Ruby master - Feature #3715

Enumerator#size and #size=

08/19/2010 05:02 AM - marcandre (Marc-Andre Lafortune)

Status:	Rejected	
Priority:	Normal	
Assignee:		
Target version:	2.0.0	

Description

=begin
It would be useful to be able to ask an Enumerator for the number of times it will yield, without having to actually iterate it.

For example:

(1..1000).to_a.permutation(4).size # => 994010994000 (instantly)

It would allow nice features like:

```
class Enumerator
def with_progress
return to_enum :with_progress unless block_given?
out_of = size || "..."
each_with_index do |obj, i|
puts "Progress: #{i} / #{out_of}"
yield obj
end
puts "Done"
end
end

# To display the progress of any iterator, one can daisy-chain with_progress:
20.times.with_progress.map do
# do stuff here...
end
```

This would print out "Progress: 1 / 20", etc..., while doing the stuff.

*** Proposed changes ***

- Enumerator#size *

```
call-seq:
e.size          -> int, Float::INFINITY or nil
e.size {block}  -> int
```

Returns the size of the enumerator.
The form with no block given will do a lazy evaluation of the size without going through the enumeration. If the size can not be determined then +nil+ is returned.
The form with a block will always iterate through the enumerator and return the number of times it yielded.

```
(1..100).to_a.permutation(4).size # => 94109400
loop.size # => Float::INFINITY

a = [1, 2, 3]
a.keep_if.size        # => 3
a                     # => [1, 2, 3]
a.keep_if.size{false}  # => 3
a                     # => []

[1, 2, 3].drop_while.size             # => nil
[1, 2, 3].drop_while.size{|i| i < 3}  # => 2
```

- Enumerator#size= *

call-seq:
e.size = sz

Sets the size of the enumerator. If +sz+ is a Proc or a Method, it will be called each time +size+ is requested, otherwise +sz+ is returned.

first = [1, 2, 3]
second = [4, 5]
enum = Enumerator.new do |y|
first.each{|o| y << o}
second.each{|o| y << o}
end
enum.size   # => nil
enum.size = ->(e){first.size + second.size}
enum.size   # => 5
first << 42
enum.size   # => 6

- Kerne#to_enum / enum_for *

The only other API change is for #to_enum/#enum_for, which can accept a block for size calculation:

class Date
def step(limit, step=1)
unless block_given?
return to_enum(:step, limit, step){|date| (limit - date).div(step) + 1}
end
...
end
end

*** Implementation ***

I implemented the support for #size for most builtin enumerator producing methods (63 in all).

It is broken down in about 20 commits: http://github.com/marcandre/ruby/commits/enum_size

It begins with the implementation of Enumerator#size{=}: http://github.com/marcandre/ruby/commit/a92feb0

A combined patch is available here: http://gist.github.com/535974

Still missing are Dir#each, Dir.foreach, ObjectSpace.each_object, Range#step, Range#each, String#upto, String#gsub, String#each_line.

The enumerators whose #size returns +nil+ are:
Array#{r}index, {take|drop}_while
Enumerable#find{_index}, {take|drop}_while
IO: all methods

*** Notes ***

- Returning +nil+ *

I feel it is best if IO.each_line.size and similar return +nil+ to avoid side effects.

We could have Array#find_index.size return the size of the array with the understanding that this is the maximum number of times the enumerator will yield. Since a block can always contain a break statement, size could be understood as a maximum anyways, so it can definitely be argued that the definition should be the maximum number of times.

- Arguments to size proc/lambda *

My implementation currently passes the object that the enumerator will call followed with any arguments given when building the enumerator.

If Enumerator had getters (say Enumerator#base, Enumerator#call, Enumerator#args, see feature request #3714), passing the enumerator itself might be a better idea.

- Does not dispatch through name *

It might be worth noting that the size dispatch is decided when creating the enumerator, not afterwards in function of the class & method name:

[1,2,3].permutation(2).size # => 6
[1,2,3].to_enum(:permutation, 2).size # => nil

- Size setter *

Although I personally like the idea that #size= can accept a Proc/Lambda for later call, this has the downside that there is no getter, i.e. no way to get the Proc/Lambda back. I feel this is not an issue, but an alternative would be to have a #size_proc and #size_proc= setters too (like Hash).

