Extracted from:

Best of Ruby Quiz

Volume One

This PDF file contains pages extracted from Best of Ruby Quiz, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragmaticprogrammer.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

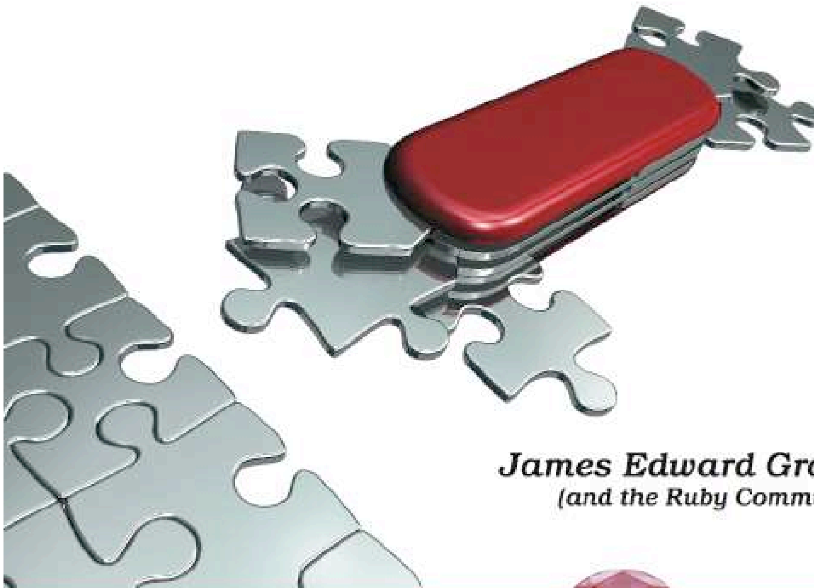
Copyright © 2005 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Programmers

Best of Ruby Quiz



James Edward Gray II
(and the Ruby Community)

Answer 8

From page 18

Roman Numerals

Solving this quiz is easy, but how easy? Well, the problem gives us the conversion chart, which is just crying out to be a Hash:

roman_numerals/simple.rb
```
ROMAN_MAP = { 1    => "I",
              4    => "IV",
              5    => "V",
              9    => "IX",
              10   => "X",
              40   => "XL",
              50   => "L",
              90   => "XC",
              100  => "C",
              400  => "CD",
              500  => "D",
              900  => "CM",
              1000 => "M" }
```

That's the version from my code, but most solutions used something very similar.

From there we just need to_roman() and to_arabic() methods, right? Sounded like too much work for a lazy bum like me, so I cheated. If you build a conversion table, you can get away with just doing the conversion one way:

roman_numerals/simple.rb
```
ROMAN_NUMERALS = Array.new(3999) do |index|
  target = index + 1
  ROMAN_MAP.keys.sort { |a, b| b <=> a }.inject("") do |roman, div|
    times, target = target.divmod(div)
    roman << ROMAN_MAP[div] * times
  end
end
```

CLICK HERE to purchase this book now.

This is the to_roman() method many solutions hit on. I just used mine to fill an Array. The algorithm here isn't too tough. Divide the target number by each value there is a Roman numeral for copy the numeral that many times reduce the target, and repeat. Ruby's divmod() is great for this.

From there, it's trivial to wrap a Unix filter around the Array. However, I do like to validate input, so I did one more little prep task:

roman_numerals/simple.rb

```
IS_ROMAN = / ^ M{0,3}
               (?:CM|DC{0,3}|CD|C{0,3})
               (?:XC|LX{0,3}|XL|X{0,3})
               (?:IX|VI{0,3}|IV|I{0,3}) $ /ix
IS_ARABIC = /^(?:[123]\d{3}|[1-9]\d{0,2})$/
```

That first Regexp is a validator for the Roman letter combinations we accept, split up by powers of ten. The second Regexp is a pattern to match 1..3999, a number in the range we can convert to and from.

Now, we're ready for the Unix filter wrapper:

roman_numerals/simple.rb

```
if __FILE__ == $0
  ARGF.each_line() do |line|
    line.chomp!
    case line
    when IS_ROMAN  then puts ROMAN_NUMERALS.index(line) + 1
    when IS_ARABIC then puts ROMAN_NUMERALS[line.to_i - 1]
    else raise "Invalid input:  #{line}"
    end
  end
end
```

In English that says, for each line of input, see whether it matches IS_ROMAN, and if it does, look it up in the Array. If it doesn't match IS_ROMAN but does match IS_ARABIC, index into the Array to get the match. If none of that is true, complain about the broken input.

