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:

```ruby
(Time.parse('2021-12-01')..Time.parse('2021-12-24')).step(1.day).to_a
```

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:

```ruby
{'a'..'z'}.step(3).first(5)
```

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

```ruby
(1.0..7.0).step(0.3).first(5)
```

(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)?

**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.

```ruby
{{[1]}}.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 Endo)**

**matz (Yukihiro Matsumoto)**

I believe that "step implemented with +" is clear and useful semantics which might help with much more than time calculations:

```ruby
require 'numo/narray'
```

```ruby
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],
```

03/07/2022
What's unfortunate in mame (Yusuke Endoh)'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):

```ruby
{{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:

```ruby
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 concatenation-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 markdown language, the level of the header is designated by the number of # before it. So, one might do something like

```ruby
PREFIXES = (''.step('#').take(7)
  # => ['", "#", "##", "###", "####", "#####", "#######"]
#
# ...or even just
PREFIXES = (''.step('#').to_a
  # => ['", "#", "##", "###", "####", "#####", "#######"]
#
# 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):

```ruby
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 09:30:00>, #<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:

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

- Quantities with measurement units:

```ruby
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":

```ruby
"....#####".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/compy 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" hyperbole 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(1)` are not mutually exclusive.