Sorting it all out

Using Ruby's #sort and #sort_by methods on arrays and hashes

Oct. 11, 2015

Like all collection classes in Ruby, the `Array` and `Hash` classes include the Enumerable module. Enumerable bestows upon classes that include it a wide selection of methods which iterate over collections in many useful ways. In order to include the Enumerable module and use its methods, a collection class must include its own version of the `#each` method, which provides the Enumerable methods with a basic strategy for iterating over collection objects of that class. In order to function at all, every method in Enumerable must call the class's `#each` method.

One of the most common operations performed on collections is to sort them according to various criteria. This commonly done using the `#sort` and `#sort_by` methods, which are found in the Enumerable module. This can be confusing, because when you are looking for a method to solve a problem, you may not find it among the methods listed in the Ruby docs for the class you expect. On first glance, `Hash` seems to have *no* sorting methods, but in fact, `Hash` (like collection classes `Range` and `Set`) uses the base Enumerable versions of `#sort` and `#sort_by`—you just have to go to the Enumerable page to find them. `Array`, on the other hand, has its own custom versions of `#sort` and `#sort_by`, which *do* appear on the list of `Array` methods on Ruby-doc.org. The take-away from this is that when looking for a method to solve a problem, it's important to also check the list for any module the class includes.

Sorting arrays

Although the `sort` and `#sort_by` methods for the `Array` class override the default `Enumerable` versions, we needn't worry about under-the-hood stuff like that here. For our purposes, we can call them on an array object and get a perfectly usable return no matter where the methods themselves are stored. In its simplest form, `Array#sort` returns a new array, rearranged in alphanumeric order, while leaving the original array unaltered:

```
$ ["oak", "pine", "maple", "elm"].sort
=> ["elm", "maple", "oak", "pine"]
$ [4, 17, 2, 9.1, 11, 6].sort
```

This works fine, so long as all of the items in the array are of the same type, and can be compared directly. That is, `#sort` works as long as the items in the collection are either *all* strings or *all* numbers (`#sort` can handle a mix of integers and floats, though). If you have a mixture of numbers and strings in your array, `#sort` will raise an `ArgumentError`.

You can `#sort` an array of arrays, but `#sort` only considers the value of the first item in each member array when sorting. The other items in the member arrays are ignored. This means that `#sort` will leave [3, 6] and [3, 2] in that order, if that's how they appear in the original array. But it also means that member arrays do not have to be of the same length, and that only the first items in each member array need to be of matching type. The second and subsequent items can be a mixture of strings, numbers, or other objects. Here are some examples of sorted arrays:

```
$ [[4, 7], [2, 1], [8, 6]].sort
=> [[2, 1], [4, 7], [8, 6]]

[[7, 14], [2], [5, 4, 9]].sort
=> [[2], [5, 4, 9], [7, 14]]

$ [["fazz", 7], ["bargle", ["a", "z"]], ["goo",
6]].sort
=> [["bargle", ["a", "z"]], ["fazz", 7], ["goo",
6]]
```

Other sorts of #sorts

If you need to sort an array by some criterion other than alphanumeric order—perhaps by length—you'll need to tell `#sort` how you want to compare the items in the array by passing a code block to the method that defines the comparison for your sort. The syntax for the code block is `{ |item_1, item_2| item_1_code <=> item_2_code }`. In this example, we're calling `#length` on each element for the sort:

```
$ ["fanz","dru", "ba"].sort{|a,b| a.length <=>
b.length }
=> ["ba", "dru", "fanz"]
```

As you can imagine, the code block for a `#sort` can become unweildy very quickly. For non-alphanumeric sorts, you might prefer the `#sort_by` method, which lets you define how to evaluate each item to be sorted just once, and eliminates the spaceship comparison operator. The syntax for the `#sort_by` code block is simply `{ |item| item_code }`, so the block for our `#length`-based sort becomes `{ |a| a.size }`. If you're calling a single method on your objects, as we are here, there's also a Ruby shorthand syntax to make it even shorter. You can use an ampersand to represent the object, and the symbol form of

the method name: `(&:size)`. Here we see both versions of `#sort_by` in action:

```
$ ["glo", "mambo", "iz", "thoo"].sort_by{ |a|
a.size }
=> ["iz", "glo", "thoo", "mambo"]

$ [[3, 6, 5], [9], [8, 7, 6, 4]].sort_by(&:size)
=> [[9], [3, 6, 5], [8, 7, 6, 4]]
```

