Ruby - Feature #6166

Enumerator::Lazy#pinch

03/18/2012 12:47 AM - trans (Thomas Sawyer)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:		

Description

In previous issue #6158 it has been determined that Enumerator::Lazy#take should be lazy. But an eager form of #take would still be useful.

To this end I'll suggest Enumerator::Lazy#pinch. Examples of usage:

```
e.lazy.pinch 1
```

```
e.lazy.pinch 1..2
```

```
e.lazy.pinch 1,2
```

It is basically equivalent to calling to_a[index], but has the advantage of being a single invocation instead of two, and reads better.

The #pinch method would be strictly a Lazy method and have no counterpart in Enumerable.

History

#1 - 03/18/2012 09:10 AM - trans (Thomas Sawyer)

Happy St. Patty's Day ;)

I don't seen any green on this site... *pinch*

#2 - 03/23/2012 06:49 PM - shugo (Shugo Maeda)

Hello,

trans (Thomas Sawyer) wrote:

> In previous issue #6158 it has been determined that Enumerator::Lazy#take should be lazy. But an eager form of #take would still be useful.
>
> To this end I'll suggest Enumerator::Lazy#pinch. Examples of usage:
>
> ```
> e.lazy.pinch 1
> ```
>
> ```
> e.lazy.pinch 1..2
> ```
>
> ```
> e.lazy.pinch 1,2
> ```
>
> It is basically equivalent to calling to_a[index], but has the advantage of being a single invocation instead of two, and reads better.

Enumerator::Lazy#pinch provides a random access feature for Enumerator::Lazy, but Enumerator::Lazy is not random accessible in general, so I prefer explicit conversion like to_a[1, 2] to pinch. How about others?

#3 - 03/26/2012 11:28 PM - trans (Thomas Sawyer)

I like having a method myself b/c it reads better. I think "pinch" conveys the sort of "closing action" of the de-lazying.

Of course, #fetch would work as well, and that's a standard method, but it's interface only accepts an index, not a range, so it's too limited as is.

#4 - 03/30/2012 02:17 AM - mame (Yusuke Endoh)

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

Hello,

If this kind of operations occur frequently, I think it is worth
to add a method. But I'm not sure because Lazy just entered trunk.

My current opinion. I like this style:

e.lazy.pinch(1, 2) == e.lazy.drop(1).first(2)

because it is:

- more explicit than pinch
- more efficient than to_a[1, 2] (when e is very long)

--
Yusuke Endoh mame@tsg.ne.jp

#5 - 04/01/2012 02:26 AM - matz (Yukihiro Matsumoto)

- Status changed from Assigned to Rejected

I like #first better.

Matz.

#6 - 04/01/2012 02:38 AM - trans (Thomas Sawyer)

But #first can't give a range. e.g. (2..3) or (2,2). So there is no way to get such without de-lazying whole enumeration, which defeats purpose of lazy.

#7 - 04/01/2012 02:48 AM - trans (Thomas Sawyer)

Also are you sure #first should be non-lazy? e.g.

max_records = 1000000

recs = records.first(max_records)

recs.pinch(@page_no, 25).each do |page_recs|
...
end

Hmm... maybe #page is better name then #pinch?

In any case, the point is I think every enumeration method that can be lazy should be lazy, and a special method that's not an enumerable method should allows us to extract subsets. That method (whether called #pinch or something else) would be the most flexible and optimized since it is designed to very task of de-lazying and extraction.

#8 - 04/01/2012 02:54 AM - matz (Yukihiro Matsumoto)

I am sure #first not to be lazy. And you can combine it with #drop to take the value in the middle.
#pinch does not suggest the behavior you've proposed to me, a non native English speaker.

Matz.

#9 - 04/01/2012 03:47 AM - trans (Thomas Sawyer)

I don't care about name "#pinch", but functionality. Using #first with #drop is not always optimal.

1. Given d = [index,length].

enum.drop(d.first).first(d.last)

1. Given a range (e.g. rng = 2...4):

if rng.exclude_end?
enum.drop(rng.begin).first(rng.end-1)
else
enum.drop(rng.begin).first(rng.end)
end

enum(*d) and enum(rng) is much better.

Also I take it #drop is lazy?

#10 - 04/01/2012 05:52 AM - marcandre (Marc-Andre Lafortune)

Sorry to be late to the party.

If this method was called slice, would it be more acceptable?

It would be easy to remember, as enum.slice would be the same as enum.to_a.slice except it would stop the iteration as early as possible.

#11 - 04/02/2012 01:38 AM - matz (Yukihiro Matsumoto)

What happens when you call #pinch (or whatever) twice on same lazy sequence?

e.g.

lz = (1..100).lazy
lz.pinch(0,2)
lz.pinch(0,2)

If second call to pinch gives [1,2], lazy sequence needs to keep all generated values inside (that makes lazy sequence very inefficient both in time and space-wise).
If it gives [3,4], I don't think it's the expected behavior for most of us.

#pinch implies offset which is not well fit with the concept of lazy sequence which is not always indexable.

Matz.

#12 - 04/02/2012 03:15 AM - trans (Thomas Sawyer)

It should have no effect on lz either way. It does not act in place.

lz = (1..100).lazy
lz.pinch(0,2) #=> [1,2]
lz.pinch(0,2) #=> [1,2]

Marc-Andre's suggestion of #slice is better name. I forgot about that method, but it is exactly what I intended --so long as #slice is not going to be lazy itself.

Sequences are indexable in that #to_a can be called, the method follows the same procedure but stops when the end sentinel is reached.

Hmmm... I suppose another option would be to let #to_a take arguments.

lz = (1..100).lazy
lz.to_a(0,2) #=> [1,2]

But that may not be good idea b/c #to_a does not take arguments in other classes.

#13 - 04/02/2012 10:42 AM - matz (Yukihiro Matsumoto)

I am not sure what you mean by "does not act in place". Lazy sequence has position. Retrieving value from sequence moves its position, even calling to_a. So to make #pinch work as if "not acting in place", sequence needs to keep the values inside, everytime it generates.

I don't want to do that since it hinders the best benefit from "lazy" sequence.

Matz.

#14 - 04/02/2012 06:23 PM - trans (Thomas Sawyer)

I don't understand what you mean by "So to make #pinch work as if "not acting in place", sequence needs to keep the values inside, everytime it generates." Pinch is no different than #to_a except that it does not need to "resolve" all elements of the enumerable, only the ones up to requested sentinel.

In other words, you reject this b/c you say it "hinders the best benefit from "lazy" sequence." But the result is that end user must use #to_a instead (e.g. enum.to_a[1..2]), which makes lazy utterly pointless. Moreover, if pinch causes this issue, why doesn't drop(b).first(e) cause the same issue, since "pinch" is effectively just a convenience for the same?

I wonder if we are not thinking about the same idea. Maybe some code would clear things up. "Pinch", which as Marc-Andre points out is effectively #slice, can be implemented basically as:

```
def slice(b, e=nil)
  if e
    drop(b).first(e)
  else
    case b
    when Range
      if b.exclude_end?
        drop(b.begin).first(b.end-1)
      else
        drop(b.begin).first(b.end)
      end
```

```
    else
      first(b).last
    end
  end
end
```