Saving Some Memory

If you don't want to build the Array, you just need to create the other converter. It's not hard. J E Bailey's script did both, so let's look at that:

roman_numerals/dual_conversions.rb

```
#!/usr/bin/env ruby

@data = [
["M"  , 1000],
["CM" , 900],
["D"  , 500],
["CD" , 400],
["C"  , 100],
["XC" ,  90],
["L"  ,  50],
["XL" ,  40],
["X"  ,  10],
["IX" ,   9],
["V"  ,   5],
["IV" ,   4],
["I"  ,   1]
]

@roman = %r{^[CDILMVX]*$}
@arabic = %r{^[0-9]*$}

def to_roman(num)
  reply = ""
  for key, value in @data
    count, num = num.divmod(value)
    reply << (key * count)
  end
  reply
end

def to_arabic(rom)
  reply = 0
  for key, value in @data
    while rom.index(key) == 0
      reply += value
      rom.slice!(key)
    end
  end
  reply
end

$stdin.each do |line|
  case line
  when @roman
    puts to_arabic(line)
  when @arabic
    puts to_roman(line.to_i)
  end
end
```

Joe Asks...

toRoman() or to_roman()?

The methods in J E's solution were originally toRoman() and toArabic(). These method names use an unusual (in Ruby circles) naming convention often referred to as *camelCase*. Typical Ruby style is to name methods and variables in *snake_case* (such as to_roman() and to_arabic()). We do typically use a variant of the former (with a capital first letter) in the names of classes and modules, though.

Why is this important?

Well, with any language first you need to learn the grammar, but eventually you want to know the slang, right? Same thing. Someday you may want to write Ruby the way that Ruby gurus do.

I told you we all used something similar to my Hash. Here it's just an Array of tuples.

Right below that, you'll see J E's data identifying Regexp declarations. They're not as exact as my versions, but certainly they are easier on the eyes.

Next we see a to_roman() method, which looks very familiar. The implementation is almost identical to mine, but it comes out a little cleaner here since it isn't used to load an Array.

Then we reach the method of interest, to_arabic(). The method starts by setting a reply variable to *0*. Then it hunts for each Roman numeral in the rom String, increments reply by that value, and removes that numeral from the String. The ordering of the @data Array ensures that an *XL* or *IV* will be found before an *X* or *I*.

Finally, the code provides the quiz-specified Unix filter behavior. Again, this is very similar to my own solution, but with conversion routines going both ways.

Romanizing Ruby

Those are simple solutions, but let's jump over to Dave Burt's code for a little Ruby voodoo. Dave's code builds a module, RomanNumerals, with

CLICK HERE to purchase this book now.

to_integer() and from_integer(), similar to what we've discussed previously. The module also defines is_roman_numeral?() for checking exactly what the name claims and some helpful constants such as DIGITS, MAX, and REGEXP.

roman_numerals/roman_numerals.rb

```
# Contains methods to convert integers to Roman numeral strings, and vice versa.
module RomanNumerals

  # Maps Roman numeral digits to their integer values
  DIGITS = {
    'I' => 1,
    'V' => 5,
    'X' => 10,
    'L' => 50,
    'C' => 100,
    'D' => 500,
    'M' => 1000
  }

  # The largest integer representable as a Roman numerable by this module
  MAX = 3999

  # Maps some integers to their Roman numeral values
  @@digits_lookup = DIGITS.inject({
    4 => 'IV',
    9 => 'IX',
    40 => 'XL',
    90 => 'XC',
    400 => 'CD',
    900 => 'CM'}) do |memo, pair|
    memo.update({pair.last => pair.first})
  end

  # Based on Regular Expression Grabbag in the O'Reilly Perl Cookbook, #6.23
  REGEXP = /^M*(D?C{0,3}|C[DM])(L?X{0,3}|X[LC])(V?I{0,3}|I[VX])$/i

  # Converts +int+ to a Roman numeral
  def self.from_integer(int)
    return nil if int < 0 || int > MAX
    remainder = int
    result = ''
    @@digits_lookup.keys.sort.reverse.each do |digit_value|
      while remainder >= digit_value
        remainder -= digit_value
        result += @@digits_lookup[digit_value]
      end
      break if remainder <= 0
    end
    result
  end
```

CLICK HERE to purchase this book now.

```
  # Converts +roman_string+, a Roman numeral, to an integer
  def self.to_integer(roman_string)
    return nil unless roman_string.is_roman_numeral?
    last = nil
    roman_string.to_s.upcase.split(//).reverse.inject(0) do |memo, digit|
      if digit_value = DIGITS[digit]
        if last && last > digit_value
          memo -= digit_value
        else
          memo += digit_value
        end
        last = digit_value
      end
      memo
    end
  end

  # Returns true if +string+ is a Roman numeral.
  def self.is_roman_numeral?(string)
    REGEXP =~ string
  end
end
```

I doubt we need to go over that code again, but I do want to point out one clever point. Notice how Dave uses a neat dance to keep things like *IV* out of DIGITS. In doing so, we see the unusual construct memo.update({pair.last => pair.first}), instead of the seemingly more natural memo[pair.last] = pair.first. The reason is that the former returns the Hash itself, satisfying the continuous update cycle of inject().

