Discover the World Around You Through Programming

Exploring Everyday Things

with R and Ruby

O'REILLY®

Sau Sheong Chang

Exploring Everyday Things with R and Ruby

If you're curious about how things work, this fun and intriguing guide will help you find real answers to everyday problems. By using fundamental math and doing simple programming with the Ruby and R languages, you'll learn how to model a problem and work toward a solution.

All you need is a basic understanding of programming. After a quick introduction to Ruby and R, you'll explore a wide range of questions by learning how to assemble, process, simulate, and analyze the available data. You'll learn to see everyday things in a different perspective through simple programs and common sense logic. Once you finish this book, you can begin your own journey of exploration and discovery.

Here are some of the questions you'll explore:

- Determine how many restroom stalls can accommodate an office with 70 employees
- Mine your email to understand your particular emailing habits
- Use simple audio and video recording devices to calculate your heart rate
- Create an artificial society—and analyze its behavioral patterns to learn how specific factors affect our real society

"A curious hacker's dream! Guides you through geeky projects of pure fun, while learning great R and Ruby skills too. This book captures the true hacker spirit more than any I've seen in years."

—Derek Sivers
founder, CD Baby, sivers.org

Sau Sheong Chang, Director of Applied Research for HP Labs Singapore, has been in software development—mostly cloud- and data-related systems—for more than 17 years. Well known in the local developer communities, he is an active speaker at various technology conferences. He has published two books on Ruby prior to this one.

US $29.99 CAN $31.99
ISBN: 978-1-449-31515-3

Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com

Exploring Everyday Things with R and Ruby

Sau Sheong Chang

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Exploring Everyday Things with R and Ruby

by Sau Sheong Chang

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Andy Oram and Mike Hendrickson
Production Editor: Kristen Borg
Copyeditor: Rachel Monaghan

Proofreader: Kiel Van Horn
Indexer: Angela Howard
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

July 2012: First Edition

Revision History for the First Edition:

2012-06-26 First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449315153* for release details.

ISBN: 978-1-449-31515-3

[LSI]

Table of Contents

Preface

Explorers Ahoy!

It's hard to compare intrepid explorers like Ferdinand Magellan, James Cook, and Roald Amundsen with someone, well, like me. While these adventurers braved the elements, wild nature, and unknown dangers to discover new worlds (at least for their civilization), my biggest physical achievement to date would probably be completing a 10-kilometer charity quarter-marathon—walking.

The explorers of old had it good, of course, when it came to choices of unexplored places to stake their claim on. Christopher Columbus only had to sail due west from Europe, and he discovered two entire continents. For us, there are far fewer choices. There isn't much landmass on Earth that is yet unexplored; even the Mariana Trench, the deepest part of the world's oceans, has been conquered.

But explorer I am, and explorer you will be in this book. While much of the known physical world has been conquered (see Figure P-1), the unknown still looms over most of us.

We are all born with a sense of wonder and amazement at the world around us. Many of us just learn to turn it off as we grow older and jaded. I believe this is partly because we don't understand what goes on in the world around us well enough, and thus we don't care either. Click the remote and the TV turns on—why and how does that work? The first time we tried to ask, we were probably given a blank stare or waved away—who cares as long as you can watch the next season of *American Idol*? That soon grows to be our reaction as well.

Figure P-1. The Scott expedition to the South Pole (photo from the Public Domain Review; http://publicdomainreview.org/2012/03/29/remembering-scott)

Well, in this book, I'll take you along winding paths to bring back the original, wide-eyed person you were. We'll find the magic again, and hopefully at the end of the book, you'll continue where we leave off and make your own way in that journey of exploration and discovery.

Data, Data, Everywhere

We are swamped with data every minute and second of our lives. I don't mean this metaphorically, and I am not simply waxing lyrical about big data either.

