Ruby trunk - Bug #15745

There is no symmetry in the beginless range and the endless range using `Range#inspect`

04/03/2019 09:37 AM - koic (Koichi ITO)

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	ruby 2.7.0dev (2019-04-03 trunk 67423) [x86_64-darwin17]	**Backport:**	2.4: UNKNOWN, 2.5: UNKNOWN, 2.6: UNKNOWN

Description

The following commit introduces beginless range.
https://github.com/ruby/ruby/commit/95f7992b89efd35de6b28ac095c4d3477019c583

```
% ruby -v
ruby 2.7.0dev (2019-04-03 trunk 67423) [x86_64-darwin17]
```

There is no symmetry with endless range when using Range#inspect method.

```
(1..).inspect # => "1.."
(..5).inspect # => "nil..5"
```

How about unifying whether it represents nil?

Related issues:

Related to Ruby trunk - Feature #14799: Startless range	**Closed**

History

#1 - 04/20/2019 11:37 AM - naruse (Yui NARUSE)

- Related to Feature #14799: Startless range added

#2 - 04/26/2019 08:04 AM - mame (Yusuke Endoh)

I have no strong opinion about this, but (nil..nil).inspect #=> ".." looks less reasonable because we cannot actually write the literal as is, so we need to decide the behavior of this corner case.

#3 - 04/27/2019 08:50 PM - Eregon (Benoit Daloze)

Maybe always showing nil is clearer, since anyway the current implementation of begin/end-less Ranges leaks that detail? (e.g., through #begin and #end)
Somewhat related about leaking Range's implementation details: #15804.

#4 - 04/27/2019 10:24 PM - mame (Yusuke Endoh)

Just confirm, Eregon (Benoit Daloze), do you mean endless range's inspect should also show nil?

```
p(1..)       #=> 1..nil
p(..1)       #=> nil..1
p(nil..nil) #=> nil..nil
```

#5 - 04/28/2019 11:42 AM - Eregon (Benoit Daloze)

Yes, I think it's a possibility and is rather consistent.
Otherwise, we need to special-case #inspect for beginless, endless and begin+endless ranges as you showed above.

I'd rather have a straightforward implementation for Range#inspect like
https://github.com/oracle/truffleruby/blob/04afba28d4ab1bdbfd8d92027f1ccc48b9a50bcb/src/main/ruby/core/range.rb#L212-L215

#6 - 04/28/2019 12:17 PM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote:

> Yes, I think it's a possibility and is rather consistent.

Agreed, it is the most consistent. IMHO, p (1..) #=> 1..nil is a bit verbose, though.

I think that an endless range will be much more commonly used than a beginless one because an endless range has many use cases but a beginless range has only one use case (DSL-like usage). Thus, I liked to make an endless one more useful and implemented the current behavior.

But I admit that the current behavior looks inconsistent. I have no strong opinion. I'll hear other committers' opinions at the next dev meeting.

#7 - 04/28/2019 09:03 PM - Eregon (Benoit Daloze)

mame (Yusuke Endoh) wrote:

> a beginless range has only one use case (DSL-like usage)

I'm not sure what you mean by DSL-like usage, but I wouldn't be surprised if many people use it like array[..-2] (instead of array[0..-2]).
In other words, I'd expect it's approximately as common as endless ranges for the purpose of indexing a sequence.

#8 - 04/29/2019 12:05 AM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote:

> I'm not sure what you mean by DSL-like usage, but I wouldn't be surprised if many people use it like array[..-2] (instead of array[0..-2]).
> In other words, I'd expect it's approximately as common as endless ranges for the purpose of indexing a sequence.

I meant "DSL-like", for example, debtors = Accounts.where(balance: ...0).

I strongly prefer ary[0..] to ary[0..-1] because the former has no magical -1. However, IMHO, there is no strong reason to prefer ary[..-2] to ary[0..-2] because 0 is not so magical in terms of indexing. It would be useful where non-zero-based indexing is used, but I don't think that it is common in Ruby.

In addition, an endless range has another big usage: (1..).each { ... }. On the other hand, a beginless range cannot iterate.

So, I believe that a beginless range has much less usages, and this is one of the reasons why a beginless range was not introduced with an endless one. That being said, I'm unsure if Range#inspect is mainly used for endless range.

#9 - 04/29/2019 10:34 AM - zverok (Victor Shepelev)

> I think that an endless range will be much more commonly used than a beginless one because an endless range has many use cases but a beginless range has only one use case (DSL-like usage). Thus, I liked to make an endless one more useful and implemented the current behavior.

