Ruby - Feature #18368

Range#step semantics for non-Numeric ranges

11/29/2021 03:30 PM - zverok (Victor Shepelev)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	

Description

I am sorry if the question had already been discussed, can't find the relevant topic.

"Intuitively", this looks (for me) like a meaningful statement:

```
(Time.parse('2021-12-01')..Time.parse('2021-12-24')).step(1.day).to_a
#                                                    ^^^^^ or just 24*60*60
```

Unfortunately, it doesn't work with "TypeError (can't iterate from Time)".
Initially it looked like a bug for me, but after digging a bit into code/docs, I understood that Range#step has an odd semantics of "advance the begin N times with #succ, and yield the result", with N being always integer:

```
('a'..'z').step(3).first(5)
# => ["a", "d", "g", "j", "m"]
```

The fact that semantic is "odd" is confirmed by the fact that for Float it is redefined to do what I "intuitively" expected:

```
(1.0..7.0).step(0.3).first(5)
# => [1.0, 1.3, 1.6, 1.9, 2.2]
```

(Like with Range#=== some time ago, I believe that to be a strong proof of the wrong generic semantics, if for numbers the semantics needed to be redefined completely.)

Another thing to note is that "skip N elements" seem to be rather "generically Enumerable-related" yet it isn't defined on Enumerable (because nobody needs this semantics, typically!)

Hence, two questions:

- Can we redefine generic Range#step to new semantics (of using begin + step iteratively)? It is hard to imagine the amount of actual usage of the old behavior (with String?.. to what end?) in the wild
- If the answer is "no", can we define a new method with new semantics, like, IDK, Range#over(span)?

UPD: More examples of useful behavior (it is NOT only about core Time class):

```
require 'active_support/all'

(1.minute..20.minutes).step(2.minutes).to_a
#=> [1 minute, 3 minutes, 5 minutes, 7 minutes, 9 minutes, 11 minutes, 13 minutes, 15 minutes, 17
minutes, 19 minutes]

require 'tod'

(Tod::TimeOfDay.parse("8am")..Tod::TimeOfDay.parse("10am")).step(30.minutes).to_a
#=> [#<Tod::TimeOfDay 08:00:00>, #<Tod::TimeOfDay 08:30:00>, #<Tod::TimeOfDay 09:00:00>, #<Tod::Ti
meOfDay 09:30:00>, #<Tod::TimeOfDay 10:00:00>]


require 'matrix'
(Vector[1, 2, 3]..).step(Vector[1, 1, 1]).take(3)
#=> [Vector[1, 2, 3], Vector[2, 3, 4], Vector[3, 4, 5]]

require 'unitwise'
(Unitwise(0, 'km')..Unitwise(1, 'km')).step(Unitwise(100, 'm')).map(&:to_s)
#=> ["0 km", "1/10 km", "1/5 km", "3/10 km", "2/5 km", "0.5 km", "3/5 km", "7/10 km", "4/5 km", "9
/10 km", "1 km"]
```

UPD: Responding to discussion points:

Q: Matz is concerned that the proposed simple definition will be confusing with the classes where + is redefined as concatenation.

A: I believe that simplicity of semantics and ease of explaining ("it just uses + underneath, whatever + does, will be performed") will make the confusion minimal.

Q: Why not introduce new API requirement (like "class of range's begin should implement increment method, and then it will be used in step)

A: require *every* gem author to change *every* of their objects' behavior. For that, they should be aware of the change, consider it important enough to care, clearly understand the necessary semantics of implementation, have a resource to release a new version... Then all users of all such gems would be required to upgrade. The feature would be DOA (dead-on-arrival).

The two alternative ways I am suggesting: change the behavior of #step or introduce a new method with desired behavior:

1. Easy to explain and announce
2. Require no other code changes to immediately become useful
3. With something like backports or ruby-next easy to start using even in older Ruby version, making the code more expressive even before it would be possible for some particular app/compny to upgrade to (say) 3.2

All examples of behavior from the code above are real irb output with monkey-patched Range#step, demonstrating how little change will be needed to code outside of the Range.

Associated revisions

Revision d450f9d6a28f01b7ca6030a925921dbf35cee439 - 08/18/2024 10:15 AM - zverok (Victor Shepelev)

Make Range#step to consistently use + for iteration (#7444)

Make Range#step to consistently use + for iteration [Feature #18368]

Previously, non-numerics expected step to be integer,
and iterated with begin#succ, skipping over step value
steps. Since this commit, numeric and non-numeric iteration
behaves the same way, by using + operator.

