Sorting, Minimum and Maximum

All Versions

Examples

Make custom classes orderable

min , max , and sorted all need the objects to be orderable. To be properly orderable, the class needs to define all of the 6 methods __lt__ , __gt__ , __ge__ , __le__ , __ne__ and __eq__ :

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

Though implementing all these methods would seem unnecessary, omitting some of them will make your code prone to bugs .

Examples:

```
alist = [IntegerContainer(5), IntegerContainer(3),
         IntegerContainer(10), IntegerContainer(7)
        ]

res = max(alist)
# Out: IntegerContainer(3) - Test greater than IntegerContainer(5)
#      IntegerContainer(10) - Test greater than IntegerContainer(5)
#      IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Out: IntegerContainer(10)

res = min(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Out: IntegerContainer(3)

res = sorted(alist)
# Out: IntegerContainer(3) - Test less than IntegerContainer(5)
#      IntegerContainer(10) - Test less than IntegerContainer(3)
#      IntegerContainer(10) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(5)
#      IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Out: [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]
```

sorted with reverse=True also uses __lt__ :

```
res = sorted(alist, reverse=True)
# Out: IntegerContainer(10) - Test less than IntegerContainer(7)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(10)
#      IntegerContainer(3) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(7)
#      IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Out: [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]
```

But sorted can use __gt__ instead if the default is not implemented:

```
del IntegerContainer.__lt__   # The IntegerContainer no longer implements "less than"

res = min(alist)
# Out: IntegerContainer(5) - Test greater than IntegerContainer(3)
#      IntegerContainer(3) - Test greater than IntegerContainer(10)
#      IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Out: IntegerContainer(3)
```

Sorting methods will raise a TypeError if neither __lt__ nor __gt__ are implemented:

```
del IntegerContainer.__gt__   # The IntegerContainer no longer implements "greater then"

res = min(alist)
```

> TypeError: unorderable types: IntegerContainer() < IntegerContainer()

---

functools.total_ordering decorator can be used simplifying the effort of writing these rich comparison methods. If you decorate your class with total_ordering , you need to implement __eq__ , __ne__ and only one of the __lt__ , __le__ , __ge__ or __gt__ , and the decorator will fill in the rest:

```
import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value


IntegerContainer(5) > IntegerContainer(6)
# Output: IntegerContainer(5) - Test less than IntegerContainer(6)
# Returns: False

IntegerContainer(6) > IntegerContainer(5)
# Output: IntegerContainer(6) - Test less than IntegerContainer(5)
# Output: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Returns True
```

Notice how the > (*greater than*) now ends up calling the *less than* method, and in some cases even the __eq__ method. This also means that if speed is of great importance, you should implement each rich comparison method yourself.

Special case: dictionaries

Getting the minimum or maximum or using sorted depends on iterations over the object. In the case of dict , the iteration is only over the keys:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Output: 'a'
max(adict)
# Output: 'c'
sorted(adict)
# Output: ['a', 'b', 'c']
```

To keep the dictionary structure, you have to iterate over the .items() :

```
min(adict.items())
# Output: ('a', 3)
max(adict.items())
# Output: ('c', 1)
sorted(adict.items())
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

```
# Output: [('a', 3), ('b', 5), ('c', 1)]
```

For sorted , you could create an OrderedDict to keep the sorting while having a dict -like structure:

```
from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Output: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Output: 3
```

By value

Again this is possible using the key argument:

```
min(adict.items(), key=lambda x: x[1])
# Output: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Output: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Output: [('b', 5), ('a', 3), ('c', 1)]
```

Default Argument to max, min

You can't pass an empty sequence into max or min :

```
min([])
```

> ValueError: min() arg is an empty sequence

However, with Python 3, you can pass in the keyword argument default with a value that will be returned if the sequence is empty, instead of raising an exception:

```
max([], default=42)
# Output: 42
max([], default=0)
# Output: 0
```

Extracting N largest or N smallest items from an iterable

To find some number (more than one) of largest or smallest values of an iterable, you can use the nlargest and nsmallest of the heapq module:

```
import heapq

# get 5 largest items from the range

heapq.nlargest(5, range(10))
# Output: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Output: [0, 1, 2, 3, 4]
```

This is much more efficient than sorting the whole iterable and then slicing from the end or beginning. Internally these functions use the binary heap priority queue data structure, which is very efficient for this use case.

