Lazy Ruby

An Exploration of Enumerators

DC Ruby Users' Group
Monthly Meeting: August 13, 2009

Michael Harrison
michaelharrison.ws
@goodmike

Taking the pulse

Who here...

- is familiar with Enumerable module?
- is using or has tried Ruby 1.9?
- has programmed in a functional language?

Roadmap

- What is laziness?
- What are enumerators?
- What’s interesting about this stuff?

Disclaimer

- The `for` loop still makes the world go round.
- Lazy evaluation and lazy sequences make some problems easier to solve, and may seem more elegant to you. Or not.
- No Fibonacci numbers have been abused in the preparation of this presentation

Lazy evaluation

- Loosely defined as the strategy of evaluating expressions only when needed, and only as much as needed.
- A core feature of languages like Haskell, Erlang, and Clojure.
- Present in most languages, at least in the `if` control structure, which short-circuits evaluation

What’s good about it?

- It’s always better not to do work you don’t have to do.
- Code can be more concise, readable without explicit control structures.
- You can represent, and use, infinite sequences as though they were finite.

Enter the enumerator

- Related to enumerables
- An enumerator is like an *external* iterator through a collection.
- It allows you to delay iteration through the collection.
- Available in Ruby 1.8 as an extension. In the core of Ruby 1.9.
- All subsequent examples are Ruby 1.9

A simple example

```
> my_enum = [1,2,3,4].to_enum
> my_emum.next  #=> 1
> my_emum.next  #=> 2
```

`to_enum` uses the collection's `each` method to generate the enumerable. As a shortcut, you can simply type

```
> my_enum = [1,2,3,4].each
```

Other examples

- You can create an enumerator that uses a method other than `each`:

```
> my_enum = "cat".enum_for(:each_char)
```

- If the method takes parameters, you can pass them to `enum_for`:

```
> my_enum = (1..10).enum_for(:each_slice, 3)
> my_enum.next # => [1, 2, 3]
```

-- from *Programming Ruby 1.9*

Generators

You can create an enumerator with `Enumerator.new &block`

```
natural_numbers = Enumerator.new do |yielder|
  number = 1
  loop do
    yielder.yield number
    number += 1
  end
end
```

Generators

This generates a lazy infinite sequence:

The variable `natural_numbers` is in essence equal to the set of natural numbers. As in, ***all of them.***

Filters

Calling #select on an infinite sequence is a bad idea. So:

```
class Enumerator
  def lazy_select(&block)
    Enumerator.new do |yielder|
      self.each do |val|
      yielder.yield(val) if block.call(val)
      end
    end
  end
end
```

-- After Brian Candler's suggestion in [ruby-core:19679], cited in *Programming Ruby 1.9*

Filters

Say I want the first five palindrome numbers that are divisible by 47?

```
p natural_numbers
    .lazy_select {|n| n % 47 == 0}
    .lazy_select {|n| palindrome_number?(n)}
    .first(5)

# => [141, 282, 1551, 14241, 15651]
```

Very little *noise* in this code. No explicit control structures for looping or breaking.

More laziness

Why stop with a lazy select?

```
def lazy_map(&block)
    Enumerator.new do |yielder|
      self.each do |value|
        yielder.yield(block.call(value))
      end
    end
end
```

More laziness

```
> p natural_numbers.lazy_map {|n| n*n}.take(10)
# => first 10 squares,
     [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

It’s not just for numbers

- Numbers are easy examples, but imagine:
- You need to parse the lines of a monster log file until you find ten lines that are similar.
- You need to search the feed from a remote service for a term.
- You need to transform the members of a collection until you get five results that satisfy some condition