I believe this addresses feature request #2673, although maybe in a different fashion. http://redmine.ruby-lang.org/issues/show/2673
=end

Related issues:

Related to Ruby master - Feature #2673: the length for an enumerator generate...	**Closed**	
Related to Ruby master - Feature #3714: Add getters for Enumerator	**Closed**	
Related to Ruby master - Feature #6636: Enumerable#size	**Closed**	**06/24/2012**

History

#1 - 08/19/2010 05:12 AM - marcandre (Marc-Andre Lafortune)

=begin
I forgot to point to basic specs: http://github.com/marcandre/rubyspec/commit/43bab#diff-1

--
Marc-André
=end

#2 - 08/20/2010 10:08 AM - runpaint (Run Paint Run Run)

=begin
An enumerator is effectively immutable. The existence of #size= and potentially invoking a Proc for each call to #size, implies that its maximum size will change over the course of the iteration. Is this likely? If not, we can remove #size=, and treat the Proc as a thunk, which would be simpler and faster.

I find the semantics of Enumerator#size's block argument confusing. I suggest that #size accept no arguments, use the size Proc, if available, or otherwise fallback to the brute force approach. That is, to find the size of an enumerator, e, constrained by a block, b, the programmer should call obj.e(&b).to_enum.size. It will often be the case that obj.e(&b) returns an object that responds to #size itself, allowing the general case of obj.e(&b).size.

I'd rather #to_enum/#enum_for take a Proc as an argument rather than a block literal. As methods can only accept a single block, I prefer it affect the method as a whole; not a specific aspect thereof.

Lastly, some of the RDoc needs word wrapping. :-)
=end

#3 - 08/20/2010 01:25 PM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:31785] [Feature #3715] Enumerator#size and #size="
on Fri, 20 Aug 2010 10:08:44 +0900, Run Paint Run Run redmine@ruby-lang.org writes:

|An enumerator is effectively immutable. The existence of #size= and
|potentially invoking a Proc for each call to #size, implies that its
|maximum size will change over the course of the iteration. Is this
|likely? If not, we can remove #size=, and treat the Proc as a thunk,
|which would be simpler and faster.

We have discussed the issue in the ruby-dev before. The conclusion we had then was that:

- it is nice to have a way to tell the number of items without actual iteration.
- but Enumerator#size is not a good API, since not all enumerators would have the way to tell the numbers of items. Permutations and combinations are rather exceptions.
- Enumerator#size= was worse, since it would make enumerators mutable.

- we haven't got the better API after some discussion, so we
  preferred not adding it to adding unsatisfying #size.

```
                matz.
```

=end

#4 - 08/21/2010 01:43 PM - marcandre (Marc-Andre Lafortune)

=begin
Hi,

On Thu, Aug 19, 2010 at 9:08 PM, Run Paint Run Run redmine@ruby-lang.org wrote:

> An enumerator is effectively immutable.

Indeed, it would be better to specify the size (value or proc) at creation time. I suggest some possibilities further down.

> I find the semantics of Enumerator#size's block argument confusing.

Maybe if it was called "count" instead which is also a verb, it would less confusing?

> It will often be the case that obj.e(&b) returns an object that responds to #size itself, allowing the general case of obj.e(&b).size.

I'm suggesting the block form just for completion's sake. The main gain is the lazy form without block and if it is deemed too confusing, the block form can definitely be left out.

On Fri, Aug 20, 2010 at 12:25 AM, Yukihiro Matsumoto matz@ruby-lang.org wrote:

> Hi,
>
> In message "Re: [ruby-core:31785] [Feature #3715] Enumerator#size and #size="
> on Fri, 20 Aug 2010 10:08:44 +0900, Run Paint Run Run redmine@ruby-lang.org writes:
>
> We have discussed the issue in the ruby-dev before. The conclusion we
> had then was that:
>
> * it is nice to have a way to tell the number of items without
> actual iteration.

