ruby's <=> operator and sort method

Asked 10 years, 1 month ago Modified 10 months ago Viewed 17k times

9

```
player1 = Player.new("moe")
player2 = Player.new("larry",60)
player3 = Player.new("curly", 125)
@players = [player1, player2, player3]
```

Above, I created some player objects and added them to the previously empty array @players.

Then, I redefined <=> to be this:

```
def <=>(other)
    other.score <=> score
end
```

I then can run this code

```
@players.sort
```

and my array of player objects in @players are sorted from high score to low score. I guess this looks a bit black boxy to me. I am a bit unclear what's going on here. How do I find out what is going on behind the scenes?

All I know is that if you take two values and use the spaceship operator/general comparison operator:

```
2 <=> 1
=> 1

1 <=> 2
=> -1

1 <=> 1
=>0
```

Sometimes, it seems like Ruby has a lot of lower level stuff going on that I can't see at the high level that I am programming in. This seems natural... but this case seems especially removed from the lower level goings on of the sort method. How does sort use the spaceship operator? Why does redefining the spaceship operator in the way that we did allow us to sort objects now?

ruby spaceship-operator

Share Improve this question Follow

asked Oct 27, 2014 at 5:26

Jwan622

11.6k ● 22 ● 101 ● 206

1 Have you read the documentation? – mu is too short Oct 27, 2014 at 5:58

possible duplicate of How does Ruby's sort method work with the combined comparison (spaceship) operator? – lulalala Oct 27, 2014 at 6:54

3 Answers

Sorted by: Highest score (default)

56

Before you can understand sorting objects. You need to understand the .sort method in Ruby. If you were to sort 5 cards with numbers on them, you could take a look at all of them, find the lowest one easily, and just pick that one as your first card (presuming you're sorting from lowest to highest, which Ruby does). When your brain sorts, it can look at everything and sort from there.

There are 2 main elements of confusion here that are rarely addressed:

1. Ruby can't sort in the way you think of the word "sort". Ruby can only 'swap' array elements, and it can 'compare' array elements.
2. Ruby uses a comparison operator, called the spaceship, to attribute numbers to help it 'sort'. Those numbers are -1,0,1. People erroneously think those 3 numbers are helping it 'sort' (eg. if there was an array with 3 numbers such a 10,20,30, then the 10 would be a -1, the 20 a 0, and the 30 a 1, and Ruby is just simplifying the sorting by reducing it to -1,0,1. This is wrong. Ruby can't "sort". It can't only compare).

Look at the spaceship operator. It's 3 individual operators lumped into one, the <, the =, and the >. When Ruby compares two variables, it results in one of these numbers.

![Spaceship Operator][1]

That said, what does "results" mean? It DOESN'T mean that one of the variables is assigned a 0,1,-1. It simply is the way that Ruby can take two variables and do something with them. Now, if you just run:

```
puts 4 <=> 5
```

You'll get the result of -1, since whatever 'part' (eg. <, =, or >) of the comparison operator (spaceship) is true, gets the number that's assigned to it (as seen in the above picture). When Ruby sees this <=> with an array though, it has 2 things it will do to the array only: Leave the array alone OR swap the elements of the array.

If Ruby uses the <=> and gets a 1, it will swap the 2 elements of the array. If Ruby gets a result of -1 or 0, it will leave the array alone.

An example is if Ruby sees the array [2,1]. The sort method would make it pull in these figures like 2<=>1. Since the part of the spaceship (if you want to think of it like that) that's true is the > (ie. 2>1 is true), the result is '1' from Ruby. When Ruby sees a 1 result from the spaceship, it *swaps* the 2 elements of the array. Now the array is [1,2].

Hopefully at this point, you see that Ruby only compares with the <=> operator, and then swaps (or leaves alone) the 2 elements in the array it compares.

Understand the .sort method is an iterative method, meaning that it's a method that runs a block of code many times. Most people are introduced to the .sort method only after they've seen a methods such as .each or .upto (you don't need to know what those do if you haven't heard of them), but those methods run through the array 1 time ONLY. The .sort method is different in that it will run through your array as many times as it needs to so that it's sorted (by sorted, we mean compared and swapped).

To make sure you understand the Ruby syntax:

```
foo = [4, 5, 6]
puts foo.sort {|a,b| a <=> b}
```

The block of code (surrounded by {}'s) is what Ruby would do any way when it sorts from lowest to highest. But suffice it to say that the first iteration of the .sort method will assign the variables between the pipes (a, b) the first two elements of the array. So for the first iteration a=4 and b=5, and since 4<5, that results in a -1, which Ruby takes it to mean to NOT swap the array. It does this for a second iteration, meaning a=5 and b=6, sees that 5<6, results in -1 and leaves the array alone. Since all the <=> results were -1, Ruby stops looping through and feels the array is sorted at [4,5,6].