Mixed #sorts

If you need to sort an array with members that are not comparable—a mix of strings and numbers, most likely—a simple `#sort` will fail, raising an `ArgumentError`. You can use `#sort_by` in conjunction with `#to_s` to sort a mixed array, but sorting numbers as strings will put **25** before **8**; alphanumeric sorting has no awareness of place value. To properly perform an alphanumeric sort on an array of mixed numbers and strings, you can use `#partition` to sift the numbers and strings into two separate sub-arrays, sort each sub-array separately, and then use `#flatten` to turn them back into a single array.

```
# Using #sort_by and #to_s…
$ ["r",7,"gobs",3].sort_by(&:to_s)
=> [3, 7, "gobs", "r"]

$ ["r",7,"gobs",356].sort_by(&:to_s)
=> [356, 7, "gobs", "r"]

# Using #partition, two #sorts, and #flatten…
$ array = ["zoi", 5, "trak", 2.3, 144, "ak"]
$ array = array.partition{|a| a.is_a? Numeric}
$ array[0].sort!
$ array[1].sort!
$ array.flatten!
=> [2.3, 5, 144, "ak", "trak", "zoi"]
```

Note that the first example works because the array contains only single-digit numbers. When we make one of the numbers three digits long, in the second example, sorting numbers as strings no longer works—7 should come before 356, but it doesn't. Negative numbers and floats would further confuse a `#sort` by strings. The third example is more complex, but it sorts numbers and strings correctly according to alphanumeric rules, no matter how many digits are in the numbers. As the sorting criterion for `#partition`, we pass a code block that checks if each item is a member of the class `Numeric`; this catches all numbers of all classes, including `Floats`, `Fixnums`, and `Bignums`. The first `#sort` sorts only the numbers, and the second sorts only the strings. We turn it all back into a single array by calling `#flatten` at the end.

The algorithm above is set up to sort the existing array in place. If you need a non-destructive version, you can write a method like this:

```
# Non-destructive method; returns a new sorted
array.
def alphanumeric_sort(array)
  sorted_array = array.clone
  sorted_array = sorted_array.partition do |a|
    a.is_a? Numeric
  end
  sorted_array[0] = sorted_array[0].sort
  sorted_array[1] =
sorted_array[1].sort_by(&:downcase)
  sorted_array.flatten
end

test_array = ["wap", 7, "Pron", 153, 7.89, "ig"]

alphanumeric_sort(test_array)
=> [7, 7.89, 153, "ig", "Pron", "wap"]
```

This `#alphanumeric_sort` method begins with the `Object#clone` method, which creates an identical but separate array object to work on and return—that's why it's non-destructive. Not only does it handle positive and negative integers as well as floats, but because it's non-destructive, we can use `#sort_by` and `#downcase` to sort the strings sub-array. This makes the method sort capital and lower-case letters together, like a dictionary. Note that I switched to a multi-line, `do/end` syntax for the `#partition` call simply to shorten the lines and avoid a confusing line-wrap on this web page—there's no reason not to use the single-line, curly-braces syntax for that code block in a real text editor.

Sorting hashes

In some ways, sorting hashes with `#sort` is trickier than arrays, because each member of a hash is a pair of values—a matched key and value. By default, `#sort` reorders the members of a hash alphanumerically, by their keys, like this:

```
test_hash = {
  toyota: "Corolla",
  ford: "Mustang",
  chevrolet: "Volt",
  dodge: "Charger"
}

test_hash.sort
=> [[:chevrolet, "Malibu"], [:dodge, "Charger"],
[:ford, "F-150"], [:toyota, "Corolla"]]
```

