Ruby trunk - Feature #5534

Redefine Range class and introduce RelativeNumeric and RelativeRange

11/01/2011 08:04 PM - alexeymuranov (Alexey Muranov)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:	2.6	

Description

I started by commenting on Feature #4541, but ended up with proposing a new feature myself.

I suggest to redefine the behavior of Range class so that all empty ranges be equal:

(2..1) == (1..-1) and (2..1) == (1...1) and (2..1) == ('z'..'a') # => true

In other fords, ranges r1 and r2 should be equal if and only if r1.include? and r2.include? give identical results for all inputs. (Why is it not includes? by the way?) Thus Range would simply be a way to store certain infinite sets.

This change will result in not being able to slice an array a from beginning and from the end simultaneously with a[1..-2]. To resolve this, i propose to introduce RelativeNumeric and RelativeRange classes.

Each RelativeNumeric would be a Numeric with an "anchor", which is an arbitrary symbol. For example:

3.from(:bottom) # would return a "relative" 3 with "anchor" :bottom

One can define shortcuts #from_bottom for #from(:bottom) and #from_top for #from_top.

A RelativeRange is a range with relative bounds. If bounds of a relative range r are relative to the same anchor and the range is seen to be empty, it should be equal to *the* empty relative range with this anchor. For example:

(3.from(:center)..2.from(:center)) == (0.from(:center)...0.from(:center)) # => true

Now, to do what is currently done by a[1..-2], one can redefine Array#slice to use instead:

a[1.from_bottom..(-1).from_top]

What do you think?

History

#1 - 11/02/2011 06:44 AM - alexeymuranov (Alexey Muranov)

By the way, i do not understand the use of ranges in conditional expressions in loops, and do not know how to be consistent about it:
http://www.techotopia.com/index.php/Ruby_Ranges#Ruby_Ranges_as_Conditional_Expressions
http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_expressions.html#S6

#2 - 11/06/2011 05:26 PM - trans (Thomas Sawyer)

What is the point of all this? I mean the equality thing is somewhat interesting but how is it really useful? And why does it have any bearing at all on slicing arrays? Really I for one can't make much sense of what you are proposing.

#3 - 11/06/2011 07:27 PM - alexeymuranov (Alexey Muranov)

I was not precise, i meant not equality, but *identity* --- that there be only one empty range: Range::EMPTY_SET. Like this:

(2..1) # => Range::EMPTY_SET

This way it would be clear what a range is --- just an infinite set of a certain form, with methods to work with it.

I am in favor of reducing the number of internal attributes of class instances when possible and simplifying their definitions. In my opinion, this should help to understand what a program is doing, to avoid side effects, to write specifications (and hence to maintain existing behavior in future versions), and to settle more easily on desire behavior for new methods.

Mathematically, ranges (2..1) et (3..0) are equal --- both empty, but Ruby remembers their "bounds" and treats them differently.

If my proposed definition of Range is accepted, the questions like the one discussed in #4541 will simply not arise: since the ranges (4..1) and

(3..1) will be identical, [1,2,3][3..1] and [1,2,3][4..1] will be giving identical result, whatever the result is.

I propose to view the "memory" of bounds of an empty range as an artifact of implementation of Range , which should not be used for operations like a[1..-2] .

My more general view point is the following: i think it will help to settle on some kind of standards for the language if reasonable relations between different methods are enforced. It is already done in cases where one method uses another, for example Comparable#> , Comparable#< , etc., use #<=> :

a < b if and only if (a <=> b) == -1

I think it would be helpful for understanding the language and coming up with some kind of standards or best practice guidelines if similar relations were imposed between methods that cannot be defined in terms of one another simply because of computer limitations (not being able to loop over infinitely many objects). For example, require in specifications that (unless overridden in a subclass) for two ranges x and y ,

x == y must be true if and only if x.include?(z) == y.include?(z) is true for every possible object z.

#4 - 11/07/2011 12:41 AM - trans (Thomas Sawyer)

"This way it would be clear what a range is --- just an infinite set of a certain form, with methods to work with it."

How is that a Range at all? A Range is only "infinite" if one of the sentinels is infinite.

"Mathematically, ranges (2..1) et (3..0) are equal --- both empty, but Ruby remembers their "bounds" and treats them differently."

How is a Range to work without remembering the bounds?

If anything I wish Ruby would understand "counting down" ranges. Currently (3..1).to_a returns [].

