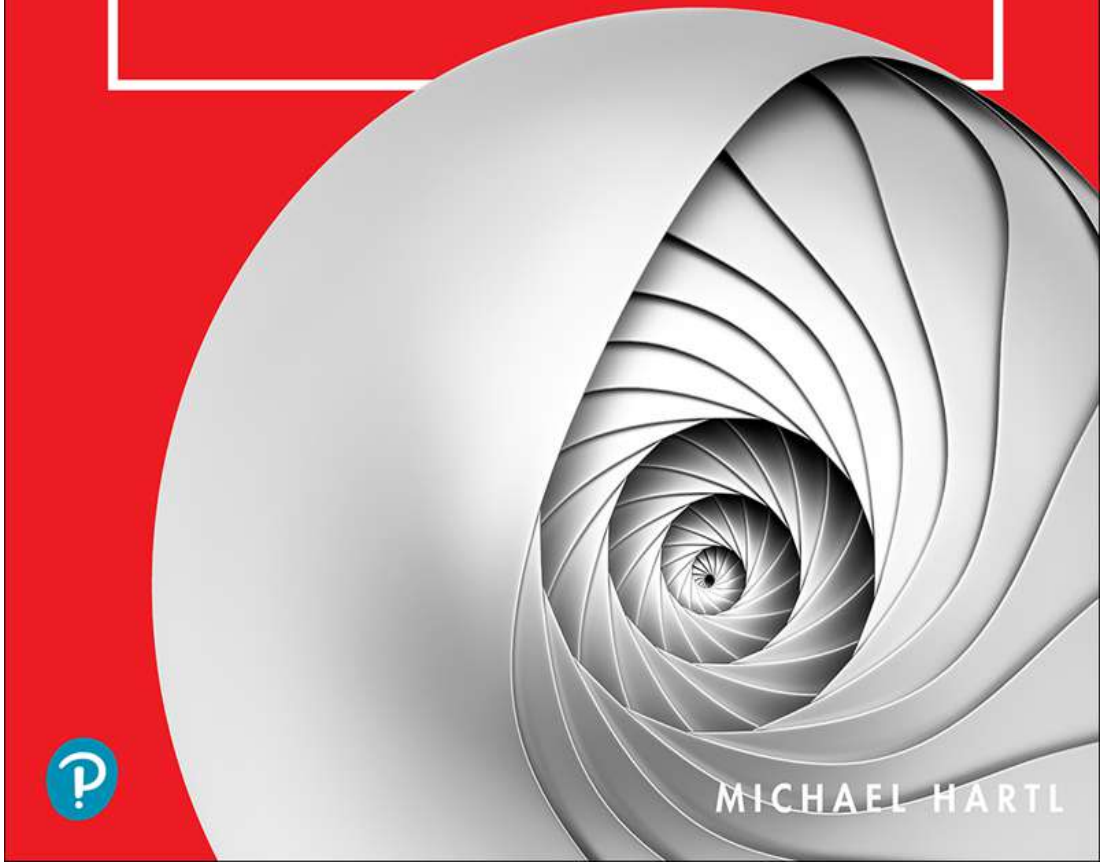
learn enough RUBY TO BE DANGEROUS

MICHAEL HARTL

learn enough

RUBY

TO BE DANGEROUS



MICHAEL HARTL

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Praise for Learn Enough Tutorials

“Just started the #100DaysOfCode journey. Today marks day 1. I have completed @mhartl’s great Ruby tutorial at @LearnEnough and am looking forward to starting on Ruby on Rails from tomorrow. Onwards and upwards.”

—Optimize Prime (@_optimize), Twitter post

“Ruby and Sinatra and Heroku, oh my! Almost done with this live web application. It may be a simple palindrome app, but it’s also simply exciting! #100DaysOfCode #ruby @LearnEnough #ABC #AlwaysBeCoding #sinatra #heroku”

—Tonia Del Priore (@toninjaa), Twitter post

Software Engineer for a FinTech Startup for 3+ years

“I must say, this Learn Enough series is a masterpiece of education. Thank you for this incredible work!”

—Michael King

“I want to thank you for the amazing job you have done with the tutorials. They are likely the best tutorials I have ever read.”

—Pedro Iatzky

Learn Enough Series from Michael Hartl

Visit **informit.com/learn-enough** for a complete list of available publications.

The **Learn Enough** series teaches you the developer tools, Web technologies, and programming skills needed to launch your own applications, get a job as a programmer, and maybe even start a company of your own. Along the way, you'll learn technical sophistication, which is the ability to solve technical problems yourself. And Learn Enough always focuses on the most important parts of each subject, so you don't have to learn everything to get started—you just have to learn enough to be dangerous. The Learn Enough series includes books and video courses so you get to choose the learning style that works best for you.

Learn Enough Ruby to Be Dangerous

Write Programs, Publish Gems, and Develop Sinatra Web Apps with Ruby

Michael Hartl

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover image: Philipp Tur/Shutterstock
Figures 1.4, 5.3, 10.1, 10.2, 10.15: Sinatra
Figures 1.5, 9.2: Amazon.com, Inc.
Figures 1.6, 4.3, 5.2, 5.6, 8.2, 8.5: Apple Inc.
Figures 1.7, 1.8, 8.3: GitHub, Inc.
Figures 1.9, 1.10, 10.3, 10.22: Salesforce, Inc.
Figure 2.9: Ruby-doc.org
Figures 2.11, 3.1, 3.2, 6.4: Courtesy of Mike Vanier
Figures 4.4, 4.5, 4.10, 7.12, 8.7: Michael Lovitt
Figures 5.5, 9.4: Google
Figure 7.10: Courtesy of David Heinemeier Hansson
Figures 9.3, 9.6: Wikimedia Foundation, Inc

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

eBook ISBN-13: 978-0-13-784418-0
eBook ISBN-10: 0-13-784418-2

ePub ISBN-13: 978-0-13-784411-1
ePub ISBN-10: 0-13-784411-5

Release 3, September 2022

Contents

Preface

Learn Enough Ruby to Be Dangerous teaches you to write practical and modern programs using the elegant and powerful Ruby programming language. You'll learn how to use Ruby for both general-purpose programming and for beginning web-application development. Although mastering Ruby can be a long journey, you don't have to learn everything to get started… you just have to learn enough to be *dangerous*.

You'll begin by exploring the core concepts of Ruby programming using a combination of interactive Ruby and text files run at the command line. The result is a solid understanding of both *object-oriented programming* and *functional programming* in Ruby. You'll then build on this foundation to develop and publish a simple self-contained Ruby package, or Ruby *gem*. You'll then use this gem in a simple dynamic web application built using the *Sinatra* web framework, which you'll also deploy to the live Web. As a result, *Learn Enough Ruby to Be Dangerous* is especially appropriate as a prerequisite to the *Ruby on Rails Tutorial*, a bestselling web-development tutorial by the same author.

In addition to teaching you specific skills, *Learn Enough Ruby to Be Dangerous* also helps you develop *technical sophistication*—the seemingly magical ability to solve practically any technical problem. Technical sophistication includes concrete skills like version control and coding, as well as fuzzier skills like Googling the error message and knowing when to just reboot the darn thing. Throughout *Learn Enough Ruby to Be Dangerous*, we'll have abundant opportunities to develop technical sophistication in the context of real-world examples.

Chapter by Chapter

In order to learn enough Ruby to be dangerous, we'll begin at the beginning with a series of simple "hello, world" programs using several different techniques (Chapter 1), including an introduction to *irb*, an interactive command-line program for evaluating Ruby code. In line with the Learn Enough philosophy of always doing things "for real", even as early as Chapter 1 we'll deploy a (very simple) dynamic Ruby application to the live Web.