Indeed

> * but Enumerator#size is not a good API, since not all enumerators
> would have the way to tell the numbers of items. Permutations and
> combinations are rather exceptions.

I'm not sure I follow. The builtin enumerators can tell the numbers of items (more than 70), except a very small number of exceptions (mainly the IO ones). Users programming their own enumerators would decide to specify a lazy counter if they choose to.

> * Enumerator#size= was worse, since it would make enumerators mutable.

Agreed, I should have thought about this better. Has the following possibility been discussed on the ruby-dev thread?

Enumerator.new's call sequence could be:

- Enumerator.new(obj, method = :each, *args)
- Enumerator.new(obj, count, method = :each, *args)
- Enumerator.new { |y| ... }
- Enumerator.new(count) { |y| ... }

where count is either a value or a Proc/method.

Similarly, to_enum's call sequence would be:

- obj.to_enum(method = :each, *args)
- obj.to_enum(count, method = :each, *args)

--

Marc-André

=end

#5 - 08/21/2010 11:15 PM - Eregon (Benoit Daloze)

=begin
Hi,

On 18 August 2010 22:02, Marc-Andre Lafortune redmine@ruby-lang.org wrote:

> It would be useful to be able to ask an Enumerator for the number of times it will yield, without having to actually iterate it. [...]
> This would print out "Progress: 1 / 20", etc..., while doing the stuff.

That seems cool, indeed.

> *** Proposed changes ***
>
> - Enumerator#size *
>
> call-seq:
> e.size -> int, Float::INFINITY or nil
> e.size {block} -> int
>
> Returns the size of the enumerator.
> The form with no block given will do a lazy evaluation of the size without going through the enumeration. If the size can not be determined then +nil+ is returned.

Interesting, though I do not feel INFINITY to be the right answer if
it is a loop (but yes, it should obviously be bigger than anything you
compare to).

> The form with a block will always iterate through the enumerator and return the number of times it yielded.
>
> (1..100).to_a.permutation(4).size # => 94109400
> loop.size # => Float::INFINITY
>
> a = [1, 2, 3]
> a.keep_if.size # => 3
> a # => [1, 2, 3]
> a.keep_if.size{false} # => 3
> a # => []
>
> [1, 2, 3].drop_while.size # => nil
> [1, 2, 3].drop_while.size{|i| i < 3} # => 2

What is the interest compared to Enumerable#count ? (also, #size with
a block does not feel natural)

> a = [1, 2, 3]
> a.keep_if.count # => 3
> a # => []

Here the result is identical.

> [1, 2, 3].drop_while.count # => 1

Here it is different, and your method behaves strangely.
The block is yielded 3 times, but your method returns 2 ?

And if you wanted the size of the resulting Array, it seem more obvious to do:
[1,2,3].drop_while { |i| i < 3 }.size # => 1

> - Enumerator#size= *

Having a mutator on Enumerator does not make sense to me, as
previously mentioned by others.
--

However, it is really interesting to know the times an enumerator will yield.

I believe checking for nil is really not beautiful in the code, so I
think it would be nicer if it actually use the block form if it can
not be determined directly.
But this could introduce unwanted side-effects (the #drop_while is an example).
These side-effects should be known, and I do not see any real use for
destructive method with #size, do you ?

About infinite enumerator, it should just yield forever (so calling #each).

I would then propose that Enumerable#count (without block) use this
code to give directly the times it would yield, or if it can not, do
the real iteration like now.

--
And this seems to be already supposed in the documentation of #count:
Returns the number of items in enum, where #size is called if it
responds to it, otherwise the items are counted through enumeration.

So, these enumerators should just provide #size, and it should work.

However, it does not behave like that:

> a = (1..1000000000000).to_enum # => #
> def a.size; 3 end
> a.size # => 3
> a.count # => take ages
> Defining on the class (Range) does not work either.

So it seems to be a problem with the implementation (in enum.c, line
~122): it does not check if it #respond_to?(:size):
It calls #each with count_all_i() as a block, because there is no
block and no argument.
count_all_i() being a simple block which just increment a counter (memo).