That's very confusing.
The only thing where the beginless range is less justified than endless is **indexing of arrays**, which I believe is a very small share of Range usage.

Other are:

- case
- grep
- storing ranges in constants
- indeed DSLs (many kinds of them)
- clamp (if the ticket would be accepted, which I believe it should)

Generally, the beginless range is an exactly equally powerful and useful concept as an endless one, and the thing that "for array indexing, the beginless range is less useful" should be considered a style detail, not the reason for deliberately "less mindful" implementation.

#10 - 04/30/2019 04:02 PM - mame (Yusuke Endoh)

I think we digress. I had no intention to discuss a beginless range itself, but maybe it started digressing. My apologies. Let's focus on the original topic in this ticket.

Do you think which is the best?

(0) Keep the current behavior: p (1..) #=> (1..), p (..1) #=> (nil..1), and p (nil..nil) #=> (nil..).
(1) Make all cases explicit: p (1..) #=> (1..nil), p (..1) #=> (nil..1), and p (nil..nil) #=> (nil..nil).
(2) Omit all nils except (nil..nil): p (1..) #=> (1..), p (..1) #=> (..1), and p (nil..nil) #=> (nil..nil).

I have no strong opinion. (Honestly, I'm not so interested in the result of inspect in this case.) But if I have to choose, my current opinion is (0).

- (0) is somewhat reasonable for me.
- (1) looks a bit verbose to me (and brings tiny incompatibility).
- I don't like (2) because of the special handling.

Let me know your opinions. I'll bring them to the next dev meeting and ask matz decide.

#11 - 04/30/2019 04:41 PM - zverok (Victor Shepelev)

My reasoning is semi-open ranges are valuable feature, and by inspect we should suggest to users their "naturalness", not "it is just a quick hack, look".
Explicit rendering of nil is *underlining* some kind of "hackiness" ("it is just a syntactic sugar for 1..nil underneath").
I believe it is an important new language feature, and it should be represented adequately.
1..nil/nil..1 is not adequate, because it is hard to read: "range from 1 to nothing"? (it doesn't read as "range without end", which it is).

So I'd say that:

1. p (1..) # => (1..), and that's the only adequate representation
2. p (..1) # => (..1), and that's the only adequate representation
3. How nil..nil is represented is not THAT important, because it is the rarest and less useful case; I don't think even (..) is that bad. It looks kinda weird, but it still is what it is: "range (designated by ..) with neither end nor beginning".

The point is, there could be a discussion about (3), and it is only marginally important; but not about (1) and (2).
In this line of reasoning, "Make (..1) looking weird for consistency with (nil..nil)" is less than desirable.

#12 - 04/30/2019 05:06 PM - mame (Yusuke Endoh)

Oops, I'm sorry, parentheses are not rendered in the result of inspect. Let me rephrase and include (3):

(0) Keep the current behavior: p (1..) #=> 1.., p (..1) #=> nil..1, and p (nil..nil) #=> nil...
(1) Make all cases explicit: p (1..) #=> 1..nil, p (..1) #=> nil..1, and p (nil..nil) #=> nil..nil.
(2) Omit all nils except (nil..nil): p (1..) #=> 1.., p (..1) #=> ..1, and p (nil..nil) #=> nil..nil.
(3) Force to omit all nils: p (1..) #=> 1.., p (..1) #=> ..1, and p (nil..nil) #=> ...

I'd like to avoid "don't care". Even if we don't care, we need to choose one of them.

#13 - 04/30/2019 05:22 PM - zverok (Victor Shepelev)

> I'd like to avoid "don't care". Even if we don't care, we need to choose one of them.

Yes, but from my perspective, the whole choice (being it a further discussion, simple voting, Matz's decision) is only about "how the nil..nil is represented", while representation of 1.. and ..1 should be just this: 1.. and ..1.

TBH, for even for nil..nil it is hard to imagine reasons for representation other than .. (following the logic described above), only if just "aestetics"?..

#14 - 04/30/2019 09:20 PM - Eregon (Benoit Daloze)

I didn't realize, but Range#to_s and Range#inspect differ by calling #to_s and #inspect on both ends:

```
puts ('a'..'z').to_s
# => a..z
puts ('a'..'z').inspect
# => "a".."z"
```

Following that, for ("a"..) we'd get a.. and "a"..nil, which I agree is weird for an open end to change representation based on #to_s/#inspect.
So in the spirit of most #inspect methods printing what one would write for the Ruby literal I'd be inclined for (2), that is 1.., ..1 and nil..nil.