"x == y must be true if and only if x.include?(z) == y.include?(z) is true for every possible object z."

For better or worse, Range serves a double (perhaps triple) purpose, e.g. it can serve as interval or it can serve as a way to define a sequential set. So the meaning of #== might not be considered that same in these cases. So Range was given a general definition that works for who it is designed --based on the range bounds. To do as you suggest would, I thing require separating Range into Interval and Sequence classes, so to speak, where you definition of #== is more fitting a Sequence, albeit one might still argument that the order of sequence should be significant too.

#5 - 11/07/2011 01:16 AM - alexeymuranov (Alexey Muranov)

Thomas Sawyer wrote:

> "This way it would be clear what a range is --- just an infinite set of a certain form, with methods to work with it."
>
> How is that a Range at all? A Range is only "infinite" if one of the sentinels is infinite.

There are infinitely many numbers, even if you only count rationals, in the range (1..2)

> "Mathematically, ranges (2..1) et (3..0) are equal --- both empty, but Ruby remembers their "bounds" and treats them differently."
>
> How is a Range to work without remembering the bounds?

It has to remember its bounds to work unless it is empty. If it is empty, and if my proposal to view it as a set is accepted, then it has no bounds (the empty set has no bounds), and methods should work like this:

(2..1).begin # => nil
(2..1).end   # => nil
(2..1)   # => Range::EMPTY_SET

> If anything I wish Ruby would understand "counting down" ranges. Currently (3..1).to_a returns [].

What you suggest here is similar to another alternative which i would accept too, but which i didn't mention because it looked to me even father from the current definition of Range: to allow "oriented" ranges. If ranges are oriented, then 3..1 is the same as 1..3, but with negative orientation. In that case one should get

(1..3).include?(2) # => true
(3..1).include?(2) # => true

Then 3..1 is not empty, and can be used to slice an array like this:

[1,2,3,4][3..1] # => [4,3,2]

> "x == y must be true if and only if x.include?(z) == y.include?(z) is true for every possible object z."

For better or worse, Range serves a double (perhaps triple) purpose, e.g. it can serve as interval or it can serve as a way to define a sequential set. So the meaning of #== might not be considered that same in these cases. So Range was given a general definition that works for who it is designed --based on the range bounds. To do as you suggest would, I thing require separating Range into Interval and Sequence classes, so to speak, where you definition of #== is more fitting a Sequence, albeit one might still argument that the order of sequence should be significant too.

Can you be more precise here please? Depending on the context and the goal, it can be "good" or "bad" that the same class serves "double or triple purpose". Which purposes a range viewed as (possibly oriented) interval cannot serve, other than slicing an array with a[1..-1] ?

My proposal is mostly about simplifying specifications and readability, to avoid questions like #4541 and make code more human-oriented --- from my point of view.

#6 - 11/07/2011 01:38 AM - trans (Thomas Sawyer)

"There are infinitely many numbers, even if you only count rationals, in the range (1..2)"

Ah, okay. See, I think that's a good case in point. I was thinking of range as a sequence, while you were thinking of it an interval.

Well, I think the only good thing about the multi-purpose Range is that we only need one literal notation, ie. x..y. Otherwise I think it would probably be better to have a separate Interval class, if only for the fact that Range, unlike an actual interval, can't exclude the starting sentinel. But other methods would likely vary too, as some of your comments make clear, such as #==.

In addition there are a lot of short-cuts taken when dealing with Ranges. If for instance we create an EvenInt class as subclass of Integer that could only represent even integers. Then...

```
r = (EvenInt[2]...EvenInt[4])
```

This range will not actually behave as we expect in all cases.

```
r.to_a            #=> [2,4]
[0,1,2,3,4][r]    #=> [2,3,4]
r.include?(3)     #=> true
```

#7 - 11/07/2011 01:40 AM - trans (Thomas Sawyer)

ADMIN! There is bad bug in the redmine interface that deletes the message if one tries to use the edit feature. I had to resubmit this post four times to get it show up again.

#8 - 11/11/2011 07:51 PM - alexeymuranov (Alexey Muranov)

I have just discovered that there is Range#cover? method which works how i would expect Range#include? to work. I would have preferred that these two behaved identically and that there were only one of them. Having the both looks to me like a way to cover up inconsistencies between different meaning and uses of Range.

