Defining an Enumerator that requires a state is currently troublesome. For example, it is not really possible to define an equivalent of `Lazy#drop` in Ruby without making an assumption on the implementation.

To address this, I propose that we:

(a) guarantee that a new `Yielder` object will be given for each enumeration
(b) add a 'state' attribute to `Yielder`.

This way, one could implement `Lazy#drop` in a way similar to:

```ruby
class Enumerator::Lazy < Enumerator
  def drop(n)
    n = n.to_i
    Lazy.new(self) do |yielder, *values|
      yielder.state ||= n
      if yielder.state > 0
        yielder.state -= 1
      else
        yielder.yield(*values)
      end
    end
  end
end
```

Note that (a) is currently true for Ruby MRI, JRuby and Rubinius, but it is not explicit in the documentation.

Related issues:

- Related to Ruby master - Bug #7696: Lazy enumerators with state can't be rewound
- Related to Ruby master - Feature #8840: Yielder#state

I understand the motivation, and how it works. It is very simple. But I hesitate to introduce state easily in this age of functional programming. Let me think for a while.

And tell me if anyone has a better idea to address the issue.

Matz.

#3 - 08/31/2013 07:23 PM - judofyr (Magnus Holm)

On Sat, Aug 31, 2013 at 12:52 AM, marcandre (Marc-Andre Lafortune) <ruby-core@marc-andre.ca> wrote:

Defining an Enumerator that require a state is currently troublesome. For example, it is not really possible to define an equivalent of `Lazy#drop` in Ruby without making an assumption on the implementation.
Can't you just use the closure?

class Enumerator::Lazy < Enumerator
  def drop(n)
    n = n.to_i
    Lazy.new(self) do |yielder, *values|
      if n > 0
        n -= 1
      else
        yielder.yield(*values)
      end
    end
  end
end

#4 - 09/01/2013 04:39 AM - marcandre (Marc-Andre Lafortune)

judofyr (Magnus Holm) wrote:

  Can't you just use the closure?

Your example will fail if iterated a second time.
It will also not work correctly when using rewind and next. Check #7696.

#5 - 09/01/2013 01:23 PM - akr (Akira Tanaka)

2013/9/1 marcandre (Marc-Andre Lafortune) ruby-core@marc-andre.ca:

  Your example will fail if iterated a second time.
  It will also not work correctly when using rewind and next. Check #7696.

How about adding a method to wrap an enumerator to add a state?

% ruby -e '  
  class Enumerator
    def with_state(init)
      Enumerator.new { |y|
        state = init.dup
        self.each { |v|
          y.yield [state, v]
        }
      }
    end
  end

  class Enumerator::Lazy
    def drop2(n)
      e = with_state([n])
      Enumerator.new { |y|
        e.each { |remain, v|
          if remain[0] == 0
            y.yield v
          else
            remain[0] -= 1
          end
        }
      }
    end
  end

  e = (1..42).lazy
  p e.drop2(40).to_a
  p e.drop2(40).to_a
  [
  41, 42
  ]

  Tanaka Akira

#6 - 10/01/2013 05:56 PM - naruse (Yui NARUSE)

Need marcandre's reply
I'm sorry for my late reply, I'm way back on many things I want to do.

The proposition of with_state is interesting, but I personally find it leads to complex/convoluted solutions and is cumbersome to use. Note that the given implementation of drop2 is slightly incomplete as it needs to return a lazy enumerator, so Enumerator.new needs to be followed by a call to lazy.

Would you explain the incompleteness concretely?
I couldn't understand.

Sure. With your code above:

e.drop2(40).map(&:odd?)  #=> [true, false]
# expected lazy enumerator, as with original drop:
e.drop(40).map(&:odd?)  #=> #<Enumerator::Lazy: #<Enumerator::Lazy: #<Enumerator::Lazy: 1..42>:drop(40)>:map>

Here is another implementation using with_state that returns a lazy enumerator:

class Enumerator::Lazy
  def drop3(n)
    Lazy.new(with_state(remain: n)) do |y, (state, v)|
      if state[:remain] == 0
        y.yield v
      else
        state[:remain] -= 1
      end
    end
  end
end

This implementation doesn't look so bad. It's probably quite a bit slower than using a Yielder#state method though.