Revision d450f9d6a28f01b7ca6030a925921dbf35cee439 - 08/18/2024 10:15 AM - zverok (Victor Shepelev)

Make Range#step to consistently use + for iteration (#7444)

Make Range#step to consistently use + for iteration [Feature #18368]

Previously, non-numerics expected step to be integer,
and iterated with begin#succ, skipping over step value
steps. Since this commit, numeric and non-numeric iteration
behaves the same way, by using + operator.

Revision d450f9d6 - 08/18/2024 10:15 AM - zverok (Victor Shepelev)

Make Range#step to consistently use + for iteration (#7444)

Make Range#step to consistently use + for iteration [Feature #18368]

Previously, non-numerics expected step to be integer,
and iterated with begin#succ, skipping over step value
steps. Since this commit, numeric and non-numeric iteration
behaves the same way, by using + operator.

History

#1 - 01/14/2022 02:58 AM - mame (Yusuke Endoh)

This topic was discussed at the dev-meeting yesterday.

A naive implementation (using begin + step iteratively) will allow the following behavior.

```
([]..).step([1]).take(3)         #=> [[], [1], [1, 1]]
(Set[1]..).step(Set[2]).take(3) #=> [Set[1], Set[1, 2], Set[1,2]]
```

@matz (Yukihiro Matsumoto) was okay to allow (timestamp1...timestamp2).step(3.hours), but wanted to prohibit the above behavior. We need to find a reasonable semantics to allow timestamp ranges and to deny container ranges.

#2 - 01/14/2022 11:41 AM - zverok (Victor Shepelev)

@mame (Yusuke Endoh) @matz (Yukihiro Matsumoto)
I believe that "step implemented with +" is clear and useful semantics which might help with much more than time calculations:

```
require 'numo/narray'

p (Numo::NArray[1, 2]..).step(Numo::NArray[0.1, 0.1]).take(5)
# [Numo::Int32#shape=[2] [1, 2],
#  Numo::DFloat#shape=[2] [1.1, 2.1],
#  Numo::DFloat#shape=[2] [1.2, 2.2],
#  Numo::DFloat#shape=[2] [1.3, 2.3],
#  Numo::DFloat#shape=[2] [1.4, 2.4]]
```

What's unfortunate in @mame's example is rather that we traditionally reuse + in collections for concatenation (it isn't even commutative!), but that's just how things are.

While stepping with array concatenation might be considered weird, I don't think it would lead to any real bugs/weird code; and it is easy to explain by "it is just what + does".
We actually have this in different places too, like, this work (with semantics not really clear):

```
([1]..[3]).cover?([1.5]) # => true
```

#3 - 01/14/2022 12:58 PM - Eregon (Benoit Daloze)

One way to achieve the same result currently is Enumerator.produce:

```
require 'time'
Enumerator.produce(Time.parse('2021-12-01')) { _1 + 24*60*60 }.take_while { _1 <= Time.parse('2021-12-24') }
```

Somewhat related to https://bugs.ruby-lang.org/issues/18136#note-15 (where <= can't be used).

But I think step should just use + and < (for exclude_end?)/<=, I don't see any reason to prevent the above cases, ([]..).step([1]).take(3) can actually be useful.

#4 - 01/30/2022 04:15 AM - Dan0042 (Daniel DeLorme)

> matz: I'd like to allow numeric-type '+', but to deny concatination-type '+'
> matz: we should not modify the behavior when the receiver is a String
> matz: I'm okay to allow (timestamp1...timestamp2).step(3.hours)
> matz: but I'd like to prohibit ([]..).step([1]).take(3)

I fully agree with the above. Here's one idea: how about adding an #increment(n) method. For Numeric, Float, Time, Date, it would be an alias to +. For String it would be equivalent to doing succ n times. So Range#step would have the semantics of using begin.increment(step) iteratively.

#5 - 01/30/2022 09:58 AM - zverok (Victor Shepelev)

> Here's one idea: how about adding an #increment(n) method. For Numeric, Float, Time, Date, it would be an alias to +. For String it would be equivalent to doing succ n times. So Range#step would have the semantics of using begin.increment(step) iteratively.