After mastering "hello, world", we'll take a tour of some Ruby *objects*, including strings (Chapter 2), arrays (Chapter 3), and other native objects like dates, hashes, and regular expressions (Chapter 4). Taken together, these chapters constitute a gentle introduction to *object-oriented programming* with Ruby.

In Chapter 5, we'll learn the basics of *functions*, an essential subject for virtually every programming language. We'll then apply this knowledge to an elegant and powerful style of coding known as *functional programming* (Chapter 6).

Having covered the basics of built-in Ruby objects, in Chapter 7 we'll learn how to make objects of our own. In particular, we'll define an object for a *phrase*, and then develop a method for determining whether or not the phrase is a *palindrome* (the same read forward and backward).

Our initial palindrome implementation will be rather rudimentary, but we'll extend it in Chapter 8 using a powerful technique called *test-driven development* (TDD). In the process, we'll learn more about testing generally, as well as how to create a *Ruby gem*.

In Chapter 9, we'll learn how to write nontrivial *shell scripts*, one of Ruby's biggest strengths. Examples include reading from both files and URLs, with a final example showing how to manipulate a downloaded file as if it were an HTML web page.

In Chapter 10, we'll develop our first full Ruby web application: a site for detecting palindromes. This will give us a chance to learn about *routes*, *layouts*, *embedded Ruby*, and *form handling*, together with a second application of TDD. As a capstone to our work, we'll deploy our palindrome detector to the live Web.

Additional Features

In addition to the main tutorial material, *Learn Enough Ruby to Be Dangerous* includes a large number of exercises to help you test your understanding and to extend the material in the main text. The exercises include frequent hints and often include the expected answers, with community solutions available by separate subscription at www.learnenough.com.

Final Thoughts

Learn Enough Ruby to Be Dangerous gives you a practical introduction to the fundamentals of Ruby, both as a general-purpose programming language and as a web-development specialist. After learning the techniques covered in this tutorial, and especially after developing your technical sophistication, you'll know everything you need to write shell scripts, publish Ruby gems, and deploy dynamic web applications with Ruby. You'll also be ready for a huge variety of other resources, including books, blog posts, and online documentation. A particularly good next step is learning how to make dynamic database-backed web applications with the *Ruby on Rails Tutorial*.

Learn Enough Scholarships

Learn Enough is committed to making a technical education available to as wide a variety of people as possible. As part of this commitment, in 2016 we created the Learn Enough Scholarship program.[1] Scholarship recipients get free or deeply discounted access to the Learn Enough All Access subscription, which includes all of the Learn Enough online book content, embedded videos, exercises, and community exercise answers.

[1] https://www.learnenough.com/scholarship

As noted in a 2019 RailsConf Lightning Talk,[2] the Learn Enough Scholarship application process is incredibly simple: just fill out a confidential text area telling us a little about your situation. The scholarship criteria are generous and flexible—we understand that there are an enormous number of reasons for wanting a scholarship, from being a student, to being between jobs, to living in a country with an unfavorable exchange rate against the U.S. dollar. Chances are that, if you feel like you've got a good reason, we'll think so, too.

[2] https://www.learnenough.com/scholarship-talk

So far, Learn Enough has awarded more than 2,500 scholarships to aspiring developers around the country and around the world. To apply, visit the Learn Enough Scholarship page at www.learnenough.com/scholarship. Maybe the next scholarship recipient could be you!

About the Author

Michael Hartl is the creator of the *Ruby on Rails Tutorial*, one of the leading introductions to web development, and is cofounder and principal author at Learn Enough. Previously, he was a physics instructor at the California Institute of Technology (Caltech), where he received a Lifetime Achievement Award for Excellence in Teaching. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Chapter 1

Hello, World!

"Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write." Or so says the official Ruby website. In my experience, this description rings true—as does the declaration by Ruby's creator that the language is "optimized for programmer happiness." Ruby feels natural to read and write, has a wealth of built-in libraries, and has a powerful object-oriented design.

Learn Enough Ruby to Be Dangerous is designed to get you started writing practical and modern Ruby programs as fast as possible, with a focus on the real tools used every day by software developers.

As a general-purpose programming language, Ruby is limited only by the developer's imagination. Ruby has enjoyed especially robust adoption in *web development*, the art of writing dynamic web applications for the World Wide Web (indeed, your trusty author is perhaps best known in the tech world as the author of the *Ruby on Rails Tutorial*, an introduction to web development using Ruby on Rails). But Ruby thrives in many other niches as well, such as shell scripting, text parsing, and package management.

The present tutorial can serve as either a prerequisite to the *Ruby on Rails Tutorial* (especially for those who haven't programmed much before) or as a natural follow-on (for those who want to solidify their command of the underlying Ruby language). In particular, *Learn Enough Ruby to Be Dangerous* includes an introduction to web development with *Sinatra*, a small and relatively simple Ruby web framework that is excellent preparation for Ruby on Rails while also being a useful tool in its own right.

As noted above, there's more to Ruby than web development, though, and we'll be treating Ruby as a general-purpose programming language right from the start. The result is a practical narrative introduction to Ruby—a perfect complement both to in-browser coding tutorials and to the voluminous but hard-to-navigate Ruby reference material on the Web.

Learn Enough Ruby to Be Dangerous broadly follows the structure of *Learn Enough JavaScript to Be Dangerous*, which can be studied either before or after this tutorial. Because many of the examples are the same, the tutorials reinforce each other nicely—there are few things more instructive in computer

programming than seeing the same basic problems solved in two different languages.

You won't learn everything there is to know about Ruby in this tutorial— that would take thousands of pages and centuries of effort—but you will learn enough Ruby to be *dangerous* (Figure 1.1).[1]

[1] Image courtesy of Kirk Fisher/Shutterstock.

Figure 1.1: Ruby knowledge, like Rome, wasn't built in a day.

There are no programming prerequisites for *Learn Enough Ruby to Be Dangerous*, although it certainly won't hurt if you've programmed before (and suggested shortcuts for experienced devs appear in Box 1.2 below). What is important is that you've started developing your *technical sophistication* (Box 1.1), either on your own or using the preceding Learn Enough tutorials. These tutorials include the following, which together make a good list of prerequisites for this book:

1. *Learn Enough Command Line to Be Dangerous*
2. *Learn Enough Text Editor to Be Dangerous*
3. *Learn Enough Git to Be Dangerous*
4. *Learn Enough HTML to Be Dangerous*
5. *Learn Enough CSS & Layout to Be Dangerous* (optional)
6. *Learn Enough JavaScript to Be Dangerous* (optional)

All of these tutorials are available for individual purchase, and we offer a subscription service—the Learn Enough All Access subscription—with access to all the corresponding online courses.

Box 1.1. Technical Sophistication

An essential aspect of using computers is the ability to figure things out and troubleshoot on your own, a skill we at Learn Enough call *technical sophistication*.

Developing technical sophistication means not only following systematic tutorials like *Learn Enough Ruby to Be Dangerous*, but also knowing when it's time to break free of a structured presentation and just start Googling around for a solution.

Learn Enough Ruby to Be Dangerous will give us ample opportunity to practice this essential technical skill.

In particular, as alluded to above, there is a wealth of Ruby reference material on the Web, but it can be hard to use unless you already basically know what you're doing. One goal of this tutorial is to be the key that unlocks the documentation. This will include lots of pointers to the official Ruby site.

