Ruby Basic Enumerable Methods

What is Enumerable?

Enumerable is a **collection of iteration methods**, a Ruby module, and a big part of what makes Ruby a great programming language.

Enumerable includes helpful methods like:

`map` **`select`** **`inject`**

Enumerable methods work by giving them a block.

In that block you tell them what you want to do with every element.

For example:

```
1.[1,2,3].map { |n| n * 2 }
```

Gives you a new array where every number has been doubled.

Exactly what happens depends on what method you use, **`map`** helps you transform all the values, **`select`** lets you filter a list & **`inject`** can be used to add up all the values inside an array.

There are over 20 Ruby Enumerable methods.

The `each_with_index` Method

This method is nearly the same as #**`each`**, but it provides some additional functionality by yielding two **block variables** instead of one as it iterates through an array. The first variable's value is the element itself, while the second variable's value is the index of that element within the array. This allows you to do things that are a bit more complex.

For example, if we only want to print every other word from an array of strings, we can achieve this like so:

```
fruits = ["apple", "banana", "strawberry", "pineapple"]

fruits.each_with_index { |fruit, index| puts fruit if index.even? }

#=> apple
#=> strawberry
#=> ["apple", "banana", "strawberry", "pineapple"]
```

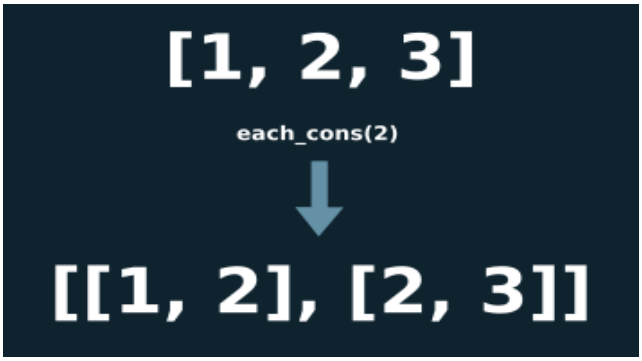
Just like with the `#each` method, `#each_with_index` returns the original array it's called on.

The Each_Cons Method

My new favorite `Enumerable` method is `each_cons`!

Here's why:

This method is really useful, you can use it to find n-grams or to check if a sequence of numbers is contiguous when combined with `all?`, another `Enumerable` method.



`each_cons` gives you sub-arrays of size `n`, so if you have `[1,2,3]`, then `each_cons(2)` will give you `[[1,2], [2,3]]`.

Let's see an example:

```
1.numbers = [3,5,4,2]

2.numbers.sort.each_cons(2).all? { |x,y| x == y - 1 }
```

This code starts by sorting the numbers, then calling `each_cons(2)`, which returns an `Enumerator` object, and then it calls the `all?` method to check if all the elements match the condition.

Here is another example, where I use `each_cons` to check if a character is surrounded by the same character, like this: `xyx`.

```
1.str = 'abcxyx'

2.str.chars.each_cons(3).any? { |a,b,c| a == c }
```

There is more!

If you wanted to know how many times this pattern occurs, instead of getting a `true` / `false` result, you can just change `any?` to `count`.

What I find even more fascinating is the implementation for the `each_cons` method.

```
1.array = []

2.each do |element|
```

3.array << element

4.array.shift if array.size > n

5.yield array.dup if array.size == n

`6.end`

The implementation starts with an empty Ruby array, then it iterates through the elements using `each`.

Everything is pretty standard until here. But then it adds the element to the array and it trims the array (using Array#shift) if the size is bigger than what we want.

The size is the argument to `each_cons`.

Then it yields a `dup` of the array if the array has the requested size.

I think this is genius, because it keeps a 'sliding window' sort of effect into our enumerable object, instead of having to mess around with array indexes.

More Fascinating Methods

Method	Description
count	Exactly what the name says, count things that evaluate to true inside a block
group_by	Group enumerable elements by the block return value. Returns a hash
partition	Partition into two groups. Returns a two-dimensional array
any?	Returns `true` if the block returns `true` for ANY elements yielded to it
all?	Returns `true` if the block returns `true` for ALL elements yielded to it
none?	Opposite of `all?`
cycle(n)	Repeat ALL the elements n times, so if you do `[1,2].cycle(2)` you would have `[1,2,1,2]`
find	Like `select`, but it returns the first thing it finds
inject	Accumulates the result of the previous block value & passes it into the next one. Useful for adding up totals
zip	Glues together two enumerable objects, so you can work with them in parallel. Useful for comparing elements & for generating hashes
map	Transforms every element of the enumerable object & returns the new version as an array