What about other, non-core classes? Will we have implicitly defined Object#increment to handle them, and if so, how it is defined? Or each and every library should define their own #increment, and if it doesn't, the range iteration just fails?

In general, I believe this approach would render the feature virtually unusable.

On the other hand, "It just uses +" follows the principle of least surprise, and is easy to explain and document. It is also totally easy to grasp why in cases where + is defined as concatenation, the result is the way it is (the same way that result of ('A'..'aa').to_a might be "somewhat surprising" at first, but is easy to explain in hindsight).

In general, what I am trying to do here is to make the semantics more intuitive, not more complicated to tailor some special case.

PS: I might even suppose a reasonable use of the new proposed step with strings. For example, in Markdown markup language, the level of the header is designated by the number of # before it. So, one might do something like

```
PREFIXES = (''..).step('#').take(7)
# => ["", "#", "##", "###", "####", "#####", "######"]

# ...or even just
PREFIXES = (''..'######').step('#').to_a
# => ["", "#", "##", "###", "####", "#####", "######"]
```

```
# and then use it like
def render_header(h)
  "#{PREFIXES[h.level]} #{h.text}"
end
```

Would this be an unreadable crime against common sense? I'd say not. YMMV.
(I am not saying you can't do '#' * h.level or that it is worse. I am just saying that nothing "weird" seems to be in the semantics of Range#step even in the String case.)

#6 - 01/30/2022 02:12 PM - zverok (Victor Shepelev)

Some clarifications after rereading the corresponding dev.meeting log:

My proposal is not about Time, but about generic behavior.

Besides Time, realistic, existing, types to handle are at least:

- Date and DateTime
- and any other date-alike/time-alike objects of third-party gems, say, "time-of-day" object (gem tod):

```
require 'tod'
require 'active_support/all'
(Tod::TimeOfDay.parse("8am")..Tod::TimeOfDay.parse("10am")).step(30.minutes).to_a
#=> [#<Tod::TimeOfDay 08:00:00>, #<Tod::TimeOfDay 08:30:00>, #<Tod::TimeOfDay 09:00:00>, #<Tod::TimeOfDay 09:3
0:00>, #<Tod::TimeOfDay 10:00:00>]
```

- ...or ActiveSupport::Duration itself:

```
(1.minute..20.minutes).step(2.minutes).to_a
#=> [1 minute, 3 minutes, 5 minutes, 7 minutes, 9 minutes, 11 minutes, 13 minutes, 15 minutes, 17 minutes, 19
minutes]
```

- Matematical vectors and matrices:

```
require 'matrix'
(Vector[1, 2, 3]..).step(Vector[1, 1, 1]).take(3)
#=> [Vector[1, 2, 3], Vector[2, 3, 4], Vector[3, 4, 5]]
```

- Quantities with measurement units:

```
require 'unitwise'
(Unitwise(0, 'km')..Unitwise(1, 'km')).step(Unitwise(100, 'm')).map(&:to_s)
#=> ["0 km", "1/10 km", "1/5 km", "3/10 km", "2/5 km", "0.5 km", "3/5 km", "7/10 km", "4/5 km", "9/10 km", "1
km"]
```

- ...any other custom type with meaningful semantic of addition

I believe that simple and explicable semantics of reusing + is enough. It creates a good quick intuition of "what would happen", which requires no exceptions and clarifications.

We already following similar approach in different places: for example, ('2.7'..'3.1') === '3.0.1' "just works" without any additional nuances, even if ('2.7'..'3.1').to_a wouldn't produce 3.0.1 as one of the "range contained elements".

For another close example, things like ('5'..'a').to_a "just work", even if rarely semantically sound, because they follow a simple rule of "just uses #succ, however it is defined".

Finally, as stated above, **I don't think that *unexpected yet useful* results of simple intuitions are bad**—vice versa, it is funny and enlightening that this "just works as expected":

```
(''..'######').step('#').to_a
# => ["", "#", "##", "###", "####", "#####", "######"]
```

#7 - 01/30/2022 09:43 PM - Dan0042 (Daniel DeLorme)

@zverok (Victor Shepelev) if understand correctly, the implication is that range.step(1) (using +) would have different semantics than range.each (using succ); I have reservations about that.

Also, due to backward compatibility I don't think it's possible to change the behavior of ("a.."z").step(3) so the simple rule of "it just uses +" would suffer from at least one special case.

zverok (Victor Shepelev) wrote in #note-5:

> What about other, non-core classes? Will we have implicitly defined Object#increment to handle them, and if so, how it is defined? Or each and every library should define their own #increment, and if it doesn't, the range iteration just fails?