Especially as the exposition gets more advanced, I'll also sometimes include the web searches you could use to figure out how to accomplish the particular task at hand. For example, how do you use Ruby to manipulate a Document Object Model (DOM)? Like this: ruby dom manipulation.

In order to learn enough Ruby to be dangerous, we'll begin at the beginning with a series of simple "hello, world" programs using several different techniques (Chapter 1), including an introduction to *irb*, an interactive command-line program for evaluating Ruby code. In line with the Learn Enough philosophy of always doing things "for real", even as early as Chapter 1 we'll deploy a (very simple) dynamic Ruby application to the live Web.

After mastering “hello, world”, we’ll take a tour of some Ruby *objects*, including strings (Chapter 2), arrays (Chapter 3), and other native objects (Chapter 4). Taken together, these chapters constitute a gentle introduction to *object-oriented programming* with Ruby.

In Chapter 5, we’ll learn the basics of *functions*, an essential subject for virtually every programming language. We’ll then apply this knowledge to an elegant and powerful style of coding called *functional programming* (Chapter 6).

Having covered the basics of built-in Ruby objects, in Chapter 7 we’ll learn how to make objects of our own. In particular, we’ll define an object for a *phrase*, and then develop a method for determining whether or not the phrase is a *palindrome* (the same read forward and backward).

Our initial palindrome implementation will be rather rudimentary, but we’ll extend it in Chapter 8 using a powerful technique called *test-driven development* (TDD). In the process, we’ll learn more about testing generally, as well as how to create a self-contained Ruby library called a *Ruby gem* (and thereby join the large and growing ecosystem of software packages managed by Ruby’s gem-hosting service, *RubyGems.org*).

In Chapter 9, we’ll learn how to write nontrivial *shell scripts*, one of Ruby’s biggest strengths. Examples include reading from both files and URLs, with a final example showing how to manipulate a downloaded file as if it were an HTML web page.

In Chapter 10, we’ll develop our first full Ruby web application: a site for detecting palindromes. This will give us a chance to learn about *routes*, *layouts*, *embedded Ruby*, and *form handling*. As a capstone to our work, we’ll deploy our palindrome detector to the live Web.

In most cases, typing in code examples by hand is the most effective way to learn, but sometimes copying and pasting is more practical. To make the latter more convenient, all code listings from this book are available online at the following URL:

Click here to view code image

```
https://github.com/learnenough/learn_enough_ruby_code_listings
```

Finally, experienced developers can largely skip the first four chapters, as described in Box 1.2.

Box 1.2. For Experienced Devs

By keeping a few diffs in mind, experienced developers can skip Chapters 1–4 of this tutorial and start with functions in Chapter 5. They can then move quickly onto functional programming in Chapter 6, consulting earlier chapters as necessary to fill in any gaps.

Here are some of the notable differences between Ruby and most other languages:

- Use `#!/usr/bin/env ruby` for the shebang line in shell scripts (Section 1.4).
- Use `#{...}` for string interpolation (Section 2.2).
- Single-quoted strings are raw strings (Section 2.2.1).
- Use `puts` for printing (Section 2.3).
- Use `elsif` for `else if` (Section 2.4).
- In a boolean context, all objects are `true` except `nil` and `false` itself— even `""`, `[]`, and `0` (Section 2.4.2).
- Boolean methods end in a question mark, as in `"".empty?` (Section 2.5).
- Iterate using `each` (Section 3.5).
- Math operations are attached to a `Math` object; e.g., `Math.sqrt(2)` (Section 4.1.1).
- Hash keys are often *symbols*, a data type for labels (Section 4.4.1).

1.1 Introduction to Ruby

Created by Yukihiro "Matz" Matsumoto (Figure 1.2),[2] Ruby was originally designed as an object-oriented scripting language—that is, a language based on *objects* that's good for writing *shell scripts*. (We'll learn more about objects starting in Chapter 2, and we'll cover shell scripts in Chapter 9.) The name *Ruby* is a reference (in part) to the *Perl* programming language, which (along with Smalltalk and Lisp) is one of Ruby's principal design influences.

[2] Image copyright © 2012 by Michael Hartl.

Figure 1.2: Yukihiro “Matz” Matsumoto, the creator of Ruby, with author Michael Hartl at RubyConf 2012.

In order to give you the best broad-range introduction to programming with Ruby, *Learn Enough Ruby to Be Dangerous* uses four main methods:

1. An interactive prompt with a Read-Evaluate-Print Loop (REPL)
2. Standalone Ruby files
3. Shell scripts (as introduced in *Learn Enough Text Editor to Be Dangerous*)
4. Ruby web applications running in a web server

We'll begin our study of Ruby with four variations on the time-honored theme of a "hello, world" program, a tradition that dates back to the early days of the C programming language. The main purpose of "hello, world" is to confirm that our system is correctly configured to execute a simple program that prints the string `hello, world!` (or some close variant) to the screen. By design, the program is simple, allowing us to focus on the challenge of getting the program to run in the first place.

Since the original application of Ruby was to write shell scripts for execution at the command line, we'll start by writing a series of programs to display a greeting in a command-line terminal: first in a REPL called *interactive Ruby*, or *irb*; then in a standalone file called `hello.rb`; and finally in an executable shell script called `hello`. We'll then write (and deploy!) a simple proof-of-concept web application using the Sinatra web framework.

(Throughout what follows, I'll assume that you have access to a Unix-compatible system like macOS, Linux, or the Cloud9 IDE, as described in the free tutorial *Learn Enough Dev Environment to Be Dangerous*. If you use the cloud IDE, I recommend creating a development environment called `ruby-tutorial`, and be sure to choose the Ubuntu Server option as shown in Figure 4 of *Learn Enough Dev Environment to Be Dangerous*. For Mac users, although it shouldn't matter in *Learn Enough Ruby to Be Dangerous*, it is recommended that you use the Bourne-again shell (Bash) rather than the default Z shell to complete this tutorial. To switch your shell to Bash, run `chsh -s /bin/bash` at the command line, enter your password, and restart your terminal program. Any resulting alert messages are safe to ignore. See the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials" for more information.)

You can check to see if Ruby is already installed by running `ruby -v` at the command line to get the version number (Listing 1.1).

Listing 1.1: Checking the Ruby version.

Click here to view code image

```
$ ruby -v
ruby 3.1.1p18 (2022-02-18 revision 53f5fc4236) [x86_64-linux]
```

This standardizes on Ruby 3, but any version of Ruby later than 2.7 should be fine for this tutorial. If instead you get a result like

Click here to view code image

```
$ ruby -v
-bash: ruby: command not found
```

or you get a version number earlier than 2.7, then you will have to install a more recent version of Ruby.

The details of installing Ruby vary by system and can require applying a little technical sophistication (Box 1.1). The different possibilities are covered in *Learn Enough Dev Environment to Be Dangerous*, which you should take a look at now if you don't already have Ruby on your system. In particular, if you end up using the cloud IDE recommended by *Learn Enough Dev Environment to Be Dangerous*, you can update the Ruby version as follows:

Click here to view code image

```
$  # on cloud IDE
$ rvm get stable
$ rvm install 3.1.1
$ rvm --default use 3.1.1
```

(This uses Ruby Version Manager, which comes preinstalled on the cloud IDE.) Once that command is finished, you can verify the Ruby version as follows:

Click here to view code image

```
$ ruby -v
ruby 3.1.1p18 (2022-02-18 revision 53f5fc4236) [x86_64-linux]
```

(Exact version numbers may differ.)