Anyway, the module is a small chunk of Dave's code, and the rest is fun. Let's see him put it to use:

roman_numerals/roman_numerals.rb

```
class String
  # Considers string a Roman numeral,
  # and converts it to the corresponding integer.
  def to_i_roman
    RomanNumerals.to_integer(self)
  end
  # Returns true if the subject is a Roman numeral.
  def is_roman_numeral?
    RomanNumerals.is_roman_numeral?(self)
  end
end
class Integer
  # Converts this integer to a Roman numeral.
  def to_s_roman
    RomanNumerals.from_integer(self) || ''
  end
end
```

CLICK HERE to purchase this book now.

First, he adds converters to String and Integer. This allows you to code things such as the following:

```
puts "In the year #{1999.to_s_roman} ..."
```

Fun, but there's more. For Dave's final magic trick he defines a class:

roman_numerals/roman_numerals.rb

```
# Integers that look like Roman numerals
class RomanNumeral
  attr_reader :to_s, :to_i

  @@all_roman_numerals = []

  # May be initialized with either a string or an integer
  def initialize(value)
    case value
    when Integer
      @to_s = value.to_s_roman
      @to_i = value
    else
      @to_s = value.to_s
      @to_i = value.to_s.to_i_roman
    end
    @@all_roman_numerals[to_i] = self
  end

  # Factory method: returns an equivalent existing object if such exists,
  # or a new one
  def self.get(value)
    if value.is_a?(Integer)
      to_i = value
    else
      to_i = value.to_s.to_i_roman
    end
    @@all_roman_numerals[to_i] || RomanNumeral.new(to_i)
  end

  def inspect
    to_s
  end

  # Delegates missing methods to Integer, converting arguments to Integer,
  # and converting results back to RomanNumeral
  def method_missing(sym, *args)
    unless to_i.respond_to?(sym)
      raise NoMethodError.new(
        "undefined method '#{sym}' for #{self}:#{self.class}")
    end
    result = to_i.send(sym,
      *args.map {|arg| arg.is_a?(RomanNumeral) ? arg.to_i : arg })
    case result
```

```
    when Integer
      RomanNumeral.get(result)
    when Enumerable
      result.map do |element|
        element.is_a?(Integer) ? RomanNumeral.get(element) : element
      end
    else
      result
    end
  end
end
```

If you use the factory method get() to create these objects, it's efficient with reuse, always giving you the same object for the same value.

Note that method_missing() basically delegates to Integer at the end, so you can treat these objects mostly as Integer objects. This class allows you to code things like thus:

```
IV = RomanNumeral.get(4)
IV + 5 # => IX
```

Even better, though, is that Dave removes the need for that first step with the following:

roman_numerals/roman_numerals.rb

```
# Enables uppercase Roman numerals to be used interchangeably with integers.
# They are autovivified RomanNumeral constants
# Synopsis:
#   4 + IV           #=> VIII
#   VIII + 7         #=> XV
#   III ** III       #=> XXVII
#   VIII.divmod(III) #=> [II, II]
def Object.const_missing sym
  unless RomanNumerals::REGEXP === sym.to_s
    raise NameError.new("uninitialized constant: #{sym}")
  end
  const_set(sym, RomanNumeral.get(sym))
end
```

This makes it so that Ruby will automatically turn constants like IX into RomanNumeral objects as needed. That's just smooth.

Finally, the listing at the top of the facing page shows Dave's actual solution to the quiz using the previous tools:

CLICK HERE to purchase this book now.

roman_numerals/roman_numerals.rb

```
# Quiz solution: filter that swaps Roman and arabic numbers
if __FILE__ == $0
  ARGF.each do |line|
    line.chomp!
    if line.is_roman_numeral?
      puts line.to_i_roman
    else
      puts line.to_i.to_s_roman
    end
  end
end
```

Additional Exercises

1. Modify your solution to scan free-flowing text documents, replacing all valid Roman numerals with their Arabic equivalents.

2. Create a solution that maps out the conversions similar to the first example in this discussion, but do it without using a 4,000-element Array kept in memory.

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

Best of Ruby Quiz
`pragmaticprogrammer.com/titles/fr_quiz`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/fr_quiz.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com