The idea is that #increment is used for addition but not concatenation. Nothing implicit. If a class has #increment defined that would be used for #step, otherwise it would fall back to using #succ, otherwise it would fail with "can't iterate" just like it does currently. Well, it's just one idea. From the dev meeting notes I also like nobu's idea of just delegating to begin_object#upto.

#8 - 01/30/2022 10:13 PM - zverok (Victor Shepelev)

> the implication is that range.step(1) (using +) would have different semantics than range.each (using succ); I have reservations about that.

Well, it is already so to some extent. Say, with numeric ranges #step returns ArithmeticSequence and not just Enumerator; and while the difference is subtle, it is there.

> Also, due to backward compatibility I don't think it's possible to change the behavior of ("a".."z").step(3) so the simple rule of "it just uses +" would suffer from at least one special case.

1. I am not sure about that, actually—how much of the code might use this? (I think there was a way to estimate with gemsearch?..) It is hard for me to imagine the reasonable use case, but I might be wrong.
2. Wouldn't maybe just a clear error message be enough to promptly port all the code affected? It is not the case where something will change semantics silently, it would be a clear and easy to understand exception
3. Worst case, there might be made a special case *only* for String to preserve old semantics. There were precedents in the past: when Range#=== was changed to use #cover?, the String ranges preserved old behavior... which turned out to be unnecessary and fixed in the next version

> The idea is that #increment is used for addition but not concatenation. Nothing implicit. If a class has #increment defined that would be used for #step, otherwise it would fall back to using #succ, otherwise it would fail with "can't iterate" just like it does currently. Well, it's just one idea. From the dev meeting notes I also like nobu's idea of just delegating to begin_object#upto.

Both #upto and #increment require *every* gem author to change *every* of their objects' behavior. For that, they should be aware of the change, consider it important enough to care, clearly understand the necessary semantics of implementation, have a resource to release a new version... Then all users of all such gems would be required to upgrade. The feature would be DOA (dead-on-arrival).

The two alternative ways I am suggesting: change the behavior of #step or introduce a new method with desired behavior:

1. Easy to explain and announce
2. Require no other code changes to immediately become useful
3. With something like backports or ruby-next easy to start using even in older Ruby version, making the code more expressive even before it would be possible for some particular app/compny to upgrade to (say) 3.2

NB: All examples of behavior from my comments are real irb output with monkey-patched Range#step, demonstrating how little change will be needed to code outside o the Range.

#9 - 02/02/2022 03:42 PM - Dan0042 (Daniel DeLorme)

"Dead-on-arrival" hyberbole aside, it does seem that using + semantics would allow Range#step to work "for free" with many existing classes.

More importantly, I realized there is no need to introduce any backward incompatibility: if begin_object is a string and step is integer, the legacy behavior can be kept instead of raising an exception. So ("a.."z").step(3) and ("..'######').step('#') are not mutually exclusive.

#10 - 08/06/2022 12:37 PM - zverok (Victor Shepelev)

- Description updated

#11 - 08/16/2022 02:45 PM - mame (Yusuke Endoh)

Let me summarize the problem and possible solutions.

Desirable behavior:

```
(timestamp1...timestamp2).step(3.hours) { ... }
(date1..date2).step(1.day) { ... }
```

Undesirable behavior:

```
([]..).step([1]).take(3)         #=> [[], [1], [1, 1]]
(Set[1]..).step(Set[2]).take(3) #=> [Set[1], Set[1, 2], Set[1,2]]
```

Solution 1: Give up the desirable behavior and keep the current status

- Pros and cons are trivial

Solution 2: Change Range#step to "+" semantics and accept the undesirable behavior

- (a..b).step(d) {...} while a <= b; ...; a += d; end
- Pros and cons are trivial

Solution 3: Introduce a new step-like method as "+" semantics

- (a..b).over(d) {...} while a <= b; ...; a += d; end
- Pros: Range#step does not change (neither desirable or undesirable behaviors are introduced) / we can use Ragne#over for 3rd-party classes implementing + without modification
- Cons: the undesirable behavior is introduced to Range#over