1.2 Ruby in a REPL

Our first example of a "hello, world" program involves a Read-Eval-Print Loop, or *REPL* (pronounced "repple"). A REPL is a program that **read**s input, **eval**-uates it, **print**s out the result (if any), and then **loop**s back to the read step. Most modern programming languages provide a REPL, and Ruby is no exception. In Ruby's case, it's called *irb*, short for "interactive Ruby", and we can run it at the command line as shown in Listing 1.2.

Listing 1.2: Bringing up the irb prompt at the command line.

```
$ irb
>>
```

Here `>>` represents a generic irb prompt, which you can achieve on your system by editing a special configuration file called `.irbrc`. Start by creating `.irbrc` in your home directory using the text editor of your choice:[3]

[3] I generally use Sublime Text or Atom for everyday editing, but for editing short configuration files and the like I usually use Vim. The reason is that Vim is incredibly fast to open and quit, which is especially convenient when the editing task itself takes only a few seconds.

```
$ vim ~/.irbrc
```

Then fill the file with the contents of Listing 1.3. This arranges to simplify the irb prompt as in Listing 1.2 while suppressing some annoying auto-indent behavior.

Listing 1.3: Adding some irb configuration.
~/.irbrc

Click here to view code image

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
IRB.conf[:AUTO_INDENT_MODE] = false
```

To apply this configuration, you should exit `irb` using `exit` or `Ctrl-D` and then rerun the `irb` command.

With that bit of configuration done, we're now ready to write our first Ruby program using the `puts` command (pronounced "put-ess"), which stands for "put string", as seen in Listing 1.4. (We'll start learning about strings in Chapter 2.)

Listing 1.4: A "hello, world" program in the REPL.

Click here to view code image

```
>> puts "hello, world!"
hello, world!
=> nil
```

That's it! That's how easy it is to print "hello, world!" interactively with Ruby.

If you're familiar with other programming languages (such as JavaScript), Listing 1.4 is notable for its lack of both parentheses and a terminating semicolon. Cleaning up punctuation in this way is very "Rubyish", i.e., characteristic of Ruby.

You might also note that the final line in Listing 1.4 includes a *return value*, which for `puts` is `nil`, a special Ruby value that means "nothing at all". We'll learn more about `nil` starting in Section 2.3.

1.2.1 Exercises

1. What happens if you use `print` in place of `puts`? How would you change `print`'s argument to get the result to match Listing 1.4? *Hint*: Recall that `\n` is the typical way to represent a newline character.

1.3 Ruby in a File

As convenient as it is to be able to explore Ruby interactively, most Real Programming® takes place in text files created with a text editor. In this section, we'll show how to create and execute a Ruby file with the same "hello, world" program we discussed in Section 1.2. The result will be a simplified prototype of the reusable Ruby files we'll start learning about in Section 5.2.

We'll start by creating a directory for this tutorial and a Ruby file (with a `.rb` file extension) for our `hello` program (be sure to exit irb first if you're still in the REPL):

Click here to view code image

```
$ cd  # Change to the home directory; use cd ~/environment on the cloud IDE.
$ mkdir -p repos/ruby_tutorial
$ cd repos/ruby_tutorial
$ touch hello.rb
```

Here the `-p` option to `mkdir` arranges to create intermediate directories if necessary. *Note*: Throughout this tutorial, if you're using the cloud IDE recommended in *Learn Enough Dev Environment to Be Dangerous*, you should replace the home directory `~` with the directory `~/environment`.

Next, using our favorite text editor, we'll fill the file with the contents shown in Listing 1.5. Note that the code is exactly the same as in Listing 1.4, with the difference that in a Ruby file there's no command prompt `>>`.

Listing 1.5: A “hello, world” program in a Ruby file.
hello.rb

```
puts  "hello, world!"
```

At this point, we’re ready to execute our program using the `ruby` command we used in Listing 1.1 to check the Ruby version number. The only difference is that this time we omit the `-v` flag and instead include an argument with the name of our file:

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

As in Listing 1.4, the result is to print “hello, world!” to the terminal screen, only now it’s the raw shell instead of an irb REPL.

Although this example is simple, it’s a huge step forward, as we’re now in the position to write Ruby programs much longer than could comfortably fit in an irb session.

1.3.1 Exercises

1. What happens if you give `puts` two arguments, as in Listing 1.6?

Listing 1.6: Using two arguments.
hello.rb

Click here to view code image

```
puts "hello, world!", "how's it going?"
```

1.4 Ruby in a Shell Script

Although the code in Section 1.3 is perfectly functional, when writing a program to be executed in the command line shell it’s often better to use an *executable script* of the sort discussed in *Learn Enough Text Editor to Be Dangerous*. Indeed, as noted in Section 1.1, shell scripting was Ruby’s original programming niche.

Let’s see how to make an executable script using Ruby. We’ll start by creating a file called `hello`:

```
$ touch hello
```

Note that we *didn't* include the `.rb` extension—this is because the filename itself is the user interface, and there's no reason to expose the implementation language to the user. Indeed, there's a reason not to: By using the name `hello`, we give ourselves the option to rewrite our script in a different language down the line, without changing the command our program's users have to type. (Not that it matters in this simple case, but the principle should be clear. We'll see a more realistic example in Section 9.3.)

There are two steps to writing a working script. The first is to use the same command we've seen before (Listing 1.5), preceded by a "shebang" line telling our system to use `ruby` to execute the script.

Ordinarily, the exact shebang line is system-dependent (as seen with Bash in *Learn Enough Text Editor to Be Dangerous* and with JavaScript in *Learn Enough JavaScript to Be Dangerous*), but with Ruby we can ask the shell itself to supply the proper command. The trick is to use the `ruby` executable available as part of the shell's *environment* (env):

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

Using this for the shebang line gives the shell script shown in Listing 1.7.

Listing 1.7: A "hello, world" shell script.
hello

Click here to view code image

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

puts "hello, world!"
```

We could execute this file directly using the `ruby` command as in Section 1.3, but a true shell script should be executable without the use of an auxiliary program. (That's what the shebang line is for.) Instead, we'll follow the second of the two steps mentioned above and make the file itself executable using the `chmod` ("change mode") command combined with `+x` ("plus executable"):

```
$ chmod +x hello
```

At this point, the file should be executable, and we can execute it by preceding the command with `./`, which tells our system to look in the current directory (dot = `.`) for the executable file. (Putting the `hello` script on the PATH, so that it can be called from any directory, is left as an exercise.) The result looks like this:

```
$ ./hello
hello, world!
```

Success! We've now written a working Ruby shell script suitable for extension and elaboration. As mentioned briefly above, we'll see an example of a real-life utility script in Section 9.3.

Throughout the rest of this tutorial, we'll mainly use irb for initial investigations, but the eventual goal will almost always be to create a file (either pure code or HTML) containing Ruby.

1.4.1 Exercises

1. By moving the file or changing your system's configuration, add the `hello` script to your environment's PATH. (You may find the steps in *Learn Enough Text Editor to Be Dangerous* helpful.) Confirm that you can run `hello` without prepending `./` to the command name. *Note*: If you have a conflicting `hello` program from following *Learn Enough JavaScript to Be Dangerous*, I suggest replacing it—thus demonstrating the principle that the file's name is the user interface, and the implementation can change language without affecting users.