Like min , max and sorted , these functions accept the optional key keyword argument, which must be a function that, given an element, returns its sort key.

Here is a program that extracts 1000 longest lines from a file:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Here we open the file, and pass the file handle f to nlargest . Iterating the file yields each line of the file as a separate string; nlargest then passes each element (or line) is passed to the function len to determine its sort key. len , given a string, returns the length of the line in characters.

This only needs storage for a list of 1000 largest lines so far, which can be contrasted with

```
longest_lines = sorted(f, key=len)[1000:]
```

which will have to hold *the entire file in memory* .

Getting a sorted sequence

Using **one** sequence:

```
sorted((7, 2, 1, 5))                # tuple
# Output: [1, 2, 5, 7]

sorted(['c', 'A', 'b'])             # list
# Output: ['A', 'b', 'c']

sorted({11, 8, 1})                  # set
# Output: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15}) # dict
# Output: ['10', '11', '3']         # only iterates over the keys

sorted('bdca')                      # string
# Output: ['a','b','c','d']
```

The result is always a new `list` ; the original data remains unchanged.

Using the key argument

Finding the minimum/maximum of a sequence of sequences is possible:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Output: (0, 10)
```

but if you want to sort by a specific element in each sequence use the `key` -argument:

```
min(list_of_tuples, key=lambda x: x[0])         # Sorting by first element
# Output: (0, 10)

min(list_of_tuples, key=lambda x: x[1])         # Sorting by second element
# Output: (2, 8)

sorted(list_of_tuples, key=lambda x: x[0])      # Sorting by first element (increasing)
# Output: [(0, 10), (1, 15), (2, 8)]

sorted(list_of_tuples, key=lambda x: x[1])      # Sorting by first element
# Output: [(2, 8), (0, 10), (1, 15)]

import operator
# The operator module contains efficient alternatives to the lambda function
max(list_of_tuples, key=operator.itemgetter(0)) # Sorting by first element
# Output: (2, 8)

max(list_of_tuples, key=operator.itemgetter(1)) # Sorting by second element
# Output: (1, 15)

sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Reversed (decreasing)
# Output: [(2, 8), (1, 15), (0, 10)]

sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed(decreasing)
# Output: [(1, 15), (0, 10), (2, 8)]
```

Getting the minimum or maximum of several values

```
min(7,2,1,5)
# Output: 1

max(7,2,1,5)
# Output: 7
```

Minimum and Maximum of a sequence

Getting the minimum of a sequence (iterable) is equivalent of accessing the first element of a `sorted` sequence:

```
min([2, 7, 5])
# Output: 2
sorted([2, 7, 5])[0]
# Output: 2
```

The maximum is a bit more complicated, because `sorted` keeps order and `max` returns the first encountered value. In case there are no duplicates the maximum is the same as the last element of the sorted return:

```
max([2, 7, 5])
# Output: 7
sorted([2, 7, 5])[-1]
# Output: 7
```

But not if there are multiple elements that are evaluated as having the maximum value:

```
class MyClass(object):
    def __init__(self, value, name):
        self.value = value
        self.name = name

    def __lt__(self, other):
        return self.value < other.value

    def __repr__(self):
        return str(self.name)

sorted([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: [second, first, third]
max([MyClass(4, 'first'), MyClass(1, 'second'), MyClass(4, 'third')])
# Output: first
```

Any iterable containing elements that support `<` or `>` operations are allowed.

Syntax

Parameters

Remarks