`#each` is an iterator method you've seen plenty of times before now that comes pre-packaged with the Array and Hash and Range classes and it basically just goes through each item in the object you

called it on and passes it to the block that you specified. It will return the original collection that it was called on:

```
> [1,2,3].each { |num| print "#{num}! " }
1! 2! 3! => [1,2,3]
```

Sometimes you also want to know what position in the array you are... so that sounds like a good chance to use Enumerable's **`#each_with_index`**, which will pass that position into the block as well:

```
> ["Cool", "chicken!", "beans!", "beef!"].each_with_index do |item, index|
>   print "#{item} " if index%2==0
> end
Cool beans! => ["Cool", "chicken!", "beans!", "beef!"]
```

What if I want to keep only the even numbers that are in an array? The traditional way would be to build some sort of loop that goes through the array, checks whether each element is even, and starts populating a temporary array that we will return at the end. It might look something like:

```
class Array
  def keep_evens
    result_array = []
    for num in self
      result_array << num if num % 2 == 0
    end
    return result_array
  end
end

> my_array = [1,2,3,4,5,6,7,8,100]
> my_array.keep_evens
=> [2,4,6,8,100]
```

That's too much code and too much hassle. When all you're doing is pulling out, or *selecting*, certain items based on some criteria, you'd be better served using Enumerable's **`#select`** instead. It will run the block on every item of your object (whether array or hash or whatever) and return a new object that contains only those items for which the original block returned `true`:

```
> my_array.select{|item| item%2==0 }
=> [2,4,6,8,100]      # wow, that was easy.
```

You win this round, Ruby. What if instead of selecting only a few items we want to keep all items but modify them somehow? That sounds a lot like we're doing something and **collect**ing the results, doesn't it? **`#collect`** will run your block and give you an object filled with whatever your block returned each time. Ruby says:

```
> my_array.collect{|num| num**2 }
=> [4,16,36,64,10000]
```

Enumerable Iterators Cheat Sheet

- #**`each`** returns the original object it was called on because it's really used for its side effects and not what it returns
- #**`each_with_index`** passes not just the current item but whatever position in the array it was located in.
- #**`select`** returns a new object (e.g. array) filled with only those original items where the block you gave it returned `true`
- #**`map`** returns a new array filled with whatever gets returned by the block each time it runs.

Up until now, all the methods we've seen run essentially independent operations on each element of our array or hash. What if we want to do something that keeps track of the result as we iterate? Like, say, summing up the elements of an array?

For that we need to use #**`inject`** (aka #**`reduce`**), which passes not just the element but whatever was returned by the previous iteration into the block. You can either specify the initial value or it will just default to the first item of the array. It ultimately returns whatever the result of the last iteration is. Here's a way to sum an array:

```
> my_array.inject(0){|running_total, item| running_total + item }
=> 120
```

Some Other Handy Methods

Enumerable is a large bunch of methods and you'll only use a half-dozen of them regularly but there are some others that you should be familiar with as well. The full list is available in the docs here.

- #**`any?`** returns true/false (see the question mark?) and answers the question, "do ANY of the elements in this object pass the test in my block?". If your block returns true on any time it runs, `any?` will return true.
- #**`all?`** returns true/false and answers the question, "do ALL the elements of this object pass the test in my block?". Every time the block runs it must return true for this method to return true.
- #**`none?`** returns true only if NONE of the elements in the object return true when the block is run.
- #**`find`** returns the first item in your object for which the block returns true.

Awesome but less common methods

Ruby Basic Enumerable Methods

- #**group_by** will run your block and return a hash that groups all the different types of returns from that block. For example:

```
> names = ["James", "Bob", "Joe", "Mark", "Jim"]
> names.group_by{|name| name.length}
=> {5=>["James"], 3=>["Bob", "Joe", "Jim"], 4=>["Mark"]}
```

- #**grep** returns an array with those items that actualy match the specified criteria (RegEx) (using a `===` match)

```
> names.grep(/J/)
=> ["James", "Joe", "Jim"]
```

Some of the methods you've already seen and use are part of Enumerable too
-- **#include?**, **#sort**, **#count** etc.

When you use the Enumerable methods, you'll sometimes see what's called an **enumerator** object pop up, usually if you forget to give them a parameter that they want like a block. What the heck is that?

Consider it an implementation detail of Enumerator. As we said before, the methods that are part of Enumerable rely on the underlying collections' `#each` method to work. `enumerable` is basically a go-between for the original collection and Enumerator. It's not really something you'll be using right off the bat but it's useful for gaining a better understanding of Enumerable.