Ruby master - Feature #7444

Array#product_set

11/27/2012 02:44 PM - marcandre (Marc-Andre Lafortune)

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

Description

I'd like to propose Array#product_set to return the product set of arrays (aka cartesian product)

```
deck = [1..13, %i(spades hearts diamond clubs)].product_set
    # => <#Enumerator ...>
deck.first(2) # => [[1, :spades], [2, :spades]]
```

product_set would return an enumerator if no block is given. It should raise an error if an element of the array is not an Enumerable, like Array#transpose or #zip do.

Although Array.product would be acceptable too, I feel that an instance method of array is best in the case, in the same way that transpose is an instance method and not a class method.

The name "product_set" is a correct mathematical term. Although the synonym "cartesian_product" would also be acceptable, I propose "product_set" because it is shorter and cute too. I feel it is even clearer than product; the first time I head of product I was convinced that [2,3,7].product # => 42.

Addressing objections raised in #6499:

1) This is not for the sake of symmetry, but because often we have an array of the arrays we want a product of.

It is cumbersome to write arrays.first.product(*arrays[1..-1]) or similar and it hides what is going on.

Writing arrays.product_set is much nicer.

2) The goal is not mainly to get a lazy version, but more to make the API better. The fact that it returns an Enumerator if no block is given is just a bonus :-)

3) [].product_set.to_a # => [[]]

This can be seen from a cardinality argument, or for example because array.repeated_permutation(n) == Array.new(n, array).product_set.to_a and array.repeated_permutation(0) == [[]].

Related issues:

Has duplicate Ruby master - Feature #8970: Array.zip and Array.product	**Open**

History

#1 - 11/27/2012 10:05 PM - trans (Thomas Sawyer)

I'd prefer Array.product, all things being the same.

But you have given me a neat idea. In Ruby Facets there is a method Enumerable#every. It works like so:

[1,2,3].every + 2 #=> [3,4,5]
[1,2,3].every * 2 #=> [2,4,6]

#every is a HOM (a higher-order method). You made me realize another good form of this would be one that applies to each new result. I am not sure what a good name for it would be, but for the moment lets just call it #apply. Then in your case of #product.

deck.apply.product #=> [[1, :spades], [2, :spades], ...]

Only problem is I haven't had any success it getting the Ruby gods to come around on HOMs :(

#2 - 12/03/2012 06:28 AM - Anonymous

IF this feature will not be considered feature creep by others, with respect to
eg. Array#repeated_combination etc. (I do not possess 100% knowledge of Array /

Enumerable / Enumerator features),
THEN +1 to your variant syntax Array.product(...)

As for HOMs, they are beautiful, but their place is in the private libraries
of connoisseurs. Beginners have hard enough time understanding Enumerator already.

#3 - 12/03/2012 07:27 AM - trans (Thomas Sawyer)

> As for HOMs, they are beautiful, but their place is in the private libraries
> of connoisseurs.

Man, I couldn't disagree with that more. It's delegation man, delegation!

#4 - 12/03/2012 06:23 PM - alexeymuranov (Alexey Muranov)

marcandre (Marc-Andre Lafortune), here are just some things that first came to my mind:

1. I do not think that an "Array instance method" is a good place for this function: otherwise every time a new function of multiple arguments is wanted, a new instance method would be added to Array (like the product of an array of numbers that you mentioned). It seems that what is needed here is some advanced multiple dispatch. Enumerator::product(*enums) would also look reasonable to me.
2. The name product_set seems to suggest that the result is a Set, but it is an Enumerator.
3. [(1..13).to_a, %w(spades hearts diamond clubs)].inject(:product) does a very similar thing to what the proposed method would do.

#5 - 12/04/2012 12:43 AM - marcandre (Marc-Andre Lafortune)

alexeymuranov (Alexey Muranov) wrote:

> marcandre (Marc-Andre Lafortune), here are just some things that first came to my mind:
>
> 1. I do not think that an "Array instance method" is a good place for this function: otherwise every time a new function of multiple arguments is wanted, a new instance method would be added to Array (like the product of an array of numbers that you mentioned). It seems that what is needed here is some advanced multiple dispatch. Enumerator::product(*enums) would also look reasonable to me.

I'm not suggesting that functions of multiple (generic) arguments be instance methods of Array. I'm proposing that this function of multiple *array* arguments (or array-like) be an instance of Array, like Array#transpose is.

> 1. The name product_set seems to suggest that the result is a Set, but it is an Enumerator.

A google search on "product set" confirms its meaning.

> 1. [(1..13).to_a, %w(spades hearts diamond clubs)].inject(:product) does a very similar thing to what the proposed method would do.

Not really. arrays.product_set.to_a and arrays.inject(:product) give only the same result if arrays.size == 2. If < or > 2, results are different. Finally, the inject isn't lazy.