Would you explain the incompleteness concretely?

Sure. With your code above:

e.drop2(40).map(&:odd?)  #=> [true, false]
# expected lazy enumerator, as with original drop:
e.drop(40).map(&:odd?)  #=> #<Enumerator::Lazy: #<Enumerator::Lazy: #<Enumerator::Lazy: 1..42>:drop(40)>:map>

Here is another implementation using with_state that returns a lazy enumerator:

class Enumerator::Lazy
  def drop3(n)
    Lazy.new(with_state(remain: n)) do |y, (state, v)|
      if state[:remain] == 0
        y.yield v
      else
        state[:remain] -= 1
      end
    end
  end
end

This implementation doesn't look so bad. It's probably quite a bit slower than using a Yielder#state method though.

Would you explain the incompleteness concretely?

Sure. With your code above:

e.drop2(40).map(&:odd?)  #=> [true, false]
# expected lazy enumerator, as with original drop:
e.drop(40).map(&:odd?)  #=> #<Enumerator::Lazy: #<Enumerator::Lazy: #<Enumerator::Lazy: 1..42>:drop(40)>:map>

Here is another implementation using with_state that returns a lazy enumerator:

class Enumerator::Lazy
  def drop3(n)
    Lazy.new(with_state(remain: n)) do |y, (state, v)|
      if state[:remain] == 0
        y.yield v
      else
        state[:remain] -= 1
      end
    end
  end
end

This implementation doesn't look so bad. It's probably quite a bit slower than using a Yielder#state method though.

Would you explain the incompleteness concretely?

Sure. With your code above:

e.drop2(40).map(&:odd?)  #=> [true, false]
# expected lazy enumerator, as with original drop:
e.drop(40).map(&:odd?)  #=> #<Enumerator::Lazy: #<Enumerator::Lazy: #<Enumerator::Lazy: 1..42>:drop(40)>:map>

Here is another implementation using with_state that returns a lazy enumerator:
This implementation doesn't look so bad. It's probably quite a bit slower than using a Yielder#state method though.

Thank you. I understand.

I still like with_state than Yielder#state because it limits stateful behaviors into a method.

--
Tanaka Akira

#11 - 01/30/2014 06:17 AM - hsbt (Hiroshi SHIBATA)
- Target version changed from 2.1.0 to 2.2.0

#12 - 05/11/2014 08:55 AM - akr (Akira Tanaka)
I have another idea now.

How about combining Enumerator.new and Enumerator#lazy addition to closure?

class Enumerator::Lazy
  def drop4(n)
    Enumerator.new {|y|
      remain = n
      self.each {|v|
        if remain == 0
          y.yield v
        else
          remain -= 1
        end
      }
    }.lazy
  end
end

e = (1..42).lazy.drop4(40)

# e is an Enumerator::Lazy object
p e #=> #<Enumerator::Lazy: #<Enumerator::Generator:0x007f1bd457cf50>:each>

# e.map(&:odd?) returns an Enumerator::Lazy object
p e.map(&:odd?) #=> #<Enumerator::Lazy: #<Enumerator::Lazy: #<Enumerator::Generator:0x007f1bd457cf50>:each>:map>

# first e.to_a works
p e.to_a #=> [41, 42]

# second e.to_a works
p e.to_a #=> [41, 42]

# e.next and e.rewind works
p e.next #=> 41
p e.next #=> 42
e.rewind
p e.next #=> 41
p e.next #=> 42

#13 - 10/22/2017 02:12 AM - knu (Akinori MUSHA)
- Status changed from Feedback to Rejected

I guess the API is not good enough if you have to do yielder.state ||= ..., and it looks like akr's suggestion works.

Since there has been no feedback, I'm closing this.