I think that Range#to_a needs to accept an argument to tell it which elements of the range to use. The same when converting a range to an Enumerator with Range#each. Maybe the Enumerable module can provide some tools to create objects like "all_integers", "all_even_integers", "all_words_in_capital_letters_A_to_Z". Maybe even constants Enumerable::INTEGERS, Enumerable::EVEN_INTEGERS, Enumerable::WORDS_IN_CAPITAL_LETTERS_A_TO_Z would be good enough to begin with:

(1..6).to_a(Enumerable::INTEGERS) # => 1, 2, 3, 4, 5, 6.to_a(Enumerable::EVEN_INTEGERS) # => [2, 4, 6]

I understand better now the difficulties in defining a range as something other than a pair of bounds: whether ("X".."AB") is empty depends on the order (lex or deglex). I can think of one possible solution: allow to specify the order in cases where more than one exists. Probably the order should be some kind of an "Order" object, but in simple cases, to start with, it can be a Symbol. The use would be like this:

Range.new("A", "AB", exclusive=false, order=:lex).include?("X")   # => false
Range.new("A", "AB", exclusive=false, order=:deglex).include?("X") # => true

Range.new("A", "C", exclusive=false, order=:lex).to_a(Enumerable::WORDS_IN_CAPITAL_LETTERS_A_TO_Z)

=> InfniteArrayError: ["A", "AA", "AAA", ...]

Range.new("A", "C", exclusive=false, order=:deglex).to_a(Enumerable::WORDS_IN_CAPITAL_LETTERS_A_TO_Z)

=> ["A", "B", "C"]

These are very rough ideas.

Update. Despite all the difficulties in finding a good definition for the Range class, i would still prefer to have an unambiguous set-like (interval-like) Range objects, and to use 2-element arrays when i only need to store two "bounds", and convert them to ranges when necessary.

Update 2011-11-23. I've learned a bit more about Enumerator class, and it seems to me that the "all_integers", "all_even_integers",

"all_words_in_capital_letters_A_to_Z" objects that i mentioned can almost be implemented as enumerators. Except that not all of them have the first element, and they need the ability to start iteration from anywhere, and cannot have repetitions.

#9 - 03/18/2012 10:36 PM - alexeymuranov (Alexey Muranov)

More of my crazy ideas. Wouldn't it be nice to be able to write:

a, b = 0.5, 10.5
(a..b).each(Integer) { |n| puts n }

#10 - 03/27/2012 10:54 PM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

#11 - 11/20/2012 09:53 PM - mame (Yusuke Endoh)

- Target version set to 2.6

#12 - 11/28/2012 11:30 PM - alexeymuranov (Alexey Muranov)

=begin
I propose to disregard the second part of my proposal: about (({RelativeNumeric})) and
(({RelativeRange})). When i look back at it now, it looks quite crazy and not
particularly useful.

However, i still would have liked to see the behavior of (({Range})) changed, as in the
first part of the proposal. In other words, i would like (({Range})) to become a ((*lazy
ordered set*)), infinite or finite.

I think i proposed (({RelativeNumeric})) simply because i didn't dare to propose
to deprecate the usage like

b = a[1..-2]

for an array (({a})) without proposing some similarly looking replacement.
Now i've decided i would prefer this usage be just deprecated.

How about a new method (({Array#clip(fixnum, fixnum)})) to write

b = a.clip(1, 1)

instead of

b = a[1..-2]

? It can also have the (({#clip!})) version. (I can start a new Feature Request if a discussion is necessary.)

It looks strange to have to convert a pair of numbers ((*m*)) and ((*n*)) into a range (({m..(-1-n)})) to simply ask an array to remove ((*m*)) elements from the beginning and ((*n*)) elements from the end. If you think about what the range (({m..(-1-n)})) actually ((*is*)), then the syntax (({a[m..(-1-n)]})) looks plain weird :).

((*Update*)): i have just discovered that in ((*Rails ActiveSupport*)) there are methods (({Array#from})) and (({Array#to})), but unfortunately they do not accept negative indices (i'll think about a PR). With negative indices, another alternative to (({a[m..(-1-n)]})) would be (({a.from(m).to(-1-n)})).

I'll consider rewriting this proposal and posting as a new one, so that this one could be closed.

((*Edited 2012-12-10.*))
=end

#13 - 12/12/2012 01:26 AM - alexeymuranov (Alexey Muranov)

I propose to close this my Feature Request, as i have opened a new one that replaces it: #7545, see also #7546.

#14 - 12/12/2012 01:45 AM - drbrain (Eric Hodel)

- Status changed from Assigned to Rejected