In fact, we're so swamped that our eyes have evolved and adapted to this fact by shutting off our environment for a very short while every millisecond. In a phenomenon called *saccadic masking*, the brain shuts down during a fast eye movement (a *saccade*) to remove blurred images that come to our retina. Blurred images are not very useful, so the brain discards them, rendering us effectively blind (without us realizing it) during a saccade.

There is much similarity between saccadic masking and the way we process data today. The data comes so fast, so frequently that we often mask it away. There is a lot of data around us that we can extract and analyze to find answers, but the problem has always been *how* to do this.

In the (distant) past, it was always geniuses who had that knack of unlocking secrets with data and insight, along with the serendipitous few who simply stumbled on the answers. Not so anymore. Although intelligence is still a prerequisite, the arrival of computers and programming has elevated us from the more mundane, repetitive, and mind-numbing tasks of processing data to extract nuggets of information.

Only, it hasn't.

At least not for most people, anyway. The exceptions are scientists and mathematicians, who long ago pounced on the tools that enable them to do their work much more efficiently. If you're someone from these two camps, you are likely already taking full advantage of the power of computers.

However, for programmers and many other people, writing computer programs started with providing tools for businesses and for improving business processes. It's all about using computers to reduce cost, increase revenue, and improve efficiency. For many professional programmers, coding is a job. It's drudgery, low-level menial work that brings food to the table. We have forgotten the promise of computers and the power of programming for discovery.

Bringing the World to Us

This book is an attempt to bring back that wonder and sense of discovery. I want this book to uncover things that you didn't know, or didn't understand. I want it to help you discover new worlds within the existing world we see every day. Finally, I want it to enable you to explore the mundane and learn new things through programming and analyzing data.

While sometimes the world we explore in this book is the real world, more often it's not. It's hard to explore the whole wide world with just bits and bytes. So if we can't explore the world we live in, we'll create our own worlds and explore those—in other words, we'll use *simulations*.

Simulations are an excellent way of exploring things that we cannot control. We do this all the time. When we were young, we often created make-believe worlds and lived in them. Doing this enabled us to understand the real world better. We still do this today, through the magic of television (especially serials and soap operas) and movies—where we live through the characters we see on the screen. And for better or worse, simulations like television affect our real lives and even our dreams. For

example, a survey by the American Psychological Association found that only 20% of people in their 60s (who grew up before color television was popular) recalled having bright and vivid dreams. However, 80% of people under the age of 30 confirmed that their dreams were in full color.[1]

In this book, we will use simulations to create experiments, isolate factors, and propose hypotheses to explain the results of the experiments. You might or might not agree with the experiments I describe or the hypotheses I suggest, but that doesn't really matter. What I would like you to get out of our journey together is the realization that there is more than business as usual to programming business solutions and processes. What I hope to achieve is for you eventually to design your own experiments, run through them, and discover your own worlds.

Packing Your Bags

So what do you need on this journey of discovery, this grand adventure through programming and analyzing data? Tools, of course. They will be the subject of the next two chapters. These are not the only tools available to you, but they are the ones we will be using in this book.

The two tools we will use are Ruby and R. I've chosen them for specific purposes. Ruby is easy to learn and to read, perfectly suited to explain concepts in human-readable code. I will be using Ruby to write simulations and to do preprocessing to get data. R, on the other hand, is great for analyzing data and for generating charts for visualization.

Although you don't need to be a Ruby or R programmer to be able to appreciate this book, I have assumed a basic understanding of programming. Specifically, I assume you have completed a computer science or related course or have done some simple programming in any programming language.

For the rest of the book, every chapter is more or less self-sufficient. Each chapter explores an idea, starting from the realization that a question exists and then attempting to answer it in either a simulation or some processing that brings out the data. We then analyze this data and make certain conclusions based on our analysis.

The ideas are drawn from diverse fields, ranging from economics to evolution, from healthcare to workplace design (in this case, figuring out the correct number of restrooms in an office). Some ideas are grander than others, and some ideas can be quite personal. The reason for this diversity is to show that the possibilities for exploration are limited only by our creativity.

