を単純化しまとめるための多大な労力の一環として)。

Python 2 では `sort()` にオプションとして比較に利用できる関数を与えることができます。関数は比較される二つの引数をとり、小さい場合には負の値を、等しい場合には 0 を、大きい場合には正の値を返さなければいけません。例えば、以下のようにできます:

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

また、比較順を逆にすることもできます:

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

Python 2.x から 3.x にコードを移植する場合、比較関数を持っている場合には key 関数に比較しなければならないような状況に陥るかもしれません。以下のラッパーがそれを簡単にしてくれるでしょう:

```
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
```

key 関数を変換するには、古い比較関数をラップするだけです:

```
>>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
[5, 4, 3, 2, 1]
```

Python 2.7 には、functools モジュールに `functools.cmp_to_key()` 関数が追加されました。

第8章　残りいくつかとまとめ

- ロケールに配慮したソートをするには、キー関数 `locale.strxfrm()` を利用するか、比較関数に `locale.strcoll()` を利用します。

- *reverse* パラメータはソートの安定性を保ちます (つまり、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
>>> standard_way
[('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
```

- あるクラスの標準のソート順を作るには、適切な拡張比較 (rich comparison) メソッドを追加するだけです:

```
>>> 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)]
```

  汎用の目的のためには、推奨されるアプローチは、6 つすべての拡張比較メソッドを定義することです。 `functools.total_ordering()` クラスのデコレータがこれの実装を楽にしてくれます。

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