Enumerable Part 1?

Let Ruby Count the Ways

Luke DeWitt - DTC Ruby Meetup - 7/2/2014 - Ruby Beginners Talk

E-Mail - luke@dewittsoft.com / Twitter - lanewalkerx

Enumerable Documentation built in

— Ruby Interactive - ri

```
> ri Enumerable
```

or

```
> ri -i
```

for interactive mode

Note: You must generate docs for this to work "`> rvm docs generate-ri`"

Ruby Collection Objects and Enumerable

Definitions -

Collection Object The Well Grounded Rubyist
- an object that responds to the method calls `each` **or** `each_entry`

Enumerable Merriam-Webster
- capable of being counted
- able to put into one-to-one correspondence with positive integers

Ruby's Collection Objects

- Ruby's Major Container Classes
  - Array
  - Hash
- Ruby's Minor Container Classes
  - Ranges
  - Sets

Construction of Collection Objects

- **Classes must define an `each` method to use `Enumerable`**
- `Enumeralbe` **in return gives collection-related behaviors to objects of the class.**
- **If using max, min, or sort functions must define `<=>`**
- `<=>` **Comparison operator**
  - `<, <=, ==, >=, >`

Construction of Collection Objects

Mixing Enumerable into Classes

```
class Bin
  include Enumerable

  def each
    # insert code here
  end

end
```

What `Enumerable` gives you

- `Bin` now has ability to call any instant method in `Enumerable`
  - Another way to find Enumerable's method's

```
> Enumerable.instance_methods(false).count
=> 48

> Enumerable.instance_methods(false).sort
```

Returns =>

What `Enumerable` gives you

```
=> [:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :detect,
:drop, :drop_while, :each_cons, :each_entry, :each_slice, :each_with_index,
:each_with_object, :entries, :find, :find_all, :find_index, :first,
:flat_map, :grep, :group_by, :include?, :inject, :lazy, :map, :max, :max_by,
:member?, :min, :min_by, :minmax, :minmax_by, :none?, :one?, :partition,
:reduce, :reject, :reverse_each, :select, :slice_before, :sort, :sort_by,
:take, :take_while, :to_a, :zip]
```

How the `each` method does it's job

- `each` must yield items, one at a time, to a code block
  - Varies from class to class
  - Array
    - Yields the first elemnt, then the second, third, etc...
  - Hash
    - Yields Key/Value pairs in the form of 2-element arrays

Unlocking `Enumerable` in your code Example 1

```
class Rainbow
  include Enumerable
  def each
    yield "red"
    yield "orange"
    yield "yellow"
    yield "green"
    yield "blue"
    yield "indigo"
    yield "violet"
  end
end
```

Unlocking `Enumerable` in your code Example 1

```
> r = Rainbow.new
 => #<Rainbow:0x007faa69182638>
> r.each { |color| puts "Next color: #{color}" }
  Next color: red
  Next color: orange
  Next color: yellow
  Next color: green
  Next color: blue
  Next color: indigo
  Next color: violet
```

`Enumerable` **Boolean Methods**

- **These methods return true or false depending on certain criteria**

```
=> [:all?, :any?, :include?, :member?, :none?, :one?]
```

NOTE: `include?` **and** `member?` **are synonyms**

`Enumerable` **searching capabilities**

```
=> [:detect, :find, :find_all, :select]
```

- `find` **locates first element which returns true**
- `find_all` **returns a new collection containing elements that match criteria**

NOTE:

`find?` **and** `detect?` **are synonyms**

`find_all` **and** `select` **are synonyms**

Enumerable **searching capabilities**

- find

```
> a=[1,2,3,4,5,6,7,8,9,10]
> a.find {|n| n > 5}
  => 6
```

- find_all

```
> a.find_all {|i| i > 5}
  => [6, 7, 8, 9, 10]
```

`Enumerable` **element-wise operations**

— **Collections contain "special-status" objects here are the tools** `Enumerable` **has for those objects**

```
=> [:drop, :first, :max, :min, :take]
```

NOTE: No general `last` **method in** `Enumerable`**, imagine a class with an infinitely yielding each method. See Die example.**

