Jason Clinton

ESSENTIAL CODE AND COMMANDS

Ruby

PHRASEBOOK

Ruby Phrasebook

ISBN-13: 978-0-672-32897-8
ISBN-10: 0-672-32897-6

Library of Congress Cataloging-in-Publication Data:

2005938020

Printed in the United States of America

First Printing August 2008

Trademarks

Warning and Disclaimer

Bulk Sales

Editor-in-Chief
Mark Taub

Development Editor
Michael Thurston

Managing Editor
Patrick Kanouse

Project Editor
Jennifer Gallant

Copy Editor
Geneil Breeze/
Krista Hansing

Indexer
Tim Wright

Proofreader
Carla Lewis/
Leslie Joseph

Technical Editor
Robert Evans

Publishing Coordinator
Vanessa Evans

Multimedia Developer
Dan Scherf

Book Designer
Gary Adair

Introduction

Audience

You can find some great Ruby books on the market. If you are new to Ruby, a friend or someone on the Internet has probably already listed some favorite Ruby books—and you *should* buy those books. But every book has its niche: Each attempts to appeal to a certain need of a programmer.

It is my belief that the best thing this book can do for you is *show you the code*. I promise to keep the chat to a minimum, to focus instead on the quality and quantity of actual Ruby code. I'll also keep as much useful information in as tight a space as is possible.

Unlike any other book on the market at the time of this writing, this book is intended to be a (laptop-bag) "pocket-size" resource that enables you to quickly look up a topic and find examples of practical Ruby code—a topical quick reference, if you will. In each of the topics covered, I try to provide as thorough an approach to each task as the size allows for; there's not as much room for coverage of topical solutions as there is in much larger books with similar goals, such as *The Ruby Way, 2nd Edition* (Sams, 2006), by Hal Fulton. Because of this, other issues that are often given equal priority are relegated to second. For instance, this is

not a tutorial; the code samples have some explanation, but I assume that you have a passing familiarity with the language. Also, when possible, I try to point out issues related to security and performance, but I make no claim that these are my highest priority.

I hope that you find this book a useful tool that you keep next to your keyboard whenever you are *phrase-mongering* in Ruby.

How to Use This Book

I have not intended for this book to be read cover to cover. Instead, you should place your bookmark at the Table of Contents so you can open the book, find the topic you are programming on at the moment, and go immediately to a description of all the issues you might run into.

The content in the book is arranged by topic instead of Ruby class. The benefit is that you can go to one place in this book instead of four or five areas in Ruby's own documentation. (Not that there's anything wrong with Ruby's documentation. It's just that sometimes you are working with several classes at a time and, because Ruby's docs are arranged by class, you have to jump around a lot.)

Conventions

Phrases throughout the book are put in dark gray boxes at the beginning of every topic.

```
Phrases look like this.
```

Code snippets that appear in normal text are in *italics*. All other code blocks, samples, and output appear as follows:

```
# code sample boxes.
```

Parentheses are optional in Ruby in some cases—the rule is: you must have parentheses in your method call *if* you are calling another function in your list of parameters, or passing a literal code block to the method. In all other cases, parentheses are optional. Personally, I'm a sucker for consistency but one of the indisputable strengths of Ruby is the flexibility of the syntax.

In an attempt to have consistency between this book and others, I will (reluctantly) use `.class_method()` to refer to class methods, `::class_variable` to refer to class variables, `#method()` to refer to instance methods, and finally `#var` to refer to instance variables. When referring to variables and methods which are members of the *same* class, I'll use the appropriate `@variable` and `@@class_varriable`.

I know that some people might find these two rules annoying—especially those coming from languages that use the '::' and '.' notation everywhere. In all practicality, you will never be so consistent—and rightfully so. One of Ruby's strengths is that there is a ton of flexibility. In fact, this flexibility has helped make Ruby on Rails so popular. This allowed the creators of Rails to make what appears to be a *domain-specific language* (a language well-suited for a specific kind of work) for web development. But really, all that is going on is a variation on Ruby syntax. And this is one of the many reasons that Ruby is more suitable for a given problem than, say, Python. Python's rigidity ("there should be

one—and preferably only one—obvious way to do it") doesn't lend itself to DSL, so the programmers in that language are forced to use other means (which might or might not turn out to be unpleasant).

I always use single quotes (') in Ruby code *unless* I actually want to make use of the double-quote (") features (interpolation and substitution).

I always put the result of the evaluation of the statement (or block) on the next line with a proceeding #=>, similar to what you would find if you were using irb or browsing Ruby's documentation.

Comments on executable lines of code start with # and are in *italics* to the end of the comment. Comments on #=> lines are in parentheses and are in *italics*.

Acknowledgments

Without the Pragmatic Programmers' freely available 1st Edition of *Programming Ruby*, I would have never discovered the wonderful world of Ruby. The Pickaxe books and the great Ruby community are what make projects like this one possible.

Thanks to my loving partner, Brandon S. Ward, for his infinite patience while working on this book.

Reporting Errata

Readers will almost certainly find topics that they wish were covered which we were overlooked when planning this book. I encourage you to please contact us and let us know what you would like to see included in later editions. Criticisms are also welcome. Contact information can be found in the front-matter of this book.

3

Working with Collections

In Ruby and other dynamic languages, "Collection" is an umbrella term for general-use lists and hashes. The ease of working with these data structures is an attractive feature and one that often contributes to making prototyping in Ruby a pleasurable experience. The implementation details of lists and hashes and their underlying mechanisms are mostly hidden from the programmer leaving him to focus on his work.

As you browse this section, keep in mind that underpinning everything you see here are traditional C-based implementations of lists and hashes; Ruby is attempting to save you the trouble of working with C—but be sure, that trouble saving can come at performance cost.

Slicing an Array

This section has a lot of analogs to the earlier section "`String` to `Array` and Back Again," in Chapter 1, "Converting Between Types." You can slice an `Array` a number of ways:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9][4]
    #=> 5 (a Fixnum object)

[1, 2, 3, 4, 5, 6, 7, 8, 9][4,1]
    #=> [5] (single element Array)

[1, 2, 3, 4, 5, 6, 7, 8, 9][4,2]
    #=> [5, 6]

[1, 2, 3, 4, 5, 6, 7, 8, 9][-4,4]
    #=> [6, 7, 8, 9]

[1, 2, 3, 4, 5, 6, 7, 8, 9][2..5]
    #=> [3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7, 8, 9][-4..-1]
    #=> [6, 7, 8, 9]

[1, 2, 3, 4, 5, 6, 7, 8, 9][2...5]
    #=> [3, 4, 5]

[1, 2, 3, 4, 5, 6, 7, 8, 9][-4...-1]
    #=> [6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8, 9][4..200]
    #=> [5, 6, 7, 8, 9] (no out of range error!)
```

Array Ranges	Positions (Counting Starts at 0, Negative Numbers Count Position from the End)
`A[{start}..{end}]`	`{start}` includes the element; `{end}` includes the element
`A[{start}...{end}]`	`{start}` includes the element; `{end}` excludes the element
`A[{start}, {count}]`	`{start}` includes the element; `{count}` positions from start to include

You might also like to select elements from the Array if certain criteria are met:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9].select { |element| ele-
ment % 2 == 0 }
    #=> [2, 4, 6, 8] (all the even elements)
```

Iterating over an Array

```
[1, 2, 3, 4, 5].each do |element|
    # do something to element
end
```

This is one of the joys of Ruby. It's so easy!

You can also do the trusty old for loop:

```
for element in [1, 2, 3, 4, 5]
    # do something to element
end
```

The difference between a for loop and an #each is that in for, a new lexical scoping is not created. That is, any variables that are created by for or that are in the loop remain after the loop ends.

To traverse the Array in reverse, you can simply use #Arrayreverse#each. Note that in this case, a copy of the Array is being made by #reverse, and then #each is called on that copy. If your Array is very large, this could be a problem.

In order for you get any more specialized than that, however, you need to work with the Enumerator module. For example, you might want to traverse an Array processing five elements at a time as opposed to the one element yielded by #each:

```
require 'enumerator'
ary = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
ary.each_slice(5) { |element| p element }
```

Outputs:

```
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
```

Creating Enumerable Classes

You may find that you need to make information in a given, custom data structure available to the rest of the world. In such a case, if the data structure that you have created to store arbitrary objects implements an `#each` method, the `Enumerable` mix-in will allow anyone who uses your class to access several traversal and searching methods, for free.

```
require 'enumerator'

class NumberStore
        include Enumerable

        attr_reader :neg_nums, :pos_nums

        def add foo_object
                if foo_object.respond_to? :to_i
                        foo_i = foo_object.to_i
                        if foo_i < 0
                                @neg_nums.push foo_i
                        else
                                @pos_nums.push foo_i
                        end
                else
```

```
                        raise "Not a number."
                end
        end

        def each
                @neg_nums.each { |i| yield i }
                @pos_nums.each { |i| yield i }
        end

        def initialize
                @neg_nums = []
                @pos_nums = []
        end
end

mystore = NumberStore.new
mystore.add 5
mystore.add 87
mystore.add(-92)
mystore.add(-1)

p mystore.neg_nums
p mystore.pos_nums

p mystore.grep -50..60
```

Produces:

```
[-92, -1]
[5, 87]
[-1, 5]
```

In the above contrived example, I have created a data structure called `NumberStore` which stores negative numbers in one list and positive numbers in another list. Because the #each method is implemented, methods like `#find`, `#select`, `#map`, and `#grep` become

available. In the last line of the code sample I use the mixed-in method `#grep` to find numbers stored in mystore that are between 50 and 60.

Sorting an `Array`

```
[5, 2, 1, 4, 3].sort
   #=> [1, 2, 3, 4, 5]
```

As long as all the objects stored in the `Array` respond to the `<=>` method, the `Array` will be sorted successfully. If you want to sort by some special criteria, you can supply a block or even map a value to each element that *can* be compared using "`<=>`". Here is a somewhat contrived example (there are many ways to accomplish this):

```
['Platinum', 'Gold', 'Silver', 'Copper'].sort_by do
|award|
    case award
    when 'Platinum': 4
    when 'Gold': 3
    when 'Silver': 2
    when 'Copper': 1
    else 0
    end
end
    #=> ["Copper", "Silver", "Gold", "Platinum"]
```

Above, a numerical value is assigned to each `String` and then the `Array` is sorted by #sort_by using those values.

Word of warning: When sorting numerical values, beware of `Floats`, they can have the value `NaN` (imaginary) which is, of course, not comparable to real numbers. `Array#sort` will fail if your array has such a `NaN`:

```
[1/0.0, 1, 0, -1, -1/0.0, (-1)**(0.5)]
    #=> [Infinity, 1, 0, -1, -Infinity, NaN]
[1/0.0, 1, 0, -1, -1/0.0, (-1)**(0.5)].sort
```

Produces:

```
ArgumentError: comparison of Fixnum with Float
failed
```

Iterating over Nested Arrays

`Array.flatten.each { |elem|` *`#do something`* `}`

You can `#flatten` the `Array` as `I` have done above. For most cases, this works just fine—it's very fast. But it's perhaps not quite as flexible as a recursive implementation:

```
class Array
    def each_recur(&block)
        each do |elem|
            if elem.is_a? Array
                elem.each_recur &block
            else
                block.call elem
            end
        end
    end
end

my_ary = [[1, 2, 3, 4],[5, 6, 7, 8]]
    #=> [[1, 2, 3, 4], [5, 6, 7, 8]]

my_ary.each_recur { |elem| print(elem, " ") }
```

Produces:

```
1 2 3 4 5 6 7 8
```

Modifying All the Values in an Array

`Array#collect`, also known as `Array#map`, is used to modify the values of an `Array` and return a new array.

```
['This', 'is', 'a', 'test!'].collect do |word|
    word.downcase.delete '^A-Za-z'
end
    #=> ["this", "is", "a", "test"]
```

If you want to do this on a nested `Array`, you need something a little stronger:

```
class Array
    def collect_recur(&block)
        collect do |e|
            if e.is_a? Array
                e.collect_recur(&block)
            else
                block.call(e)
            end
        end
    end
end

[[1,2,3],[4,5,6]].collect_recur { |elem| elem**2 }
    #=> [[1, 4, 9], [16, 25, 36]]
```

Sorting Nested Arrays

```
[[36, 25, 16], [9, 4, 1]].flatten.sort
    #=> [1, 4, 9, 16, 25, 36]
```

We have to `#flatten` the `Array` because the `#sort` uses `<=>` to compare two `Arrays`, which in turn, compares

their elements for either all elements being less than all elements in the other `Array` or vice-versa (if neither condition is met they are considered equal). It doesn't descend in to the `Arrays` to sort them. Here is what would happen if we didn't flatten:

```
[[36, 25, 16], [9, 4, 1]].sort
    #=> [[9, 4, 1], [36, 25, 16]]
```

Once again, the first code will work in most cases but a recursive implementation is able to accommodate working with the `Array` in place without destroying the heirarchy (note that this sorts in place, for simplicity):

```
class Array
    def sort_recur!
        sort! do |a,b|
            a.sort_recur! if a.is_a? Array
            b.sort_recur! if b.is_a? Array
            a <=> b
        end
    end
end

p [[36, 25, 16], [9, 4, 1]].sort_recur!
```

Produces:

```
[[1, 4, 9], [16, 25, 36]]
```

Building a Hash from a Config File

```
my_hash = Hash::new
tmp_ary = Array::new

"a = 1\nb = 2\nc = 3\n".each_line do |line|
    if line.include? '='
        tmp_ary = line.split('=').collect { |s|
s.strip }
        my_hash.store(*tmp_ary)
    end
end

p tmp_ary
p my_hash
Produces:
["c", "3"] (from the last loop)
{"a"=>"1", "c"=>"3", "b"=>"2"}
```

This is very similar to an earlier example in the section "Searching Strings," in Chapter 2, "Working With Strings." Here we are processing a simple format config file. This is a sample of what such a file looks like:

```
variable1 = foo
variable2 = bar
variable3 = baz
```

For the sake of simplicity, instead of a File for simulated input, this example uses a simple `String` with some `\n` (newline) separators.

In plain English, those inner lines mean, "Take the current line and `call the #split` on it, splitting on the '=' character; pass each element of the resulting two-element `Array` in to the block; `call the #strip method on the Strings` to remove any whitespace, and return the modified `Array` to `tmp_ary`. `Hash#store` expects two

parameters, not an Array, so we use the splat (*) operator to expand the tmp_ary Array down so that it appears to be a list of parameters."

Sorting a Hash by Key or Value

```
my_hash = {'a'=>'1', 'c'=>'3', 'b'=>'2'}
my_hash.keys.sort.each { |key| puts my_hash[key] }
Produces:
1
2
3
```

Hashes are unsorted objects because of the way in which they are stored internally. If you want to access a Hash in a sorted manner by *key*, you need to use an Array as an indexing mechanism as is shown above.

You can also use the Hash#sort method to get a new sorted Array of pairs:

```
my_hash.sort
    #=> [["a", "1"], ["b", "2"], ["c", "3"]]
```

You can do the same by *value*, but it's a little more complicated:

```
my_hash.keys.sort_by { |key| my_hash[key] }.each do
|key|
    puts my_hash[key]
end
```

Or, you can use the Hash#sort method for values:

```
my_hash.sort { |l, r| l[1]<=>r[1] }
    #=> [["a", "1"], ["b", "2"], ["c", "3"]]
```

This works by using the `Emmuerator#sort_by` method that is mixed into the `Array` of `keys. #sort_by` looks at the value `my_hash[key]` returns to determine the sorting order.

Eliminating Duplicate Data from `Arrays` (Sets)

```
[1, 1, 2, 3, 4, 4].uniq
    #=> [1, 2, 3, 4]
```

You can approach this problem in two different ways. If you are adding all your data to your `Array` up front, you can use the expensive way, `#uniq`, above, because you have to do it only once.

But if you will constantly be adding and removing data to your collection and you need to know that all the data is unique at any time, you need something more to guarantee that all your data is unique, but without a lot of cost. A set does just that.

Sets are a wonderful tool: They ensure that the values you have stored are unique. This is accomplished by using a `Hash` for its storage mechanism, which, in turn, generates a unique signifier for any keys it's storing. This guarantees that you won't have the same data in the set while also keeping things accessible and fast! Beware, however, sets are not ordered.

```
require 'set'
myset = Set::new [1, 1, 2, 3, 4, 4]
    #=> #<Set: {1, 2, 3, 4}>
```

Adding duplicate data causes no change:

```
myset.add 4
    #=> #<Set: {1, 2, 3, 4}>
```

Working with Nested Sets

You should be aware that `Set` does not guarantee that nested sets stored in it are unique. This is because `foo_set.eql? bar_set` will never return `true` – even if the sets have exactly the same values in them. Other kinds of objects in Ruby exhibit this behavior, so keep your eyes open.

If you would like to iterate over the contents of sets without having to worry about the nested data possibly colliding with the upper data, you cannot use `Set#flatten`. Here is a simple method to recursively walk through such a set:

```
class Set
    def each_recur(&block)
        each do |elem|
            if elem.is_a? Set
                elem.each_recur(&block)
            else
                block.call(elem)
            end
        end
    end
end

my_set = Set.new.add([1, 2, 3, 4].to_set).add([1, 2,
3, 4].to_set)
    #=> #<Set: {#<Set: {1, 2, 3, 4}>, #<Set: {1, 2,
```

```
3, 4}>}>

my_set.each_recur { |elem| print(elem, " ") }
```

Produces:

```
1 2 3 4 1 2 3 4
```

Index

D

E

F

Q-R

S

T

U-V-W

X-Y-Z