We can sort from high to low by simply swapping the order of the variables.

```
bar = [5, 1, 9]
puts bar.sort {|a,b| b <=> a}
```

Here's what Ruby is doing:

Iteration 1: Array [**5,1**,9]. a=5, b=1. Ruby sees the b<=>a, and says is 1 < 5? Yes. That results in -1. Stay the same.

Iteration 2: Array [5,**1,9**]. a=1, b=9. Ruby sees the b<=>a, and says is 9 < 1? No. That results in 1. Swap the 2 array elements. The array is now [5,9,1]

Iteration 3: Array [**5,9**,1]. Starting over b/c there was a +1 result in the array before going through it all. a=5, b=9. Ruby sees the b<=>a, says is 9<5? No. That results in 1. Swap. [9, 5, 1]

Iteration 4: Array [9,**5,1**]. a=5, b=1. Ruby sees the b<=>a, says is 1<5? Yes. That results in -1. Therefore, no swapping is performed. Done. [9,5,1].

Imagine an array with the number 50 for the first 999 elements, and a 1 for element 1000. You fully understand the sort method if you realize Ruby has got to go through this array thousands of times doing the same simple compare and swap routine to shift that [1] all the way to the beginning of the array.

Now, we can finally look at .sort when comes to an object.

```
def <=>(other)
    other.score <=> score
end
```

This should now make a little more sense. When the .sort method is called on an object, like when you ran the:

```
@players.sort
```

it pulls up the "def <=>" method with the parameter (eg. 'other') which has the current object from @players (eg. 'whatever the current instance object is of '@players', since it's the sort method, it's eventually going to go through all of the elements of the '@players' array). It's just like when you try to run the puts method on a class, it automatically calls the to_s method inside that class. Same thing for the .sort method automatically looking for the <=> method.

Looking at the code inside of the <=> method, there must be a .score instance variable (with an accessor method) or simply a .score method in that class. And the result of that .score method should (hopefully) be a String or number - the 2 things ruby can 'sort'. If it's a number, then Ruby uses it's <=> 'sort' operation to rearrange all of those objects, now that it knows what part of those objects to sort (in this case, it's the result of the .score method or instance variable).

As a final tidbit, Ruby sorts alphabetically by converting it to numerical values as well. It just considers any letter to be assigned the code from ASCII (meaning since upper

case letters have lower numerical values on the ASCII code chart, upper case will be sorted by default to be first).

Hope this helps! [1]: https://i.sstatic.net/CpwOf.jpg

Share
Improve this answer
Follow

edited Feb 1 at 21:52
lagoupo
99 • 10

answered Jan 18, 2015 at 20:31

Tony DiNitto
1,229 • 1 • 16 • 29

4 super explanation loved it. – thinkingmonster Mar 5, 2016 at 13:16

You forget one thing: to use <=> method in some class you should include Comparable module in it. – kaleb4eg Mar 24, 2017 at 11:25

3

In your example

```
@players.sort
```

is equivalent to

```
@players.sort { |x, y| x <=> y }
```

The elements are sorted depending on the return of the `<=>` method. If `<=>` returns `-1` the first element gets sorted before the second, if it returns `1` the second is sorted before the first. If you change the return value (e.g. swap the elements) than the order changes according to the return values.

Share
Improve this answer
Follow

edited Oct 27, 2014 at 5:52

answered Oct 27, 2014 at 5:42

spickermann
107k • 9 • 112 • 143

2

sort is actually an Enumerable method which relies on the implementation of `<=>`. From Ruby doc itself:

> If Enumerable#max, #min, or #sort is used, the objects in the collection must also implement a meaningful <=> operator, as these methods rely on an ordering between members of the collection.

Try it yourself:

```ruby
class Player
  attr_accessor :name, :score

  def initialize(name, score=0)
    @name = name
    @score = score
  end

  def <=> other
    puts caller[0].inspect
    other.score <=> score
  end
end

player1 = Player.new("moe")
player2 = Player.new("larry",60)
player3 = Player.new("curly", 125)
@players = [player1, player2, player3]
puts @players.sort.inspect

#=> "player.rb:19:in `sort'"
#=> "player.rb:19:in `sort'"
#=> [#<Player:0x007fe87184bbb8 @name="curly", @score=125>, #
<Player:0x007fe87184bc08 @name="larry", @score=60>, #<Player:0x007fe87184bc58
@name="moe", @score=0>]
```

You see, when we use `sort` on `@players` array, the object of `Player` is called with `<=>`, if you do not implement it, then you'll probably get:

> player.rb:14:in `sort': comparison of Player with Player failed (ArgumentError) from player.rb:14:in'

Which makes sense, as object doesn't know how to deal with `<=>`.

Share

Improve this answer

Follow

edited Oct 27, 2014 at 6:08

answered Oct 27, 2014 at 6:03

Surya

16k ● 3 ● 54 ● 76