1.5 Ruby in a Web Browser

Although originally designed for shell scripting, Ruby's flexibility and expressiveness led Danish programmer David Heinemeier Hansson (often known as "DHH" for short) to choose it to implement a project-management application called Basecamp. From Basecamp, DHH extracted a general-purpose framework for making dynamic web applications, which he named Ruby on Rails (a satirical reference to a heavyweight Java framework called "Struts"). Due in large part to the success of the Rails framework, Ruby has since become a major player in web development. In recognition of this, our final example of a "hello, world" program will be a live web application, written in the simple but powerful *Sinatra* micro-framework (Figure 1.3).[4]

[4] Image courtesy of UtCon Collection/Alamy Stock Photo.

Figure 1.3: Frank Sinatra, notable both for his mellifluous singing voice and his astonishing skill at web development.

We'll begin by installing a couple of self-contained pieces of Ruby software, known as *Ruby gems*. First, we'll do a little preparation by adding some configuration settings to prevent the time-consuming installation of local Ruby documentation, which needs to be done only once per system (definitely don't worry about trying to understand this command):

Click here to view code image

```
$ echo "gem: --no-document" >> ~/.gemrc
```

Now we're ready to install the `sinatra` gem (together with a web server called `puma`) using the `gem` command, which is installed automatically as part of Ruby:

```
$ gem install sinatra -v 2.2.2
$ gem install puma    -v 5.6.5
```

Believe it or not, those two commands install all of the software needed to run a simple but full-strength web application on our local system (where "local" might refer to the cloud if you're using the cloud IDE recommended in *Learn Enough Dev Environment to Be Dangerous*).

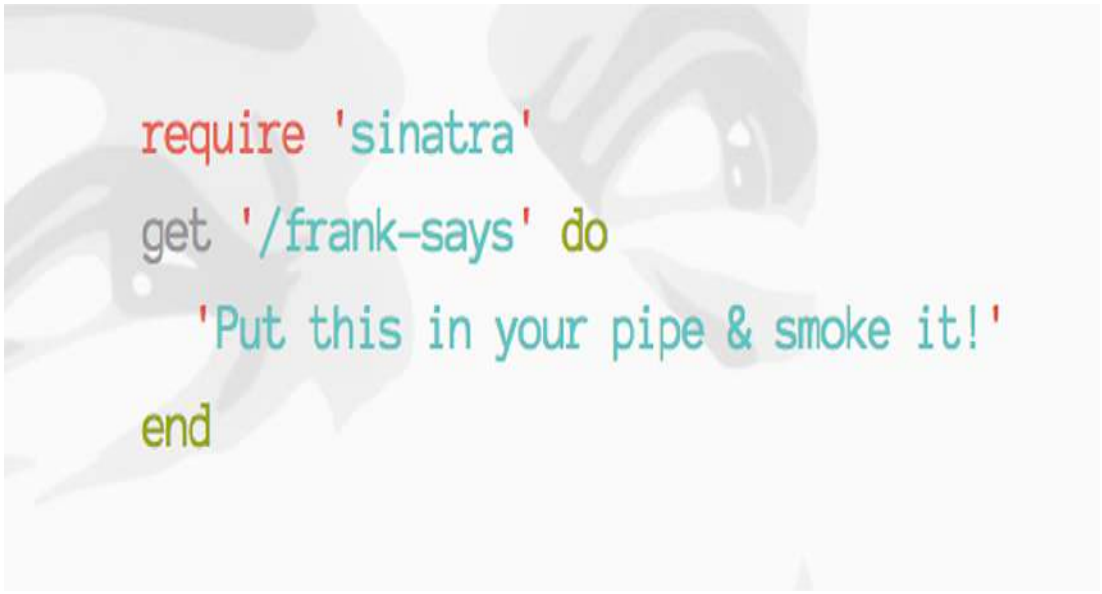


Figure 1.4: A sample program from the Sinatra homepage.

We'll put our "hello, world" app in a file called `hello_app.rb`:

```
$ touch hello_app.rb
```

The code itself closely parallels the program in Figure 1.4, as seen in Listing 1.8. (If you're wondering about the use of both single-quoted strings in Listing 1.8 and the double-quoted strings we saw in Section 1.2, you're ahead of the game; we'll learn the difference between the two in Chapter 2.)

Listing 1.8: A “hello, world” web app.
ruby_tutorial/hello_app.rb

Click here to view code image

```
require 'sinatra'

get '/' do
  'hello, world!'
end
```

We’ll cover the techniques in Listing 1.8 in more detail starting in Section 5.4, but the basic idea is that it defines the behavior for the *root URL* / when responding to an ordinary browser request (known as `GET`). The response itself is the required “hello, world!” string, which will be returned to the browser as a (very simple) web page.

To run the web application in Listing 1.8, all we need to do is call the `hello_app.rb` file using the same `ruby` command we used in Section 1.3; the `sinatra` gem magically takes care of the rest (Listing 1.9).

Listing 1.9: Running the Sinatra app with `ruby`.

Click here to view code image

```
$ ruby hello_app.rb
== Sinatra has taken the stage on 4567 for development with
Maximum connections set to 1024
Listening on localhost:4567, CTRL+C to stop
```

Here I’ve shown the output on my system, which runs a local web server on port number 4567 by default. This means you can view the app by visiting localhost:4567 in your browser. As seen in Figure 1.5, the effect on the cloud IDE is slightly different, but the idea is basically the same, and in either case the result should look something like Figure 1.6.