1. Okada, Hitoshi, Kazuo Matsuoka, and Takao Hatakeyama. "Life Span Differences in Color Dreaming." *Dreaming* 21, no. 3 (2011), 213–220.

Each chapter usually starts off small, and we gradually add on layers of complexity to flesh out its central idea. The hypotheses, conclusions, and results from the experiments surrounding the base idea are incidental. You might, for example, agree or disagree with my conclusions and interpretation of the results. For this book at least, the journey is more important than the results.

With that, we're off! Have fun with the next two chapters, and enjoy the rest of the explorations, intrepid explorer!

Conventions Used in This Book

The following typographical conventions are used in this book:

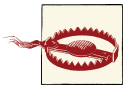
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user; also used for emphasis within program listings.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

All examples and related files in this book may be downloaded from GitHub.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require

permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Exploring Everyday Things with R and Ruby* by Sau Sheong Chang (O'Reilly). Copyright 2012 Sau Sheong Chang, 978-1-449-31515-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreil.ly/everyday-things-r-ruby

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This is the part where I finally get to thank the people who helped me create the book you now hold in your hands. Writing a book is never the sole effort of a lonely author, as I have learned over the years, but the collective work of the author, a professional team, and a community of reviewers and supporters. In no particular order, I would like to thank:

- Mike Hendrickson for agreeing to this rather different type of programming book. It was a wild shot sending in the book proposal and I didn't really expect it to be picked up, except that it was.
- Andy Oram for being patient to a first time O'Reilly author, and arranging really long distance Skype calls halfway around the world, and waking up really early to speak to me every Tuesday evening.
- Kristen Borg, Rachel Monaghan, and the whole production editing team for doing such an awesome and professional job with the book.
- Jeremy Leipzig, Ivan Tan, Patrick Haller, and Judith Myerson for their help in doing the technical reviews and giving great advice. In particular, Patrick Haller, whom I badgered with emails about his comments on my R scripts. Thanks, Patrick!
- Rully Santosa, Chen Way Yen, Ng Tze Yang, Kelvin Teh, George Goh, and the rest of the HP Labs Singapore Applied Research team, to whom I have bounced off countless ideas and have given me innumerable remarks. Special thanks to Rully, Way Yen, and George for their feedback in Chapter 6.

- The Ruby community, especially the Singapore Ruby Brigade, where I made and continue to make good friends with common interests in exploring the world through Ruby. It's a great community to be in, and I relish the (now) annual RedDotRubyConf organized by the ever efficient Andy Croll.

Finally, I would like to dedicate this book to my family, who is my inspiration and my motivation in everything I do. To my lovely wife Wooi Ying, who has been patient yet again (for the third time), thanks for understanding why I simply have to understand everything and how it works. To my soon-to-be teenage son Kai Wen, I hope this book will also be an inspiration to you in being the wide-eyed explorer that I have been all my life.

CHAPTER 1

The Hat and the Whip

Indiana Jones is one of my favorite movie trilogies of all time, and Harrison Ford was a hero to me when I was growing up. Something I always loved about Indy was how he cracked his whip. In fact, I first learned what a bullwhip was watching *Raiders of the Lost Ark*.

The first two movies—*Raiders of the Lost Ark*, and *Indiana Jones and the Temple of Doom*—dealt with Indiana Jones the adult, already fully hardened and cranky. As I watched one movie after another, I wondered about his trademark hat and whip—why the fedora and why on earth a whip?

Finally, all was answered in the third movie of the trilogy, *Indiana Jones and the Last Crusade*. It was one of those satisfying *aha* moments that—although not at all that important in the overall scheme of things—gave Indy an origin, explaining the hat and the whip and why he did what he did.

So what does this have to do with a programming book? Just as the hat and the whip were indispensable tools for Indy, Ruby and R will be our two main tools in the rest of this book. And just as the hat and whip were not conventional tools for archaeology professors doing field work, neither are Ruby and R conventional tools for exploring the world around us. They just make things a whole lot more fun.

