HowTo/Sorting

Sorting Mini-HOWTO

Original version by Andrew Dalke

Python lists have a built-in `sort()` method. There are many ways to use it to sort a list and there doesn't appear to be a single, central place in the various manuals describing them, so I'll do so here.

Sorting basic data types

A simple ascending sort is easy; just call the `sort()` method of a list.

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

Sort takes an optional function which can be called for doing the comparisons. The default sort routine is equivalent to:

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

where `cmp()` is the built-in function that compares two objects, `x` and `y`, and returns a negative number, 0 or a positive number depending on whether x<y, x==y, or x>y. During the course of the sort the relationships must stay the same for the final list to make sense.

If you want, you can define your own function for the comparison. For integers (and numbers in general) we can do:

```
>>> def numeric_compare(x, y):
>>>    return x-y
>>>
```

Or, more verbosely, but a little more understandable:

```
>>> def numeric_compare(x, y):
>>>    if x>y:
>>>       return 1
>>>    elif x==y:
>>>       return 0
>>>     else: # x<y
>>>         return -1
>>>
>>> a = [5, 2, 3, 1, 4]
>>> a.sort(numeric_compare)
>>> print a
[1, 2, 3, 4, 5]
```

By the way, this function won't work if the result of the subtraction is out of range, as in `sys.maxint - (-1)`.

Or, if you don't want to define a new named function you can create an anonymous one using `lambda`, as in:

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort(lambda x, y: x-y)
>>> print a
[1, 2, 3, 4, 5]
```

Python 2.4 adds three keyword arguments to `sort()` that simplify many common usages: `cmp`, `key`, and `reverse`. The `cmp` keyword is for providing a sorting function; the previous examples could be written as:

```
>>> a.sort(cmp=numeric_compare)
>>> a.sort(cmp=lambda x,y: x-y)
```

The `reverse` parameter is a Boolean value; if it's true, the list is sorted into reverse order.

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

For Python versions before 2.4, you can reverse the sense of the comparison function:

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

(a more general implementation could return `cmp(y,x)` or `-cmp(x,y)`).

However, it's faster if Python doesn't have to call a function for every comparison, so the most efficient solution is to do the forward sort first, then use the `reverse()` method.

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

Sorting by keys

Python 2.4's `key` parameter lets you derive a sorting key for each element of the list, and then sort using the key.

For example, here's a case-insensitive string comparison:

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

The value of the `key` parameter should be a function that takes a single argument and returns a key to use for sorting purposes.

Often there's a built-in that will match your needs, such as `string.lower()`. The `operator` module contains a number of functions useful for this purpose. For example, you can sort tuples based on their second element using `operator.itemgetter()`:

```
>>> import operator
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3)]
>>> map(operator.itemgetter(0), L)
['c', 'd', 'a', 'b']
>>> map(operator.itemgetter(1), L)
[2, 1, 4, 3]
>>> sorted(L, key=operator.itemgetter(1))
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

Versions of Python before 2.4 don't have the convenient `key` parameter of `sort()`, so you have to write a comparison function that embodies the key-generating logic:

```
>>> a = "This is a test string from Andrew".split()
>>> a.sort(lambda x, y: cmp(x.lower(), y.lower()))
>>> print a
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

This goes through the overhead of converting a word to lower case every time it must be compared, roughly O(n lg n) times. Python 2.4's `key` parameter is called once for each item in the list, which is O(n) and therefore more efficient. You can manually perform the same optimization by computing the keys once and using those values to control the sort order:

```
>>> words = "This is a test string from Andrew.".split()
>>> deco = [ (word.lower(), i, word) for i, word in enumerate(words) ]
>>> deco.sort()
>>> new_words = [ word for _, _, word in deco ]
>>> print new_words
['a', 'Andrew.', 'from', 'is', 'string', 'test', 'This']
```

This idiom is called Decorate-Sort-Undecorate after its three steps:

- First, the initial list is decorated with new values that control the sort order.
- Second, the decorated list is sorted.
- Finally, the decorations are removed, creating a list that contains only the initial values in the new order.

This idiom works because tuples are compared lexicographically; the first items are compared; if they are the same then the second items are compared, and so on.

It is not strictly necessary in all cases to include the index `i` in the decorated list. Including it gives two benefits:

- The sort is stable - if two items have the same key, their order will be preserved in the sorted list.
- The original items do not have to be comparable because the ordering of the decorated tuples will be determined by at most the first two items. So for example the original list could contain `complex` numbers which cannot be sorted directly.

Another name for this idiom is Schwartzian transform, after Randal L. Schwartz, who popularized it among Perl programmers.

For large lists and lists where the comparison information is expensive to calculate, and Python versions < 2.4, DSU is likely to be the fastest way to sort the list.

Comparing classes

The comparison for two basic data types, like ints to ints or string to string, is built into Python and makes sense. There is a default way to compare class instances, but the default manner isn't usually very useful. You can define your own comparison with the `__cmp__` method, as in:

```
>>> class Spam:
>>>     def __init__(self, spam, eggs):
>>>         self.spam = spam
>>>         self.eggs = eggs
>>>     def __cmp__(self, other):
>>>         return cmp(self.spam+self.eggs, other.spam+other.eggs)
>>>     def __str__(self):
>>>         return str(self.spam + self.eggs)
>>>
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4, 6)]
>>> a.sort()
>>> for spam in a:
>>>   print str(spam)
5
10
12
```

Sometimes you may want to sort by a specific attribute of a class. If appropriate you should just define the `__cmp__` method to compare those values, but you cannot do this if you want to compare between different attributes at different times.

Python 2.4 has an `operator.attrgetter()` function that makes this easy:

```
>>> import operator
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4, 6)]
>>> a.sort(key=operator.attrgetter('eggs'))
>>> for spam in a:
>>>   print spam.eggs, str(spam)
3 12
4 5
```

```
6 10
```

In Python 2.4 if you don't want to import the operator module you can:

```
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4, 6)]
>>> a.sort(key=lambda obj:obj.eggs)
>>> for spam in a:
>>>   print spam.eggs, str(spam)
3 12
4 5
6 10
```

Again, earlier Python version require you to go back to passing a comparison function to sort, as in:

```
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4, 6)]
>>> a.sort(lambda x, y: cmp(x.eggs, y.eggs))
>>> for spam in a:
>>>   print spam.eggs, str(spam)
3 12
4 5
6 10
```

If you want to compare two arbitrary attributes (and aren't overly concerned about performance) you can even define your own comparison function object. This uses the ability of a class instance to emulate an function by defining the `__call__` method, as in:

```
>>> class CmpAttr:
>>>     def __init__(self, attr):
>>>         self.attr = attr
>>>     def __call__(self, x, y):
>>>         return cmp(getattr(x, self.attr), getattr(y, self.attr))
>>>
>>> a = [Spam(1, 4), Spam(9, 3), Spam(4,6)]
>>> a.sort(CmpAttr("spam"))  # sort by the "spam" attribute
>>> for spam in a:
>>>    print spam.spam, spam.eggs, str(spam)
1 4 5
4 6 10
9 3 12

>>> a.sort(CmpAttr("eggs"))   # re-sort by the "eggs" attribute
>>> for spam in a:
>>>    print spam.spam, spam.eggs, str(spam)
9 3 12
1 4 5
4 6 10
```

Of course, if you want a faster sort you can extract the attributes into an intermediate list and sort that list.

So, there you have it; about a half-dozen different ways to define how to sort a list:

1. sort using the default method

2. sort using a comparison function

3. reverse sort not using a comparison function

4. sort on an intermediate list (two forms)
5. sort using class defined cmp method
6. sort using a sort function object

Topics to be covered

- Rich comparisons
    - With custom comparisons, it is possible to create insane combinations, such as `((x<y) and (x==y))` or `((x<y) and not (x<=y))`.
    - The most important comparisons are eq (but be sure to update hash as well), and lt (which is used by the sorting algorithms in practice).
- Sorting stability
    - Python sorts are stable. Guido has indicated that this is a promise of the language. Therefore, if x == y, sorted ([x, y]) returns [x, y] but sorted([y, x]) returns [y, x].
- The sorted() function
    - It takes any iterable, and returns a sorted version. If the items in the iterable are not sensibly compared, it will still return a canonical ordering, unless someone went out of their way to prevent one.

See Also

- SortingListsOfDictionaries