#6 - 12/04/2012 01:25 AM - stomar (Marcus Stollsteimer)

> > 1. [(1..13).to_a, %w(spades hearts diamond clubs)].inject(:product) does a very similar thing to what the proposed method would do.
>
> Not really. arrays.product_set.to_a and arrays.inject(:product) give only the same result if arrays.size == 2. If < or > 2, results are different.

Please elaborate.

#7 - 12/04/2012 02:24 AM - marcandre (Marc-Andre Lafortune)

stomar (Marcus Stollsteimer) wrote:

> > Not really. arrays.product_set.to_a and arrays.inject(:product) give only the same result if arrays.size == 2. If < or > 2, results are different.
>
> Please elaborate.

I'm not sure how I was not clear, but in concrete examples:

```
[].inject(:product) # => nil
[].product_set.to_a # => [[]]

[[1,2]].inject(:product) # => [1,2]
[[1,2]].product_set.to_a # => [[1], [2]]

[[1,2], [3,4], [5,6]].inject(:product) # => [[[1, 3], 5], [[1, 3], 6], [[1, 4], 5], [[1, 4], 6], [[2, 3], 5],
[[2, 3], 6], [[2, 4], 5], [[2, 4], 6]]
[[1,2], [3,4], [5,6]].product_set.to_a # => [[1, 3, 5], [1, 3, 6], [1, 4, 5], [1, 4, 6], [2, 3, 5], [2, 3, 6],
 [2, 4, 5], [2, 4, 6]]

# etc...
```

As noted, I also have to call to_a on product_set to compare.

#8 - 12/04/2012 02:51 AM - stomar (Marcus Stollsteimer)

Thanks and sorry for being unclear.
It seemed to me that you did not specify the expected behavior for the proposed method in the case of e.g. 3 arrays.

#9 - 12/04/2012 03:13 AM - alexeymuranov (Alexey Muranov)

marcandre (Marc-Andre Lafortune) wrote:

> alexeymuranov (Alexey Muranov) wrote:
>
> 1. I do not think that an "Array instance method" is a good place for this function: otherwise every time a new function of multiple arguments is wanted, a new instance method would be added to Array (like the product of an array of numbers that you mentioned). It seems that what is needed here is some advanced multiple dispatch. Enumerator::product(*enums) would also look reasonable to me.
>
> I'm not suggesting that functions of multiple (generic) arguments be instance methods of Array. I'm proposing that this function of multiple *array* arguments (or array-like) be an instance of Array, like Array#transpose is.

In your example the first element was not an Array but a Range. The method is called on the *outer* array, and constructs a certain product of its elements. I would see no reason to forbid such "collecting" operations for other element types (Set, Integer, etc.).

transpose is special because it treats the array and its contents as a whole as a matrix.

> 1. The name product_set seems to suggest that the result is a Set, but it is an Enumerator.
>
> A google search on "product set" confirms its meaning.

Exactly, and this is not what the method returns (enumerator) :).

In usual terminology, the Cartesian product of sets is a *product set*, the Cartesian product of categories is a *product category*, the product of enumerators would be a *product enumerator*, the product of arrays would be a *product array*, etc. (The product of enumerators and the product of arrays are not defined, but can be defined of course.)

> 1. [(1..13).to_a, %w(spades hearts diamond clubs)].inject(:product) does a very similar thing to what the proposed method would do.
>
> Not really. arrays.product_set.to_a and arrays.inject(:product) give only the same result if arrays.size == 2. If < or > 2, results are different. Finally, the inject isn't lazy.

Yes, but i wanted to point out that this operation would look more natural to me if defined in terms of a binary operation on enumerators:

[(1..13), %i(spades hearts diamond clubs), [:deck_1, :deck_2]].inject([[]], :smartly_collecting_product)

where smartly_collecting_product is to be defined lazy and with other desired properties.

Edited 2012-12-05.

#10 - 12/05/2012 04:11 AM - alexeymuranov (Alexey Muranov)

=begin

Ok, is think i understand why it can be defined as an instance methods of (({Array})): it is called on an array and produces a sequence of arrays of the same length. Then maybe (({Array#each_combination}))?

How about this:

class Enumerator
def collecting_product(enum)
...
end

```
def collecting_product!(enum)
  # ...
end
```

end

class Array
def each_combination(&block)
result = [[]].to_enum
each do |element|
result.collecting_product! element.to_enum
end
block_given? ? result.each(&block) : result
end
end
=end

#11 - 03/01/2014 06:41 PM - marcandre (Marc-Andre Lafortune)

- Has duplicate Feature #8970: Array.zip and Array.product added

#12 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)

- Target version deleted (2.6)