Solution 4: Make Range#step delegate to begin#step or #upto for unknown types

- (a..b).step(d) {...} a.step(b, d) {...}
- (a...b).step(d) {...} a.step(b, d, exclude_end: true) {...}
- or
- (a..b).upto(d) {...} a.upto(b, false, by: d) {...}
- (a...b).upto(d) {...} a.upto(b, true, by: d) {...}
- Pros: the desirable behavior is introduced without the undesirable behavior
- Cons: 3rd-party classes (and some builtin classes like Time) have to implement #step or #upto protocol

Solution 5: Make Range#step use range.begin#increment for unknown types

- (a..b).step(d) {...} while a <= b; ...; a = a.increment(d); end
- Pros: the desirable behavior is introduced without the undesirable behavior
- Cons: builtin and 3rd-party classes have to implement the new #increment protocol

#12 - 08/16/2022 03:03 PM - Eregon (Benoit Daloze)

IMHO 2) is the best. I think trying to prevent the "undesirable behavior" is too artificial.
Also that behavior seems intuitive and simple when knowing + is used.
Someone might even use that array example in practice (e.g., for some initialization code or initial value of a cache).

#13 - 08/16/2022 03:06 PM - zverok (Victor Shepelev)

Thanks, @mame (Yusuke Endoh), for a summary.

I'd like to emphasize, though, that the "undesirability" of undesirable behavior looks very mild to me:

1. I don't see how something like this can happen accidentally (save for some very weird code that does range_passed_by_user.step(value_passed_by_user) and deliberately **relies** on the error being raised for values with the semantics of + being other than addition)
2. When somebody consciously tries to use #step (or #over) with arrays, I don't see how the results would be confusing: "it just uses +" is really easy to explain and understand
3. It is rather a thing that can bring accidental joy by its simple consistency, as shown in https://bugs.ruby-lang.org/issues/18368#note-5

#14 - 08/16/2022 05:34 PM - Dan0042 (Daniel DeLorme)

I also think 2) is the best
while reiterating that we can and should keep the behavior of ("a".."z").step(num) for legacy purposes.
Pros: Range#step behavior is enriched while preserving full backward compatibility

#15 - 02/09/2023 05:07 AM - matz (Yukihiro Matsumoto)

I accept solution 2, which someone may specify an array or a set for the range edge, and get an unnatural result. I consider it is the responsibility of the user.

Matz.

#16 - 02/09/2023 09:32 AM - zverok (Victor Shepelev)

Thanks @matz (Yukihiro Matsumoto) I'll be working on the implementation.

#17 - 03/04/2023 10:49 AM - zverok (Victor Shepelev)

The PR is here: https://github.com/ruby/ruby/pull/7444

Clarification of semantics led to a few minor changes of behavior for numeric steps, too:

- Consistent support for negative step:

```
p (1..-10).step(-3).to_a
#=> [1, -2, -5, -8] -- ArithmeticSequence backward iteration, on Ruby 3.2 and master
(1..-10).step(-3) { p _1 }
# Ruby 3.2: step can't be negative (ArgumentError) -- inconsistent with ArithmeticSequence behavior
# master: prints 1, -2, -5, -8, consistent with ArithmeticSequence
```

- Less greedy float conversion:

```
require 'active_support/all'

p (1.0..).step(2.minutes).take(3)
# 3.2: [1.0, 121.0, 241.0] -- forces any passed value to be float if it has #to_f
# master: [1.0, 2 minutes and 1.0 second, 4 minutes and 1.0 second] -- properly uses step#coerce to find a
 suitable type
```

- Drop support for generic #to_int. Before, it was considered that integer is (almost) always the intended step value, so the step tried to be converted to #to_int if it wasn't numeric:

```
o = Object.new
def o.to_int
  2
end

p (1..6).step(o).to_a
#=> [1, 3, 5] on Ruby 3.2

# Now, no assumptions on the step are made other than it should be `+`-able to `begin`:
p (1..6).step(o).to_a
# master: `+': Object can't be coerced into Integer

# But:
def o.coerce(other)
  [other, 2]
end

p (1..6).step(o).to_a
#=> [1, 3, 5] on master
```

I am open to discussing those changes, but to the best of my understanding, neither of them should be severely breaking, and they are naturally following the change of the semantics.

#18 - 03/24/2023 10:33 AM - zverok (Victor Shepelev)

Can please somebody review the PR?.. I have joined Ukrainian army but still have a bit of time to fix notes and prepare it to merge.

In a few weeks it can become impossible.