Not only are hashes sorted by their keys, but they aren't returned in a hash! Instead, they come back as a two-dimensional array, because that's how `#sort` (and all the rest of the `Enumerable` methods) handles them. We can easily turn the returned array back into a hash by adding `#to_h` after the `#sort` call, though:

```
test_hash.sort.to_h
=> {:chevrolet=>"Malibu", :dodge=>"Charger",
:ford=>"F-150", :toyota=>"Corolla"}
```

As with arrays, `#sort` works on hashes so long as the keys are all the same data type. Usually, the keys in a hash will all be symbols, or all be strings; `#sort` also works if they are all numbers, too, though this isn't very common in real programs. Here we see a hash of the households on Clue Street, with house numbers as keys and residents' surnames as values:

```
clue_street_homes = {
  67 => "Plum",
  145 => "Mustard",
  13 => "Scarlet",
  119 => "Green"
}

clue_street_homes.sort.to_h
=> {13=>"Scarlet", 67=>"Plum", 119=>"Green",
145=>"Mustard"}
```

It's pretty unlikely that you'll have a hash with keys of mixed data types, but if you do, a simple `#sort` won't work on it—it will raise the same `ArgumentError` we got above, with arrays. As long as the keys are what you want to sort by, you can use the same techniques I described for arrays. If you need to sort the values, however, the easiest way to do that is with `#sort_by`.

Sorting by hash values

The easiest way to sort a hash by the value in its key-value pairs is to use the `#sort_by` method. To make `#sort_by` use the values to sort the pairs, use this code block: `{ |key,value| value }`. This tells `#sort_by` to perform its default alphanumeric sort on the hash, based on the values in the key-value pairs instead of the keys. If you have a different search criterion, such as the length of the values, you can add the appropriate method to the code block as well. In this example, I also added `#to_h` to turn the two-dimensional array returned by `#sort_by` back into a hash, as I did in the previous example.

```
capitals = {
  texas: "Austin",
  california: "Sacremento",
  florida: "Tallahassee"
```

```
}

# Sorted alphanumerically (default) by values
capitals.sort_by{ |k,v| v }.to_h
=> {:new_york=>"Albany", :texas=>"Austin",
:california=>"Sacremento", :florida=>"Tallahassee"}

# Sorted by value length
capitals.sort_by{ |k,v| v.length }.to_h
=> {:texas=>"Austin", :new_york=>"Albany",
:california=>"Sacremento", :florida=>"Tallahassee"}
```

Stupid #sort tricks

Since we're on the subject of sorting collections, there are two `Array` methods that I think of as being connected with `#sort`, at least conceptually if not technically. These are `Array#reverse` and `Array#shuffle`. Each of these methods does to an array exactly what you expect it to: `#reverse` reverses the order, and `#shuffle` randomizes the order. Here they are in action:

```
# A normal sort, as a baseline
$ [3, 76, 19, 5, 190, 42, 81, 150].sort
=> [3, 5, 19, 42, 76, 81, 150, 190]

# A one-line sort-and-reverse
$ [3, 76, 19, 5, 190, 42, 81, 150].sort.reverse
=> [190, 150, 81, 76, 42, 19, 5, 3]

# A shuffle of the same array
$ [3, 76, 19, 5, 190, 42, 81, 150].shuffle
=> [19, 42, 150, 3, 76, 190, 81, 5]
```

If you have an unsorted set, and you need to access it from largest to smallest, there's no reason not to put the `#sort` and the `#reverse` on the same line—doing so is more concise and more clear than writing the two calls on separate lines of code. Of course, if you're shuffling your array, there's no reason sort it first. I include it here because I think of `#sort` and `#shuffle` as being opposites. If you want to assign your students to study groups in alphabetic order, sort them before you allocate them; if you want them assigned randomly, shuffle them first.

`#reverse` and `#shuffle` are strictly `Array` methods, not `Enumerable` methods, so you can't use them directly on hashes, sets, or ranges. But it's not hard to use `#to_a` to convert a hash or set into an array in order to reverse it or shuffle, and then convert it back to its original class with `#to_h` or `#to_s`. Since a range is an ascending sequence of integers by definition, there's not much point in reversing or shuffling it. You might use a range in conjunction with a loop to populate a large array of numbers without typing in every member, and then reverse or shuffle it, though.