I believe Enumerable#count should respect its documentation, and that
your work would help to implement #size on the Enumerator which sizes
can be known it immediately. Then we would have a consistent behavior,
already described in the documentation !

What do you think ?

Regards,
B.D.

=end

#6 - 08/24/2010 02:44 AM - runpaint (Run Paint Run Run)

=begin

> I find the semantics of Enumerator#size's block argument confusing. Maybe
> if it was called "count" instead which is also a verb, it would less
> confusing?

I don't think so. As Benoit notes, Enumerator already responds to #count, with
different semantics.

> Interesting, though I do not feel INFINITY to be the right answer if it is a
> loop (but yes, it should obviously be bigger than anything you compare to).

Infinity makes sense to me given that Object#loop is, indeed, an infinite loop.

- but Enumerator#size is not a good API, since not all enumerators would have the way to tell the numbers of items. Permutations and combinations are rather exceptions.

> I'm not sure I follow. The builtin enumerators can tell the numbers of items
> (more than 70), except a very small number of exceptions (mainly the IO
> ones). Users programming their own enumerators would decide to specify a lazy
> counter if they choose to.

I concur with Marc-André. The bulk of his patch proves that this ability can typically be retrofitted. The pathological case is an enumerator unable to calculate its size without iteration, yet to do so would be impractical--perhaps due to side-effects. However, I see this as analogous to Range#each inasmuch as it is a method that does not work for certain, narrowly-defined types of input.

=end

#7 - 09/18/2010 12:55 AM - Eregon (Benoit Daloze)

=begin
On 23 August 2010 19:44, Run Paint Run Run runrun@runpaint.org wrote:

> I find the semantics of Enumerator#size's block argument confusing. Maybe if it was called "count" instead which is also a verb, it would less confusing?

I don't think so. As Benoit notes, Enumerator already responds to #count, with different semantics.

> Interesting, though I do not feel INFINITY to be the right answer if it is a loop (but yes, it should obviously be bigger than anything you compare to).

Infinity makes sense to me given that Object#loop is, indeed, an infinite loop.

Yes, the only thing is returning a Float does not seems right (ideally it should be an Integer infinity, but that is no sense in the implementation). So, I guess Infinity is a good answer.

> * but Enumerator#size is not a good API, since not all enumerators would have the way to tell the numbers of items. Permutations and combinations are rather exceptions.
>
> I'm not sure I follow. The builtin enumerators can tell the numbers of items (more than 70), except a very small number of exceptions (mainly the IO ones). Users programming their own enumerators would decide to specify a lazy counter if they choose to.

I concur with Marc-André. The bulk of his patch proves that this ability can typically be retrofitted. The pathological case is an enumerator unable to calculate its size without iteration, yet to do so would be impractical--perhaps due to side-effects. However, I see this as analogous to Range#each inasmuch as it is a method that does not work for certain, narrowly-defined types of input.

So, what about using Marc-André work to improve Enumerable#count (with no arguments or block),
by defining #size in the Enumerator which can support it, and make #count behaves the way described (I quoted it earlier)
(check if #size is defined, and then just call it).

The block form seems rather complicated and can easily be done in other ways, if I understand correctly, so I think this should not be implemented right now.

The only problem is then #count for iterators which have side-effects as Array#drop_while,
but who would want to actually have an efficient way to #count something like that ?

I think a solution is simply to not implement #size for these (so not implement for the enumerators which mutate the receiver), as it is not interesting and may give an unexpected result (eg: actually not dropping the elements).

I would like to contribute to this feature, and so to adapt the patch with what I just described.

Does it seems a good idea to you, should I go ahead?
Marc-André, are you according to this? (do you want to do it yourself,
is it ok I work with your code ?)

The first thing would then to make a list of enumerators for which we
define #size,
and to choose if we implement it for infinite enumerators, or let it
loop forever.

Regards,
B.D.

=end

#8 - 06/28/2011 06:14 AM - nahi (Hiroshi Nakamura)

- Target version changed from 1.9.3 to 2.0.0

#9 - 03/18/2012 06:07 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Rejected

As matz already rejeted I hereby close this issue. If you still need it, please open a new ticket with revisied AIPs. Thank you.