Ruby

Each of these tools will need its own chapter. We'll start off first with Ruby and then discuss R in the next chapter. Obviously, there is no way I can explain the entire Ruby programming language in a single chapter of a book, so I will give enough information to whet your appetite and hopefully entice you to proceed to the juicier books that discuss Ruby in more depth.

Why Ruby

One of the first questions you might ask (unless you're a Ruby enthusiast and you already know, in which case you can just nod along) is why did I choose Ruby as one of the two tools used in this book? There are a number of very good reasons. However, there are a couple that I want to focus on, specific to the goals of this book.

First, Ruby is a programming language for human beings. Yukihiro "Matz" Matsumoto, the creator of Ruby, often said that he tried to make Ruby natural, not simple, in a way that mirrors life. Ruby programming is a lot like talking to your good friend, the computer. Ruby was designed to make programming fun and to put the human back into the equation for programming. For example, to print "I love Ruby" 10 times on the screen, simply tell the computer to do exactly that:

```
10.times do
  puts "I love Ruby"
end
```

If you're familiar with C programming and its ilk, like Java, you'll already know that to check whether the variable `a_statement` is true, you need to do something like this (note that in C you will need to use the integer `1` instead of `true`, since C doesn't have a Boolean type):

```
a_statement = true;
if (a_statement == true) {
  do_something();
}
```

While you can certainly do the same in Ruby, it also allows you to do something like this:

```
do_something if a_statement
```

This results in code that is very easy to read and therefore to maintain. While Ruby can have its esoteric moments, it's generally a programming language that can allow someone else to read and understand it easily. As you can imagine, this is a feature that is very useful for this book.

Secondly, Ruby is a dynamic language, and what that means for you as a reader of this book is that you can copy the code from this book, plop it in a file (or the Interactive Ruby shell, as you will see later), and run it directly. There is no messy setting up of makefiles or getting the correct paths for libraries or compiling the compiler before running the examples. Cut, paste, and run—that's all there is to it.

While these are the two primary reasons I used Ruby in this book, if you're keen to understand why many other programmers have turned to Ruby, you can take a look at the Ruby website (*http://www.ruby-lang.org*) or search around the Internet, and you'll find plenty of people gushing over it.

Installing Ruby

Of course, before we can even start using Ruby, we need to get it into our machines. This is generally a simple exercise. There are three main ways of getting Ruby in your platform of choice, depending on how gung-ho you are.

Installing Ruby from source

If you're feeling pretty ambitious, you can try compiling Ruby. This mostly means that you need to have the tools to compile Ruby in your platform, so unless you really want to get serious with Ruby, I suggest that you install it from a precompiled binary, either through a third-party tool or your platform's usual package management tool.

To compile Ruby from source, go to *http://www.ruby-lang.org/en/downloads* and download the source, then compile it using your platform compiler. You can get more information from the same site.

Installing Ruby using third-party tools

Alternatively, you can use one of these popular third-party tools. The recommended approach is to go with the first, which is Ruby Version Manager if you're running on OS X or Linux, and RubyInstaller if you're on Windows.

Ruby Version Manager (RVM). RVM is probably the most popular third-party tool around for non-Windows platforms. A distinct advantage of using RVM is that you will be able to install multiple versions of Ruby and switch to any of them easily. Installing RVM, while not very difficult, is not a single-liner. As of today at least, this is the way to install RVM.

First, you need to have Git and curl installed. Then, issue this command in your console:

```
$ curl -L get.rvm.io | bash -s stable
```

Then, reload your shell by issuing this (or a similar command, depending on your shell):

```
$ source ~/.profile
```

This will allow you to run `rvm`. The next thing you should do is to check whether you have all you need to install Ruby:

```
$ rvm requirements
```

Once you have that, use `rvm` to install the version of Ruby you want. In our case, we'll be using Ruby 1.9.3:

```
$ rvm install 1.9.3
```

After this, check whether the Ruby version you wanted is correctly installed:

```
$ rvm list
```

You should see a list (or at least one) of RVM Rubies installed. If this is your first time installing, there will not be any default Ruby, so you will need to set one by issuing the following command:

```
$ rvm alias create default ruby_version
```

Replace *ruby_version* with the version you've just installed (such as `ruby 1.9.3p125`), and you're done! Check out the RVM website at *https://rvm.io/* for more installation instructions in case you're stuck at any point in time.