Enumerable element-wise operations

— first

```
> [1,2,3,4].first
  => 1
> [4,3,2,1].first
  => 4
```

— min and max

```
> [4,3,2,1].min
  => 1
> [4,3,2,1].max
  => 4
```

Enumerable **element-wise operations**

— :take

```
> states = %w{ NJ NY CT MA VT FL }
  => ["NJ", "NY", "CT", "MA", "VT", "FL"]
> states.take(2)
  => ["NJ", "NY"]
```

— :drop

```
> states.drop(2)
  => ["CT", "MA", "VT", "FL"]
> states
  => ["NJ", "NY", "CT", "MA", "VT", "FL"]
```

`Enumerable` **relatives of each**

— `Enumerable` **gives you other methods similar to** `each`**, these methods cycle through the entire collection and return elements from that collection.**

```
=> [:reverse_each, :each_with_index, :each_slice, :each_cons, :inject]
```

`Enumerable` relatives of each

`reverse_each` iterates backwards through an enumerable.

```
> states.reverse_each {|s| puts "#{s}"}
FL
VT
MA
CT
NY
NJ
 => ["NJ", "NY", "CT", "MA", "VT", "FL"]
```

NOTE: Do not do this on infinite iterators if you call it on the Die class it will throw ruby into an infinite loop.

`Enumerable` **relatives of each**

`each_with_index` **this yields an extra piece of information the ordinal position of the item.**

```
> names = ["Tom", "Jerry", "Sylvester", "Tweety"]
  => ["Tom", "Jerry", "Sylvester", "Tweety"]
> names.each_with_index { |char, i| puts "#{i}. #{char}" }
  0. Tom
  1. Jerry
  2. Sylvester
  3. Tweety
  => ["Tom", "Jerry", "Sylvester", "Tweety"]
```

`Enumerable` relatives of each

`each_slice` and `each_cons` break up a collection in two different ways. The former handles each element only once, while the latter creates groupings at each element.

Enumerable **relatives of each**

```
> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> a.each_slice(3) {|slice| p slice}
  [1, 2, 3]
  [4, 5, 6]
  [7, 8, 9]
  [10]
  => nil
```

`Enumerable` **relatives of each**

```
> a.each_cons(3) {|cons| p cons}
  [1, 2, 3]
  [2, 3, 4]
  [3, 4, 5]
  [4, 5, 6]
  [5, 6, 7]
  [6, 7, 8]
  [7, 8, 9]
  [8, 9, 10]
  => nil
```

NOTE: So this passes to the block each consecutive subarray of size 3.

`Enumerable` relatives of each

`inject` **Combines all elements of a collection by applying an operation, specified by a block or a symbol that names a method or operator.**

NOTE: `inject` **synonym for** `reduce`

`Enumerable` **relatives of each**

This way shows you step-by-step what is happening with `inject`.

```
> [1,2,3,4].inject(0) do |accumulator, number|
>   puts "adding #{accumulator} and #{number}...#{accumulator+number}"
>   accumulator + number
> end
  adding 0 and 1...1
  adding 1 and 2...3
  adding 3 and 3...6
  adding 6 and 4...10
  => 10
```

`Enumerable` **relatives of each**

This is the short hand way of doing the same thing.

```
> [1,2,3,4].inject(0) {|acc,n| acc+n}
=> 10
```

This is not limited to mathematical operations.

```
> longest = %w{ cat sheep bear }.inject do |memo, word|
>     memo.length > word.length ? memo : word
> end
=> "sheep"
```

`Enumerable` more great methods and ideas Part 2?

- `cycle`
- `grep`
- `group_by`
- `map`

Defining comparison classes <=>.

Deeper dive into murky waters of enumerators the next dimension in enumerability

References

- The Well-Grounded Rubyist 2nd Edition
  - David A. Black
  - Manning © 2014
- Programming Ruby 1.9 & 2.0
  - Dave Thomas
  - Pragmatic Programmers © 2013
- Merriam Webster
  - http://www.merriam-webster.com/

References

- Samuel Mullen
  - http://www.samuelmullen.com/2012/01/up-and-running-with-ruby-interactive-ri/

Additional Sides