#19 - 03/24/2023 02:57 PM - Dan0042 (Daniel DeLorme)

I notice this breaks compatibility for ('a'..'e').step(2).to_a

Why? Why break backward compatibility so pointlessly, when you don't even need to?

#20 - 04/02/2023 01:46 PM - zverok (Victor Shepelev)

@Dan0042 Can you please elaborate your question (especially considering its extremely strong wording)?

This ticket is about changing the semantics of step to use + instead of succ, and Matz agreed to give it a try.

How exactly do you imagine implementing the change without "pointlessly" breaking compatibility?

Thanks in advance.

#21 - 04/03/2023 06:23 PM - Dan0042 (Daniel DeLorme)

Ah yes, sorry for that. Whenever I write and review something before posting, it seems fine. And whenever I re-read it a week later it feels too strong. Maybe it's a curse. Please accept my advance apologies if I'm doing the same thing in this post.

As I tried saying in #note-9 and #note-14, it's easy to keep a special case for a string range with a numeric step. I noticed you removed the special case for Symbol ranges (and I was quite surprised it existed in the first place). But it would have been just as easy to keep the special case in there, and add the same thing for Strings, e.g.

```
    else if (STRING_P(b) && (NIL_P(e) || STRING_P(e)) && step_num_p) {
/* strings + numeric step are special */
        if (NIL_P(e)) {
            rb_str_upto_endless_each(b, step_i, (VALUE)iter);
        }
        else {
            rb_str_upto_each(b, e, EXCL(range), step_i, (VALUE)iter);
        }
```

```
    }
```

By "pointless" I meant that breaking compatibility here is not needed. We can have the improved behavior without breaking compatibility. Breaking this particular case has no benefit; something that previously worked now raises an error, and may break someone's code. Now, if str + int had a defined behavior I would agree that breaking compatibility serves a purpose. But since that's not the case, we're just going from "it works" to "it raises" and that serves no purpose, hence why I said "pointless". Again sorry for the strong wording.

I don't mean to kick up a fuss about your work. I really appreciate the effort you've made in making ranges more generally useful in ruby. I just find it disappointing when compatibility is sacrificed when not strictly needed.

#22 - 04/05/2023 12:54 PM - zverok (Victor Shepelev)

@Dan0042 I don't think that "maintaining compatibility" here is valuable unless proven otherwise (e.g. it is a very common idiom for iterating through strings - while working on this ticket, I found no evidence for that).

My reasoning is that it is very confusing when some simple and "math-alike" functionality is specialized for exactly one type. What if some user's type has both T+T and T#succ, should it also maintain "two behaviours depending on argument type"? If not, why?

Breaking this particular case has a benefit of clear semantics, explained by one phrase, which never breaks user's mental model with "well, the method does this, but with String boundaries and Int argument it also behaves like that."

If we are bold enough to change generic method's semantics (I believe Matz is open to it), maintaining legacy behaviours totally unrelated to the new semantics is a forever burden on the language.

#23 - 04/05/2023 01:48 PM - Dan0042 (Daniel DeLorme)

I guess we just have different values regarding backward compatibility. Your arguments are not false, but to me they have a value several orders of magnitude below compatibility. It doesn't need to be a "very common" idiom, it just needs to be used by a few people, who discover their code breaks upon upgrade, and after a few times of getting angry due to breakage decide to throw the towel and never again choose ruby for a new project.

I'm not sure if @matz (Yukihiro Matsumoto) meant "yes, it's ok to break compatibility" or "yes, it's ok since it doesn't break compatibility".

#24 - 04/05/2023 02:16 PM - zverok (Victor Shepelev)

@Dan0042 I value backwards compatibility a lot (I mentioned it in original ticket).

I though believe that in *this* particular case the old behaviour for non-numeric ranges is so weird that

1. Very small amount of code, if any, would be broken
2. The error would be very clear
3. Preserving both behaviours just for strings would be very hard to explain in hindsight and will affect language's quality and learnability.

#25 - 04/05/2023 02:32 PM - Dan0042 (Daniel DeLorme)

Ah, from https://github.com/ruby/dev-meeting-log/blob/master/2022/DevMeeting-2022-01-13.md

> matz: we should not modify the behavior when the receiver is a String

I take that to mean Matz prefers not breaking compatibility.

#26 - 05/09/2023 07:07 PM - janosch-x (Janosch Müller)

This is a cool improvement! I think it's fine to keep the special String behavior, and maybe that of Symbols. These special cases are not counterintuitive as there is no naturally intuitive way for them to behave. The burden on the language also looks manageable as it seems unlikely that a lot of complexity will be built on top of this part in particular.

#27 - 06/09/2023 07:48 AM - mame (Yusuke Endoh)

The three clarifications described in #note-17 were discussed at the dev meeting.

@matz (Yukihiro Matsumoto) said he wanted to make sure if the following pseudo code meets @zverok's expectation.

```
class Range
  def step(n)
    # TODO: we need to care "exclude end"
    # TODO: we should actually use <=> for comparison instead of >= or <=
    a, b = self.begin, self.end
    case n <=> 0
    when -1
      while a >= b
        yield a
        a += n
      end
    when 0
```

```
      raise "step can't be 0"
    when 1
      while a <= b
        yield a
        a += n
      end
    end
  end
end
```

If it does, @zverok's three clarifications are all approved. However, it was noted that the following case does not work well with @zverok's patch.

```
(1..-1).step(-1.seconds) { p _1 }
# expected: 1, 0 seconds, -1 seconds
# actual: no output
```

Is it possible to fix this issue?

@matz (Yukihiro Matsumoto) approved the third clarification "Drop support for generic #to_int.", but he also asked if we could keep the behavior by respecting #to_int on the #coerce side. Welcome if anyone can consider this.

#28 - 11/26/2023 11:15 AM - zverok (Victor Shepelev)

I am sorry that only now I had time to further work on the feature.

I understand it is almost Decemeber and the feature might not make it in the 3.3 (though I would be happy if it would).

The generic backward iteration was implemented:

```
(Time.utc(2022, 3, 1)..Time.utc(2022, 2, 24)).step(-24*60*60) { puts _1 }
# Prints:
#   2022-03-01 00:00:00 UTC
#   2022-02-28 00:00:00 UTC
#   2022-02-27 00:00:00 UTC
#   2022-02-26 00:00:00 UTC
#   2022-02-25 00:00:00 UTC
#   2022-02-24 00:00:00 UTC
```

The coercion for numeric values and custom objects works as expected:

```
val = Struct.new(:val) do
  def coerce(num) = [num, val]
end

p (1..3).step(val.new(1)).to_a
# => [1, 2, 3]
```

So I believe the feature is ready for the final review/merge.

Two nuances I am currently aware of:

1. I didn't change the behavior of numeric iteration, but it might be considered inconsistent:

```
(1r..).step(1).take(3)
#=> [(1/1), 2.0, 3.0]
(1r..).step(1.0).take(3)
#=> [1.0, 2.0, 3.0]
```

One might expect that those examples would return [1r, 2r, 3r], but **that's how it always worked**.

2. The rbs tests are broken

That's not because the RBS itself is broken by the change, but because one of the tests uses string range with the default and numeric steps, which are now incorrect. I don't think it represents some realistic use case, and RBS tests should be fixed.

I will be grateful for the instruction how to do that (I mean, what's the process to adjust bundled gem's tests that became irrelevant for the new Ruby version).

#29 - 12/07/2023 09:14 AM - zverok (Victor Shepelev)

@naruse (Yui NARUSE) Is there a chance for this change to be included in 3.3?

#30 - 08/18/2024 10:15 AM - zverok (Victor Shepelev)

- Status changed from Open to Closed

Applied in changeset git|d450f9d6a28f01b7ca6030a925921dbf35cee439.

Make Range#step to consistently use + for iteration (#7444)

Make Range#step to consistently use + for iteration [Feature #18368]

Previously, non-numerics expected step to be integer,
and iterated with begin#succ, skipping over step value
steps. Since this commit, numeric and non-numeric iteration
behaves the same way, by using + operator.

#31 - 08/22/2024 01:29 AM - matz (Yukihiro Matsumoto)

- Status changed from Closed to Open

It seems this change breaks step over string ranges (e.g. "a".."z").step(3)). We need to handle string ranges specifically.

Matz.

#32 - 08/22/2024 02:44 AM - knu (Akinori MUSHA)

We have little choice but to specialize it for strings as we don't want to add support for "a" + 3 that will most certainly bring disaster. □□

#33 - 08/22/2024 03:11 AM - knu (Akinori MUSHA)

If we take this compatibility as important and go with it, should we give up ("a".."aaaa").step("a") or support it?

#34 - 08/22/2024 06:44 AM - zverok (Victor Shepelev)

@matz (Yukihiro Matsumoto) I can certainly implement the specialization, but just to clarify: are there any evidence that people use ("a".."z").step(3) in any meaningful way? (Unfortunately, I have no way of doing the investigation myself, but probably those with access to “search through all existing gems code” server might shed some light on it?)

What I am concerned about is that having it specialized just for strings has an intuition-breaking consequences and would certainly be perceived like a “bug” somewhere down the road. We had the same with switching from include? to cover? as a Range#=== implementation:

- In 2.6, strings were excluded from the change
- ...but after several reports, it turned out that it is better to make string ranges behave consistently with everything else, which happened in 2.7.

In general, Range already have a history of such specializations (like specifically considering Time “linear” value in 2.3 to fix case Time.now ... behavior, before generic switching to cover? fixed it consistenlty), and I believe it is mostly confusing to the language’s users: some core class behaving a “special” way that can’t be imitated by any other class, breaking all and every “duck typing” intuition.

So, a big question for me is if the specialization is justified by some existing use cases, or by a generic caution against breaking things (and in the latter case, who guarantees that in some codebase nobody have relied on some *custom* object range and integer step behavior?.. IDK)

So, I see these options here:

1. Leave the “consistent first” option: no specialization for String (unless there is a strong proof that some popular gem or a widespread idiom is hurt by this);
2. Specialize it for strings to only work with numeric steps, and document that “strings are special, deal with it”
3. Specialize it for strings to work with both numeric *and* string steps (by explicit type-checking “if the range is string & the step is integer”)—that’s what @knu (Akinori MUSHA) asks about in #18368#note-33. Maybe even avoid documenting the special behavior, or document it is discouraged (so the old code—if it exists—wouldn’t break, but the new code wouldn’t rely on it)
4. Maybe make it always work with integer steps as a fallback (though there is no way to reliably check “whether this particular class supports obj+int directly, or we need to switch back to the ‘number of steps’ behavior”)
5. Rethink the decision in general (again, if there is a firm ground for concerns over the compatibility), and introduce *another* method that will behave like the currently implemented #step relying on #+, reverting #step to the old behavior.

Of those, I would honestly prefer to avoid (2) (very confusing for new users, teaching, and general language’s image) and (5); I don’t think (4) is generally possible; which leaves us with (1) (how is it now) and (3) (additional specialization for strings which *also* supports numeric steps on the basis of range begin and step class, in addition to supporting string steps).

TBH, I still believe there would be very low amount of incompatibility in existing code, but I might genuinely miss some common knowledge.

#35 - 08/22/2024 01:02 PM - mame (Yusuke Endoh)

@zverok (Victor Shepelev) Please fix the incompatibility first before having such a discussion. If it takes long, I will have to revert the change.

#36 - 08/22/2024 01:04 PM - zverok (Victor Shepelev)

@mame (Yusuke Endoh) It doesn’t take long, but I’ll have time for that on the weekend (most probably Sunday).

I hope that some clarifications might happen in a few days before that.

#37 - 08/22/2024 08:43 PM - Dan0042 (Daniel DeLorme)

matz (Yukihiro Matsumoto) wrote in #note-31:

> It seems this change breaks step over string ranges (e.g. "a".."z").step(3)). We need to handle string ranges specifically.

Thank you Matz, I am truly overjoyed the one in charge still cares about backward compatibility.

knu (Akinori MUSHA) wrote in #note-33:

> If we take this compatibility as important and go with it, should we give up ("a".."aaaa").step("a") or support it?

I'd say definitely support it. That way Range#step behavior remains largely consistent, with just one little special extra for compat.

BTW not sure if it's important but (:a .. :z).step(3) is also supported.

#38 - 08/25/2024 06:14 PM - zverok (Victor Shepelev)

Here we go: https://github.com/ruby/ruby/pull/11454

In the absence of further discussion, I went with option (3): String ranges support both “old” and “new” behavior.

#39 - 09/10/2024 01:22 PM - knu (Akinori MUSHA)

The compatibility with (:a .. :z).step(3) has also been restored.

https://github.com/ruby/ruby/pull/11573

#40 - 09/30/2024 05:00 AM - mame (Yusuke Endoh)

- Status changed from Open to Closed

I think there is no task in this ticket. Thank you @zverok (Victor Shepelev)