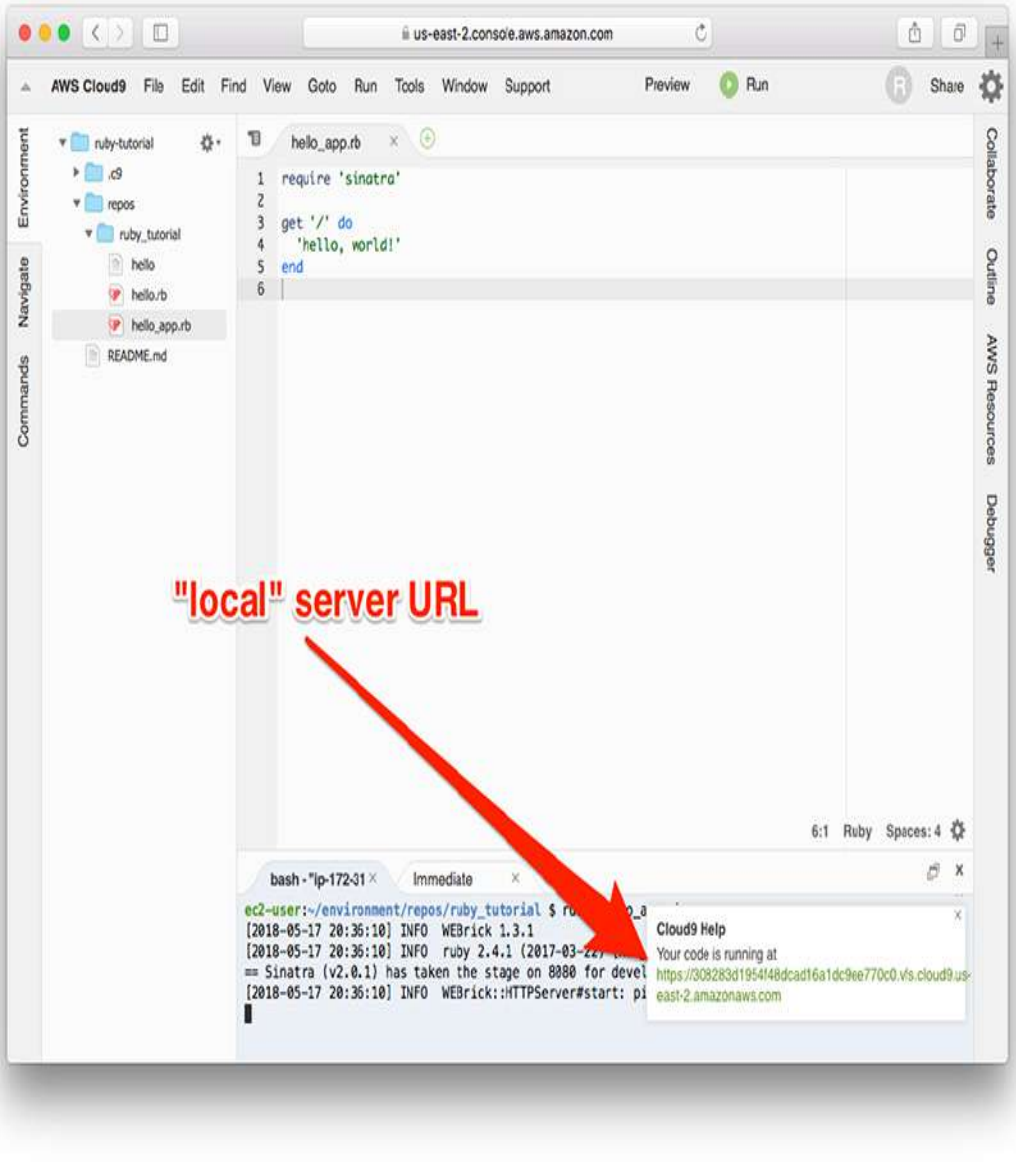


Figure 1.5: The “local” server running on the cloud IDE.

Figure 1.6: The hello app running locally.

It's worth noting that using Ruby to run a Sinatra app as in Listing 1.9 suffers from a major inconvenience: Seeing the effect of changes to the code requires quitting and restarting the server. This is fine for a quick change, but quickly becomes impractical for larger projects. We'll see how to get around this restriction using the `rerun` gem starting in Section 10.1.

1.5.1 Deployment

Now that we've got our app running locally, we're ready to deploy it to a production environment. This used to be practically impossible to do in a beginning tutorial, but

nowadays we can do it using a great hosting platform called Heroku. There's a bit of overhead to deploy something the first time, but deploying early and often is a core part of the Learn Enough philosophy of *shipping* (Box 1.3). Moreover, a simple app like "hello, world" is the best kind of app for first-time deployment, because there's so much less that can go wrong.

Box 1.3. Real Artists Ship

As legendary Apple cofounder Steve Jobs once said: *Real artists ship.* What he meant was that, as tempting as it is to privately polish in perpetuity, makers must *ship* their work—that is, actually finish it and get it out into the world. This can be scary, because shipping means exposing your work not only to fans but also to critics. "What if people don't like what I've made?" *Real artists ship.*

It's important to understand that shipping is a separate skill from making. Many makers get good at making things but never learn to ship. To keep this from happening to us, we'll follow the practice started in *Learn Enough Git to Be Dangerous* and ship several things in this tutorial. Shipping the "hello, world" app in this section is only the beginning!

As with the GitHub Pages deployment option used in previous tutorials (*Learn Enough CSS & Layout to Be Dangerous* and *Learn Enough JavaScript to Be Dangerous* among them), our first step is to put our project under version control with Git (as covered in *Learn Enough Git to Be Dangerous*, which you should consult now if your system isn't already configured for Git):

Click here to view code image

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

Although not strictly necessary, it's a good idea to push any newly initialized repository up to a remote backup. As in previous Learn Enough tutorials, we'll use GitHub for this purpose (Figure 1.7).[5]

[5] Previous versions of this tutorial used GitLab instead of GitHub because at the time private repos at GitHub weren't free. Since videos are harder to update than text, the screencasts that accompany this book still use GitLab, but the steps for GitHub are much the same (and are covered in several other Learn Enough tutorials, including *Learn Enough Git to Be Dangerous*). As usual, use your technical sophistication (Box 1.1) to resolve any discrepancies.

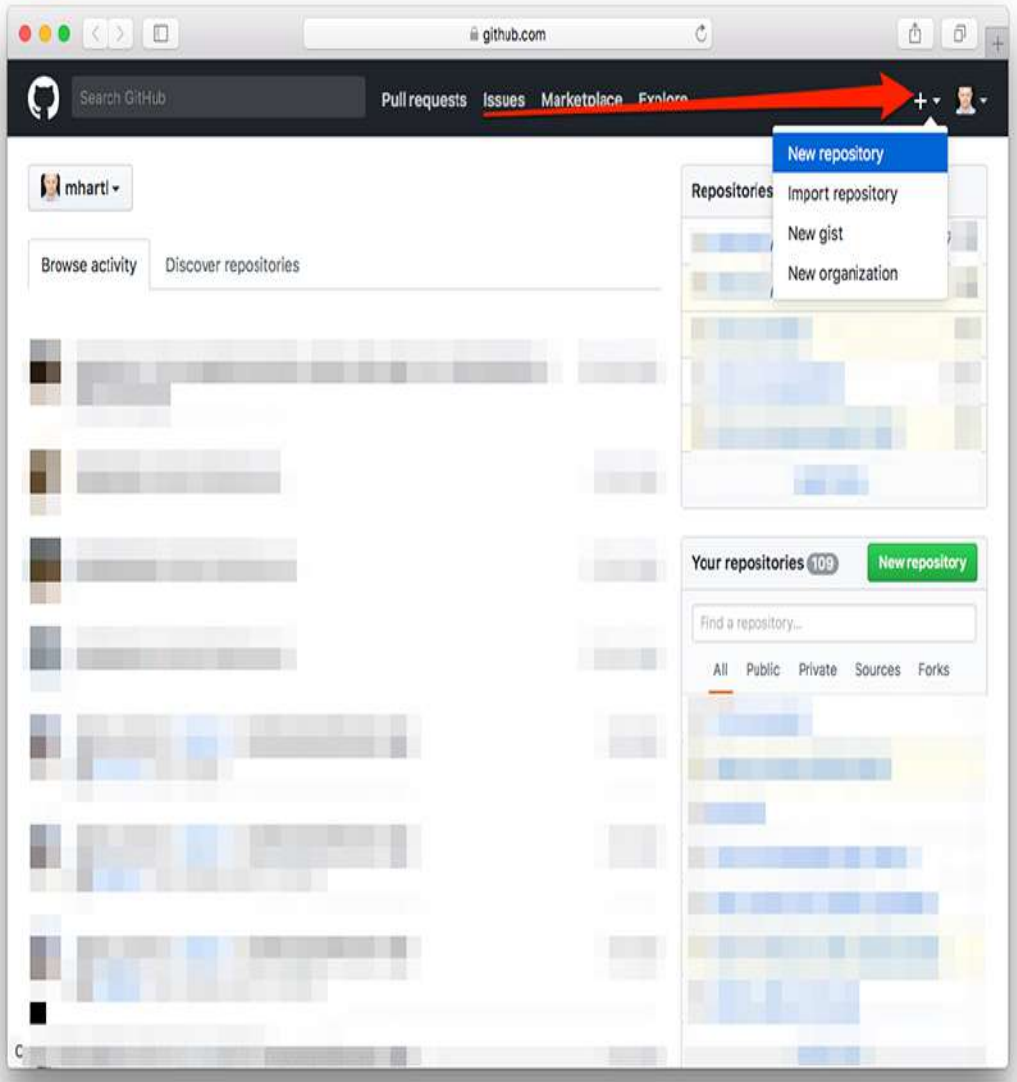


Figure 1.7: Creating a new repository at GitHub.