RubyInstaller. If you're using Windows, you can't install RVM. In that case, you can either create a virtual machine, install your favorite GNU/Linux distro, and then proceed; or just use RubyInstaller, which is frankly a lot easier. Simply go to *http://rubyinstaller.org/downloads*, download the correct version, and then install it. RubyInstaller includes many native C-based extensions, so that's a bonus. It is a graphical installer, so it's pretty simple to get a fresh installation set up quickly.

Installing Ruby using your platform's package management tool

If none of the approaches listed so far suits you, then you can opt to use your system's package management tool. For Debian systems (and this includes Ubuntu), you can use this command:

```
$ sudo apt-get install ruby1.9.1
```

This will install Ruby 1.9.2. Yes, it's weird.

For Macs, while Ruby comes with OS X, it's usually an older version (Lion comes with Ruby 1.8.7, and the previous versions come with even older versions of Ruby). There is a popular package management tool in OS X named Homebrew, which helps you to replace this with the latest version of Ruby. As you would guess, you'll need to install Homebrew first. Run this command on your console:

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.github.com/gist/323731)"
```

Then install Ruby with this simple command:

```
$ brew install ruby
```

Homebrew is actually just a set of Ruby scripts.

Running Ruby

Once you have installed Ruby with any of the preceding methods, it's time to start using it! Unlike compiled languages such as C, C++, or Java, you don't need to have an intermediate step to generate executable files before running Ruby.

There are a few ways of running Ruby code, but the easiest way to get started is probably using the interactive Ruby tool that's built into your Ruby installation. *irb* is a Ruby REPL (read-eval-print loop) application, an interactive programming environment that allows you to type in Ruby commands and have them evaluated in real time:

```
$ irb
ruby-1.9.3-p125 :001 > puts "hello world!"
hello world!
 => nil
ruby-1.9.3-p125 :002 >
```

Note that once you have typed in a Ruby statement (in this case, we are placing the string "hello world!" to the standard output), the statement is evaluated immediately, resulting in "hello world!" being printed on the screen. After that, *irb* tells you the statement evaluates to nil, because the Ruby `puts` statement returns a nil. If you have put in a statement like this:

```
$ irb
ruby-1.9.3-p125 :001 > 1 + 1
 => 2
ruby-1.9.3-p125 :002 >
```

This statement returns `2`, which is the result of the evaluation. *irb* is a tool you will quickly get used to and will be using whenever you're not sure what the result is going to be.

Another common method of running Ruby is to save your code in a file and then run your file through the Ruby interpreter. For example, you could save `puts "hello world!"` to a file named *hello_world.rb*. After that, you can try this command at the console:

```
$ ruby hello_world.rb
hello world!
```

Most of the examples in this book will be run this way.

Requiring External Libraries

While you can probably get away with writing simpler Ruby programs without any other libraries than the ones built into Ruby itself, most of the time you'll need some external libraries to make life easier. Two sets of Ruby libraries come preinstalled with Ruby.

Core
: This is the default set of classes and modules that comes with Ruby, including `String`, `Array`, and so on.

Standard
: These libraries, found in the */lib* folder of the Ruby source code, are distributed with Ruby but are not included by default when you run it. These include libraries such as Base64, Open URI, and the Net packages (HTTP, IMAP, SMTP, and so on).

To use the standard libraries and any other libraries other than the Ruby core, you will need to *require* them in your program:

```
require 'base64'
```

In addition to the standard libraries, you will often need to use external libraries developed by the Ruby community or yourself. The most common way to distribute Ruby libraries is through RubyGems, the package manager for Ruby. It's distributed as part of Ruby in the standard library, so you can use it out of the box once Ruby is installed.

Just as *apt-get* and *yum* manage packages on a Linux distribution, RubyGems allows you to easily install or remove libraries and Ruby applications. To be distributed through RubyGems, the library or application needs to be packaged in something called a *gem*, which is a package of files to install as well as self-describing metadata about the package.

Gems can be distributed locally (passed around in a *.gem* file) or remotely through a gem server. A few public gem servers provided gem hosting in the past, including RubyForge, GitHub, and GemCutter, but recently they have been more or less replaced by RubyGems. In RubyGems lingo, gem servers are also known as *sources*. You can also deploy a private gem server where you publish private gems that you prepackage for internal use.

To add sources to your RubyGems installation, you can do this:

```
$ gem sources -add http://your.gemserver.org
```

To install a local gem, you can do the following at the console:

```
$ gem install some.gem -local
```

You can do away with the `-local` option, but doing so will add a bit of time because the command will search the remote sources. Setting the local option tells RubyGems to skip that. To add a gem from a remote source, you can generally do this:

```
$ gem install some_gem
```

You can also install specific versions of a gem like so:

```
$ gem install some_gem -version 1.23
```

To list the gems that you have installed locally, you can do this:

```
$ gem list -local
```

Basic Ruby

With the setup complete, let's get started with Ruby!

Strings

Manipulating strings is one of the most basic things you normally do in a program. Any programming language worth its salt has a number of ways to manipulate strings, and Ruby is no exception. In fact, Ruby has an embarrassment of riches in terms of its capability to manipulate strings.

Ruby strings are simply sequences of characters. There are a few ways of defining strings. The most common ways are probably through the single(') and double(") quotes. If you define a string with double quotes, you can use escape sequences in the string and also perform substitution of Ruby code into the string using the expression `#{}`. You can't do this inside single-quoted strings:

```
"There are #{24 * 60 * 60} seconds in a day"
=> "There are 86400 seconds in a day"

'This is also a string'
=> "This is also a string"
```

Strings can also be defined using `%q` and `%Q`. `%q` is the same as single-quoted strings, and `%Q` is the same as double-quoted strings, except that in these cases the delimiters can be anything that follows `%q` or `%Q`:

```
%q/This is a string/
=> "This is a string"

%q{This is another string}
=> "This is another string"

%Q!#{'Ho! ' * 3} Merry Christmas\!!
=>"Ho! Ho! Ho!  Merry Christmas!"
```

Finally, you can also define a string using a *here-document*. A here-document is a way of specifying a string in command-line shells (sh, csh, ksh, bash, and so on) and in programming or scripting languages such as Perl, PHP, Python, and, of course, Ruby. A here-document preserves the line breaks and other whitespace (including indentation) in the text:

```
string = <<END_OF_STRING
   The quick brown fox jumps
   over the lazy dog.
END_OF_STRING
=> "   The quick brown fox jumps\n   over the lazy dog.\n"
```

Take note that the delimiter is the string after the `<<` characters—in this case, `END_OF_STRING`.

Although I can't list everything that Ruby provides for string manipulation in this section, here are a few things it can do:

```
a = "hello "
b = "world"

a + b
=> "hello world"                  # string concatenation (this adds b to a
                                  # to create a new string)

a << b
=> "hello world"                  # append to string (this modifies a)

a * 3
=> "hello hello hello"            # you can repeat strings by simply
                                  # multiplying them

c = "This is a string"            # splitting a string according to a delimiter,
                                  # any space being the default delimiter
c.split
=> ["This", "is", "a", "string"]
```

Arrays and hashes

Just as important as strings, and perhaps sometimes even more so, is being able to manipulate data structures. The two most important data structures, which you'll meet very often in this book (and also in Ruby programming), are arrays and hashes.

Arrays are indexed containers that hold a sequence of objects. You can create arrays using square brackets (`[]`) or using the `Array` class. Arrays are indexed through a running integer starting with 0, using the `[]` operator:

```
a = [1, 2, 'this', 'is', 3.45]
a[0]  #  1
a[1]  #  2
a[2]  # "this"
```

There are other ways of indexing arrays, including the use of ranges:

```
a[1..3]  # [2. 'this', 'is']
```

You can also set items in the array using the same operator:

```
a[4] = 'an'
a   # [1, 2, 'this', 'is', 'an']
```

Arrays can contain anything, including other arrays:

```
a[5] = ['another', 'array']
a   # [1, 2, 'this', 'is', 'an', ['another', 'array']]
```

If you're used to manipulating data structures, you might be wondering why I'm discussing only arrays and hashes in this section. What about the other common data structures, like stacks, queues, sets, and so on? Well, arrays can be used for them as well:

```
stack = []
stack.push 1
stack.push 2
stack.push 'hello'
stack   # [1, 2, 'hello']

stack.pop   # 'hello'
stack   # [1, 2]
```

Tons of other methods can be used on arrays; you can find them through the reference documentation on the Ruby website, or even better, by firing up *irb* and playing around with it a bit. A common way of iterating through arrays is using the `each` method:

```
a = ['This', 'is', 'an', 'array']

a.each do |item|
  puts item
end
```

This will result in each item in the array being printed out at the standard output (i.e., the console). In the preceding code, the loop starts with `do` and ends with `end`. It runs for each of the four items in the array; here, we chose the variable `item` to represent the item within the loop. We use vertical bars to surround the variable name `item`. Sometimes, for brevity, we can replace the `do` … `end` with a pair of curly braces {}. This code produces the following results:

```
This
is
an
array
```

Notice that the items in the array are printed in the same sequence in which they are defined.

While arrays have a lot of methods, you should also be aware that `Array` inherits from the `Enumerable` module, so it also implements those methods. We'll get to `Enumerable` shortly.

Hashes are dictionaries or maps, data structures that index groups of objects. The main difference is that instead of having an integer index, hash indices can be any object. Hashes are defined using curly braces {} or the `Hash` class, and indexed using square brackets:

```
h = { 'a' => 'this', 'b' => 'is', 'c' => 'hash'}

h['a']      # "this"
h['b']      # "is"
h['c']      # "hash"
```

Setting an item in a hash also uses the square brackets:

```
h['some'] = 'value'
h   # { 'a' => 'this', 'b' => 'is', 'c' => 'hash', 'some' => 'value'}
```

The *hash rocket* style of assigning values to keys in hashes was changed in Ruby 1.9. While that still works, the new syntax is simpler and more crisp. The following lines of code do exactly the same thing:

```
h = { canon: 'camera', nikon: 'camera', iphone: 'phone'}
# is the same as
h = { :canon => 'camera', :nikon => 'camera', :iphone => 'phone'}
```

There are many ways of iterating through hashes, but here's a common way of doing it:

```
h = { canon: 'camera', nikon: 'camera', iphone: 'phone'}

h.each do |key, value|
  puts "#{key} is a #{value}"
end
```

Just as we used vertical bars earlier to name `item` as the variable to represent items from an array, here we use vertical bars to name two variables. The first represents each key in the hash, and the second represents its associated value. This code produces the following results:

```
canon is a camera
nikon is a camera
iphone is a phone
```

Both `Array` and `Hash` inherit from—that is, are subclasses of—`Enumerable`. `Enumerable` is a module that provides collection classes with a number of capabilities, including several traversal and searching methods, and the ability to sort. A very useful method (we'll get to methods in a bit) in `Enumerable` is the `map` method, which runs through each item in the collection, performs the action given by the block, and then returns a new array with the new values. The input to `map` in the following example is a range of digits (1, 2, 3, and 4), and its output is the square of each input:

```
(1..4).map do |i|
  i*i
end   #[1, 4, 9, 16]
```

The `max_by` and `min_by` methods are also useful. These, as you might have guessed, return the maximum or minimum item in the array: