Ruby - Feature #14423

Enumerator from single object

01/30/2018 09:47 AM - zverok (Victor Shepelev)

Status:	Closed	
Priority:	Normal	
Assignee:		
Target version:		

Description

UPD: Current proposal

Introduce method Object#enumerate for producing infinite enumerator by applying block to result of previous call.

Reference implementation:

```
class Object
  def enumerate(&block)
    Enumerator.new { |y|
      val = self
      y << val
      loop do
        val = block.call(val)
        y << val
      end
    }
  end
end
```

Possible usages:

```
# Most idiomatic "infinite sequence" possible:
p 1.enumerate(&:succ).take(5)
# => [1, 2, 3, 4, 5]

# Easy Fibonacci
p [0, 1].enumerate { |f0, f1| [f1, f0 + f1] }.take(10).map(&:first)
#=> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

# Enumerable pagination
page.enumerate { |page| Faraday.get(page.next) if page.next }.take_while { |p| !p.nil? }
```

Reference to similar things:

- Clojure iterate "Returns a lazy sequence of x, (f x), (f (f x)) etc." No converging, just infinite sequence... And maybe that is even more basic and idiomatic. The name is nice, too.
- WolframLang FixedPoint
- Ramda converge
- Elixir Stream#unfold (ends iteration when nil is returned)
- Scala Iterator#iterate (just infinite sequence)

Initial proposal

Sometimes (or rather often), there is a programming pattern of "start from one object, do something, look at the result, do the same, look at the result (and so on)".

Examples:

- fetch page by URL, if pagination present, fetch next page;
- take 10 jobs from the queue, process them, exit when queue is empty;

Typically, those are represented by while or loop + break which somehow feels "not functional enough", and even "not Ruby

enough", so much less expressive than map and other Enumerable/Enumerator-based cycles.

In some functional languages or libraries, there is function named FixedPoint or converge, whose meaning is "take an initial value, repeat the block provided on the result on prev computation, till it will not 'stable'". I believe this notion can be useful for Rubyists too.

Reference implementation (name is disputable!):

```
class Object
  def converge(&block)
    Enumerator.new { |y|
      prev = self
      y << self
      loop do
        cur = block.call(prev)
        raise StopIteration if cur == prev
        y << cur
        prev = cur
      end
    }
  end
end
```

Examples of usage:

```
# Functional kata: find the closest number to sqrt(2):
1.0.converge { |x| (x + 2 / x) / 2 }.to_a.last # => 1.414213562373095
Math.sqrt(2) # => 1.4142135623730951

# Next page situation:
get(url).converge { |page| page.next }
# => returns [page, page.next, page.next.next, ...til the result is nil, or same page repeated]

# Job queue situation:
queue.top(10).converge { |jobs|
  jobs.each(&:perform)
  queue.top(10)
}
# => takes top 10 jobs, till queue is empty (`[]` is returned two successful times)

# Useful for non-converging situations, too:
2.converge { |x| x ** 2 }.take(4)
# => [2, 4, 16, 256]

# Idiomatic Fibonacci:
[0, 1].converge { |f0, f1| [f1, f0 + f1] }.take(10).map(&:first)
# => [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Reference to similar things:

- Clojure iterate "Returns a lazy sequence of x, (f x), (f (f x)) etc." No converging, just infinite sequence... And maybe that is even more basic and idiomatic. The name is nice, too.
- WolframLang FixedPoint
- Ramda converge
- Elixir Stream#unfold (ends iteration when nil is returned)
- Scala Iterator#iterate (just infinite sequence)

Possible call-seq:

- If converges: Object#converge(&block), Enumerator.converge(object, &block);
- If just an infinite sequence: Object#iterate(&block), Object#deduce(&block) (as opposed to reduce), Enumerator.iterate(object, &block), Enumerator#enumerate(object, &block).

WDYT?..

PS: Can imagine somebody already proposed that, yet can't find nothing similar in the tracker for all keywords I've tried.

Related issues:

History

#1 - 01/30/2018 09:47 AM - zverok (Victor Shepelev)

- Description updated

#2 - 01/30/2018 02:35 PM - sos4nt (Stefan Schüßler)

> exit when queue is empty [...] til the result is nil, or same page repeated

All those conditions are quite different. Your reference implementation only handles the latter (cur == prev), it doesn't check for nil, let alone "queue is empty".

You actually need two blocks: one to calculate the next value and another one to determine whether the loop should end.

An infinite sequence seems to make more sense.

#3 - 01/30/2018 07:18 PM - zverok (Victor Shepelev)

> All those conditions are quite different.

Well, you can say that next page fetching "converges" up to "no more pages left" (which in some APIs represented by infinite repetition of the same page, while in other with an empty page), and job queue "converges" to empty queue.

But the more I think about it, the more I feel like just "infinite sequence" would be quite enough. You can make the "converger" or anything from the infinite enumerator, with take_while and other nice Enumerable stuff.

#4 - 01/31/2018 04:55 AM - shan (Shannon Skipper)

I found it really interesting to compare Object#converge with an Object#unfold based on Elixir's Stream.unfold/2. Here's my Ruby implementation:

```
class Object
  def unfold
    Enumerator.new do |yielder|
      next_acc = self

      until next_acc.nil?
        element, next_acc = yield next_acc
        yielder << element
      end
    end
  end
end
```

Comparing examples, when it's converging, #converge really shines.

```
1.0.converge { |x| (x + 2 / x) / 2 }.to_a.last
#=> 1.414213562373095

1.0.unfold { |x| next_x = (x + 2 / x) / 2; [x, (next_x unless x == next_x)] }.to_a.last
#=> 1.414213562373095
```

When the return value isn't mapped at all, #converge is also simpler.

```
2.converge { |x| x ** 2 }.take(4)
#=> [2, 4, 16, 256]

2.unfold { |x| [x, x ** 2] }.take(4)
#=> [2, 4, 16, 256]
```

When it's unfolding, #unfold shines.

```
[0, 1].converge { |f0, f1| [f1, f0 + f1] }.take(10).map(&:first)
#=> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

[0, 1].unfold { |f0, f1| [f0, [f1, f0 + f1]] }.take(10)
#=> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

I'm intrigued by this proposal.

#5 - 01/31/2018 10:10 AM - zverok (Victor Shepelev)

BTW, just infinite enumerator is converted to "converger" without lot of problems:

```
class Object
  def iterate(&block)
    Enumerator.new { |y|
      prev = self
      y << self
      loop do
        cur = block.call(prev)
        y << cur
        prev = cur
      end
    }.lazy
  end
end

p 1.0.iterate { |x| (x + 2 / x) / 2 }.each_cons(2).take_while { |prev, cur| prev != cur }.force.last.last
```

So, "just lazy infinite sequence" is probably the way to go.

#6 - 02/01/2018 08:23 AM - sos4nt (Stefan Schüßler)

Maybe it could be as simple as:

```
class Object
  def iterate
    Enumerator.new { |y|
      obj = self
      loop do
        y << obj
        obj = yield obj
      end
    }
  end
end

1.0.iterate { |x| (x + 2 / x) / 2 }.slice_when(&:==).first.last
#=> 1.414213562373095
```

#7 - 02/01/2018 08:43 AM - zverok (Victor Shepelev)

```
1.0.iterate { |x| (x + 2 / x) / 2 }.slice_when(&:==).first.last
#=> 1.414213562373095
```

(°ل͜°)
Nice.

#8 - 04/19/2018 07:25 AM - zverok (Victor Shepelev)

- Description updated

Updated proposal after discussion in comments.

#9 - 05/17/2018 07:03 AM - matz (Yukihiro Matsumoto)

I am not sure how much we need such method. Any (real-world) use-case?

Matz.

#10 - 05/17/2018 10:17 AM - zverok (Victor Shepelev)

@matz (Yukihiro Matsumoto)
Basically, Object#enumerate is a method for replace *lots* of loop and while with proper Enumerator (like each replaced for). Some examples:

```
# API pagination, as shown above:
fetch(first_page_url).enumerate { |page| fetch(page.next_page_url) if page.has_next? }.
  take_while { |page| !page.nil? } # take all, or having enumerator, take some: first(3)

# Or, traversing tree-like structure:
node.enumerate(&:parent)
```

```
# Or, layouting algorithm (it is pseudocode based on my magic_cloud gem),
# trying to find for some complicated shape the place on board which still is not taken
# by other shapes:
find_initial_place(shape).enumerate(&:next_place).take_while { |place| board.can_take?(place) }.last

# as shown above, it is even the most readable way to enumerate from some `start` with `succ`
# or other incrementor:
1.enumerate(&:succ).take_while { |i| something_something }
# ^ ok, that can be represented with (1..Float::INFINITY).each, or new "endless ranges",
# but `enumerate` seems cleaner

2.enumerate { |i| i**2 }
# ^ what about this? Sequences like this are quite helpful in a lot of algorithms
```

I really believe it could be a ton of other examples, Enumerator seems to win a lot of games used this way.

#11 - 05/18/2018 03:39 PM - zverok (Victor Shepelev)

As asked by @shyouhei (Shyouhei Urabe) at Ruby dev meeting log, I released it as an experimental gem.

Note: It is meant just as a proof-of-concept, I believe that the only way for this method to be useful is to be in language core. Nobody this days will install a gem with one method extending one core class :)

I also considered several options of syntax, and it seems like extending Object is the only clean way.

Also, here are some more working examples of usage (from gem's README):

```
# Find next Tuesday
require 'date'
Date.today.enumerate(&:succ).detect { |d| d.wday == 2 }
# => #<Date: 2018-05-22 ((2458261j,0s,0n),+0s,2299161j)>


# Tree navigation
# ---------------
require 'nokogiri'
require 'open-uri'

# Find some element on page, then make list of all parents
p Nokogiri::HTML(open('https://www.ruby-lang.org/en/'))
  .at('a:contains("Ruby 2.2.10 Released")')
  .enumerate(&:parent)
  .take_while { |node| node.respond_to?(:parent)  }
  .map(&:name)
# => ["a", "h3", "div", "div", "div", "div", "div", "div", "body", "html"]

# Pagination
# ----------
require 'octokit'

Octokit.stargazers('rails/rails')
# ^ this method returned just an array, but have set `.last_response` to full response, with data
# and pagination. So now we can do this:
p Octokit.last_response
  .enumerate { |response| response.rels[:next].get } # pagination: `get` fetches next Response
  .first(3)                                          # take just 3 pages of stargazers
  .flat_map(&:data)
# `data` is parsed response content (stargazers themselves)
  .map { |h| h[:login] }
# => ["wycats", "brynary", "macournoyer", "topfunky", "tomtt", "jamesgolick", ...
```

#12 - 05/18/2018 10:47 PM - inopinatus (Joshua GOODALL)

looks like syntactic sugar for yield_self into an enumerator, c.f.

```
1.yield_self { |value| Enumerator.new { |y| loop { y << value; value = value.succ } } }.take(5)
#=> [1, 2, 3, 4, 5]
```

also

- This is a generator, no question. Definition of generator: *"a generator looks like a function but behaves like an iterator."*
- Name of #enumerate will be absurd in some code e.g. sending enumerate to an array won't enumerate the array!
- Therefore if implemented should be called Object#generator to avoid confusing name of #enumerate.
- Restriction to infinite sequences seems unnecessary, why not honour StopIteration?

#13 - 05/18/2018 11:02 PM - shevegen (Robert A. Heiler)

> Object#generate to avoid confusing name of #enumerate.

I think #generate as name is problematic too. We may expect this
method on Object to generate an object, which I do not think
aligns with what the proposed suggestion does.

> Nobody this days will install a gem with one method extending one core class :)

I think the comment by shyouhei was not so much as "keep it as a gem and
separate from ruby core", but more to show how/if useful it may be, e. g.
the comment "start as a gem".

#14 - 05/19/2018 10:43 AM - zverok (Victor Shepelev)

@inopinatus I updated the README with reasoning about name. Copying from there:

The reasons behind the name #enumerate:

- Core method names should be short and mnemonic, not long and descriptive. (Otherwise, we'd have yield_each instead of each, yield_each_and_collect instead of map, parallel_each instead of zip and don't even get me started on reduce). It is if you can't guess what core method *exactly* does *just* from the name, more important to have it easily remembered and associative.
- Code constructs using the name should be spellable in your head. I pronounce it "1: enumerate succeeding numbers", "last result: enumerate all next" and so on. Judge yourself.
- Concept is present in other languages and frequently named iterate (for example, Clojure and Scala). As we call our iterators Enumerator, it is logical to call the method enumerate.
- Once you got it, the name is hard to confuse with anything (the only other slightly similar name in core is #to_enum, but it is typically used in a very far context).

The only other reasonable option I can think about is deduce, as an antonym for reduce which makes the opposite thing. Elixir follows this way, calling the methods fold (for our reduce) and unfold (for method proposed).

#15 - 05/20/2018 12:56 AM - inopinatus (Joshua GOODALL)

I agree with the abstract reasons, but for exactly those reasons I think #enumerate is a confusing and poorly chosen name whilst #generator is a good name.

Although specifically I think this is an over-reach:

> we call our iterators Enumerator, it is logical to call the method enumerate

That doesn't follow, because this new proposed method is not a general-purpose constructor for all Enumerators. It is a specialist constructor for a certain kind of iterators, a kind that is conventionally called a *generator*. The most obvious example is "Pseudo-random number *generator*" which has exactly this pattern: from a given seed, a function is applied to produce an infinite sequence.

The obvious day-to-day issue with #enumerate is when we have an array:

```
[1,2,3,4,5].enumerate { ... }
```

Almost anyone approaching this cold will assume that #enumerate is a synonym for #each, because that fits the dictionary definition of the word "enumerate". Same with #iterate.

Whilst #generator is clearly a different concept. Also, it's simply the correct term in a compsci sense. "Dear object, please use this block to return a generator over your data". I really like the concept and I know I will use it, but I think it should be very precisely named.

#16 - 10/08/2018 03:10 PM - jwmittag (Jörg W Mittag)

[Oops. I didn't see #14781.]

The general version of this method is the category-theoretical dual of a fold (called inject and reduce in Ruby), so its name should probably reflect that. unfold is a name that is used in some other communities, e.g. Haskell, Elixir, Scalaz.

inject is a name that comes from Smalltalk, where the method is called inject: aValue into: aBinaryBlock, and a dual method could be called for example generate: aValue from: aBinaryBlock; so, by analogy to inject, a Ruby version could be called generate which also works as a dual of reduce.

So, I would propose generate or unfold.

If Object#generate(&blk) is considered too "heavy", an alternative could be Enumerator::generate(initial, &blk). In fact, it makes much more sense for this method to be seen as a factory / constructor for enumerators than an instance method of all objects.

#17 - 11/20/2019 05:04 AM - zverok (Victor Shepelev)

This one can be closed (Enumerator#produce was implemented in 2.7)

#18 - 07/29/2022 03:09 PM - zverok (Victor Shepelev)

- Status changed from Open to Closed

#19 - 08/01/2024 06:29 AM - mame (Yusuke Endoh)

- Related to Feature #20625: Object#chain_of added