Because web apps sometimes include sensitive information like passwords or API keys, I like to err on the side of caution and use a *private* repository. Accordingly, be sure to select the Private option when creating the new repository at GitHub, as shown in Figure 1.8.

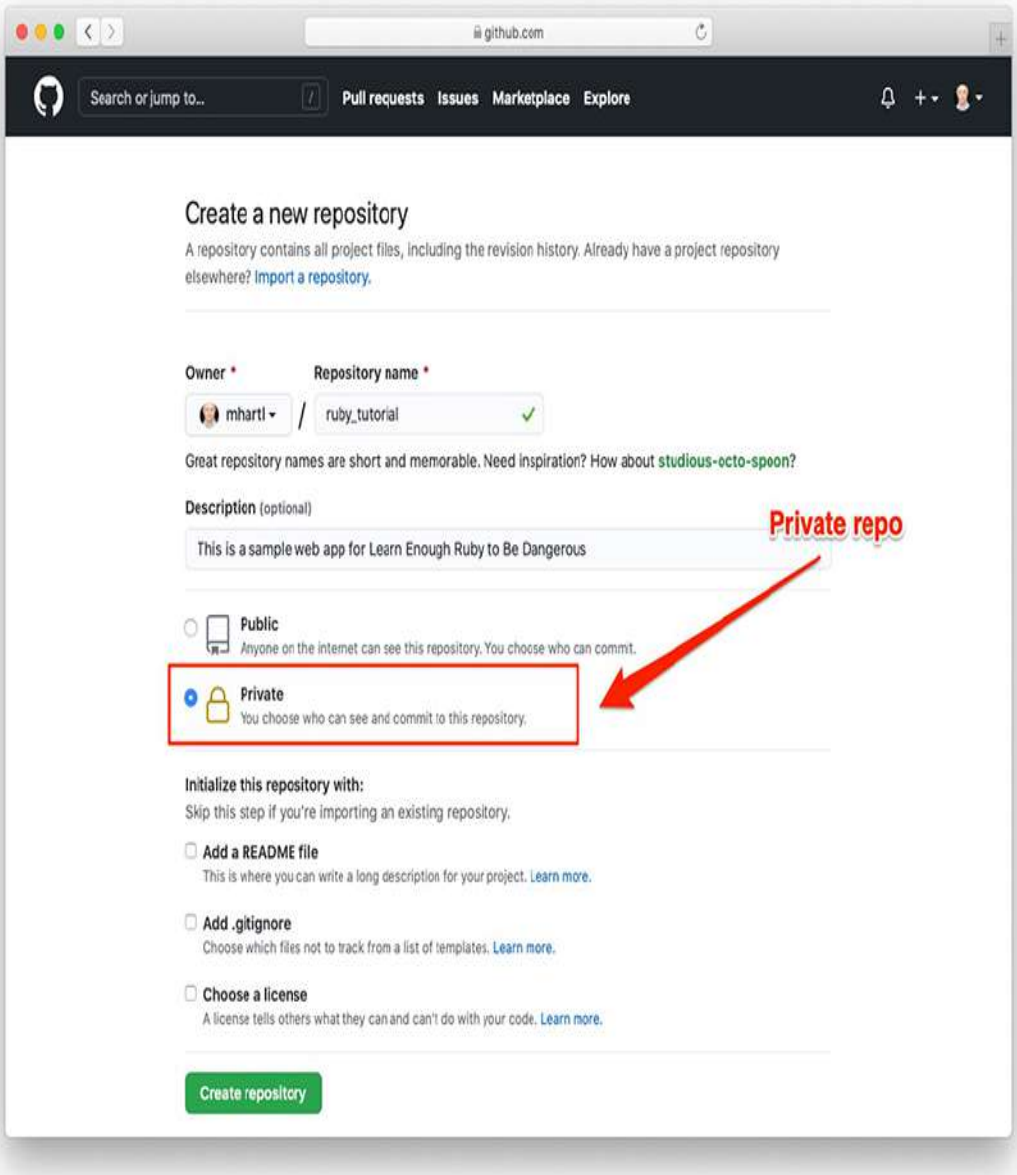


Figure 1.8: Using a private repo.

Next, configure your local system with the remote repository and push it up (taking care to fill in `<username>` with your GitHub username and using a GitHub personal access token when prompted for a password):

Click here to view code image

```
$ git remote add origin https://github.com/<username>/ruby_tutorial.git
$ git push -u origin main
```

Because videos are relatively hard to update, the screencasts that accompany this book use `master`, which was the default branch name for the first 15+ years of Git's existence, but the text has been updated to use `main`, which is the current preferred default. See the Learn Enough blog post "Default Git Branch Name with Learn Enough and the Rails Tutorial" for more information.

Next you'll have to create and configure a new Heroku account if you don't already have one. The first step is to sign up for Heroku. As part of this, you should set up Multi-Factor Authentication on your account.

The next step is to check to see if your system already has the Heroku command-line client installed:

Click here to view code image

```
$ heroku --version    # will work only if heroku is installed
heroku: command not found
```

This will display the current version number if the `heroku` command-line interface (CLI) is available, but on most systems it will be necessary to install the Heroku CLI by hand.[6] In particular, if you're working on the cloud IDE, you can install Heroku using the command shown in Listing 1.10.

[6] toolbelt.heroku.com

Listing 1.10: The command to install Heroku on the cloud IDE.

Click here to view code image

```
$ source < (curl -sL https://cdn.learnenough.com/heroku_install)
```

After running the command in Listing 1.10, you should now be able to verify the installation by displaying the current version number (details may vary):

Click here to view code image

```
$ heroku --version
heroku/7.59.2 linux-x64 node-v12.21.0
```

Once you've verified that the Heroku command-line interface is installed, the next step is to use the `heroku` command to log in to your account. If you're using a native development environment, simply type `heroku` at the command line, which

will automatically spawn a browser and let you log in with your Heroku email and password:

Click here to view code image

```
$ heroku login        # on a native system but not on the cloud IDE
$ # Spawns a browser window. Log in with your email and Heroku password.
```

If you're using the cloud IDE, you need to pass the `--interactive` option, which prevents the `heroku` command from trying to spawn a browser (which wouldn't work in the cloud). You also won't be able to log in using your regular Heroku password; instead, you'll have to create an *API Key* using the interface on your Heroku Account page (Figure 1.9). Once you've followed that step (and saved the result somewhere safe), you can log in using your email and the Account Key as your password:

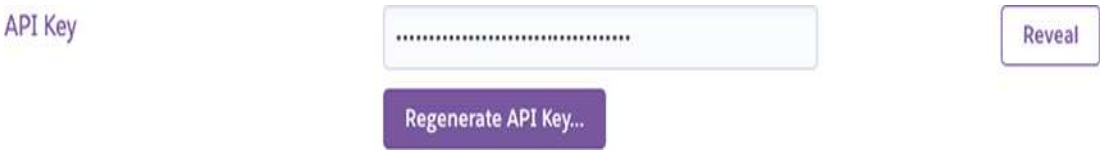


Figure 1.9: The API key at Heroku.

Click here to view code image

```
$ heroku login --interactive    # on the cloud IDE
Email: <your email>
Password: <your API Key, NOT your Heroku password>
```

After you've logged in, you can use the `heroku create` command to create a place on the Heroku servers for the sample app to live (Listing 1.11).

Finally, use the `heroku create` command to create a place on the Heroku servers for the sample app to live (Listing 1.11).

Listing 1.11: Creating a new application at Heroku.

Click here to view code image

```
$ heroku create
Creating app... done, ⬢ damp-depths-3
https://damp-depths-3.herokuapp.com/  |  https://git.heroku.com/damp-
depths-3.git
```

The `heroku` command creates a new subdomain just for our application, available for immediate viewing. There's nothing there yet, though, so let's get busy deploying.

The final steps involve some configuration that you can practically copy (with only minor modifications) from the Heroku documentation. We need only two more files, a "Rackup" (`.ru`) file called `config.ru` and a `Gemfile` specifying which gems our app uses (in this case, `sinatra` and `puma`, which is the production-grade web server that we saw briefly in Section 1.5):

```
$ touch config.ru Gemfile
```

Using your favorite text editor, fill these files with the contents shown in Listing 1.12 and Listing 1.13.

Listing 1.12: The Rack configuration file.
ruby_tutorial/config.ru

Click here to view code image

```
require './hello_app'
run Sinatra::Application
```

Listing 1.13: The hello app `Gemfile`.
ruby_tutorial/Gemfile

Click here to view code image

```
source 'https://rubygems.org'

ruby '3.1.2'   # Change this line if you're using a different Ruby v
ersion.

gem 'sinatra',  '2.2.2'
gem 'puma',     '5.6.5'
```

Now we're almost ready to deploy. We first need to *bundle* our gems (well, gem) using *Bundler*, and then add the files to Git:

Click here to view code image

```
$ gem install bundler -v 2.3.10
$ bundle _2.3.10_ install
$ bundle _2.3.10_ lock --add-platform x86_64-linux
```

```
$ git add -A
$ git commit -m "Add deployment configuration"
```

Note that the first three lines include an exact Bundler version number (2.3.10) for maximum compatibility. The third line may or may not be necessary depending on the exact system you're using, but in any case it does no harm to include it.

Finally, we can deploy to Heroku with a simple `git push`:

```
$ git push heroku main
```

That's it! Once the deployment is complete, our hello app is running in production (Figure 1.10). (*Note*: Heroku displays the Web URL of the app upon deployment, but you can run `heroku apps:info` at any time to see it again.)

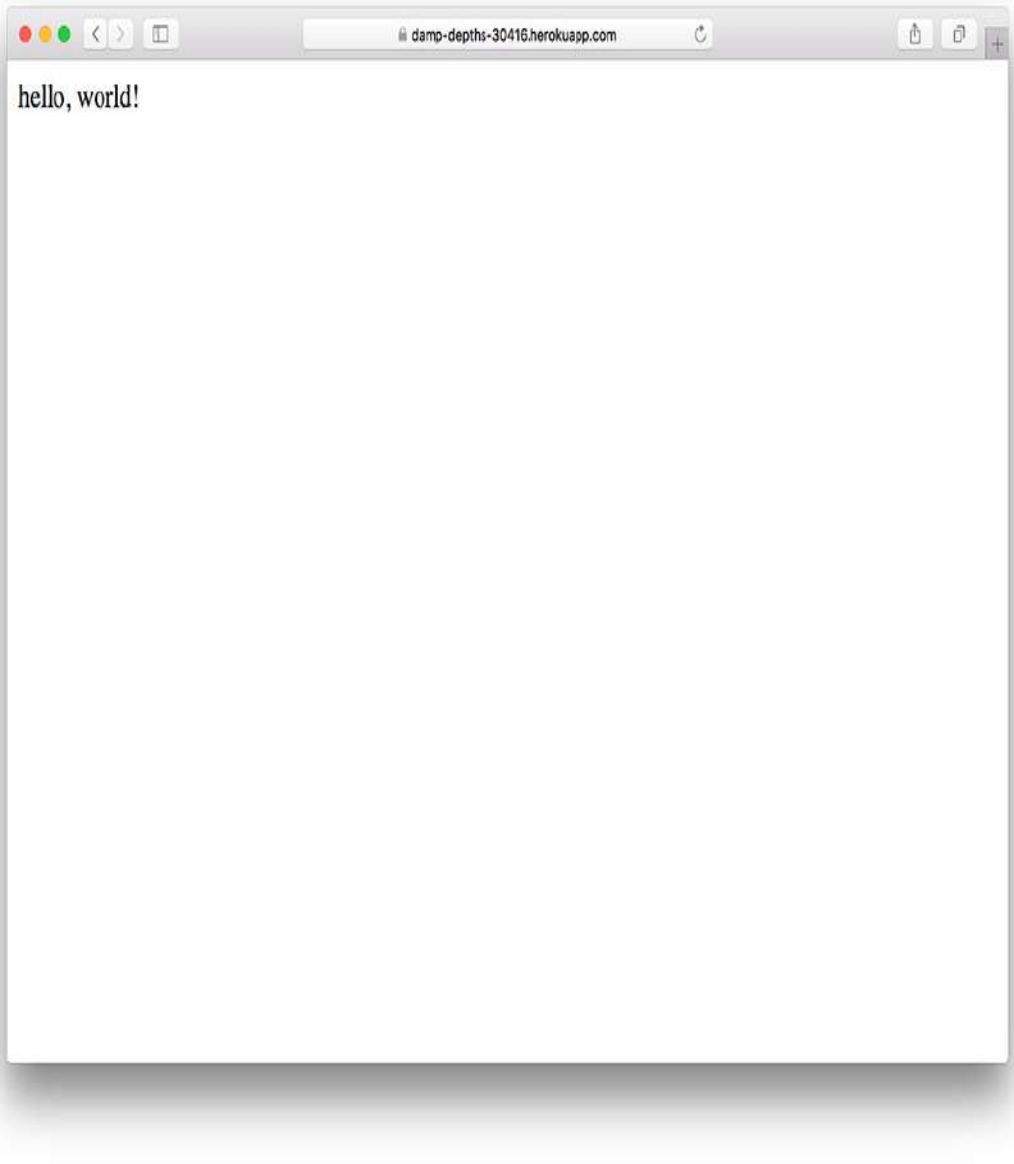


Figure 1.10: The hello app running in production.

"It's alive!" (Figure 1.11).[7]

[7] Image courtesy of Niday Picture Library/Alamy Stock Photo.

Figure 1.11: Bringing a site to life is easier than it used to be.

Although there were quite a few steps involved here, being able to deploy a real site so early is nothing short of miraculous. It may be a simple app, but it’s a real one, and being able to deploy it to production is an enormous step.

1.5.2 Exercises

1. Change “hello, world!” to “goodbye, world!” in `hello_app.rb`. Does the change display right away? What about after restarting the local server?
2. Commit your “goodbye, world!” changes and deploy the results to Heroku. Confirm that you can use `git push heroku` (omitting the branch name `main`) to complete the deployment.

Chapter 2

Strings

Strings are probably the most important data structure for everyday computing. They're used in practically every kind of program imaginable, and are also the raw material of the Web. As a result, strings make an excellent place to start our Ruby programming journey.

2.1 String Basics

Strings are made up of sequences of characters in a particular order. We've already seen several examples in the context of our "hello, world" programs in Chapter 1. Let's see what happens if we type a string by itself (without `puts`) into interactive Ruby:

```
$ irb
>> "hello, world!"
=> "hello, world!"
```

A sequence of characters typed literally is called a *string literal*, which we've created here using the double quote character `"`. The REPL prints the result of *evaluating* the line, which in the case of a string literal is just the string itself.

A particularly important string is one with no content, consisting simply of two quotes. This is known as an *empty string* (or sometimes *the* empty string):

```
>> ""
=> ""
```

We'll have more to say about the empty string in Section 2.4.2 and Section 3.1.

2.1.1 Exercises

1. Ruby supports common special characters such as tabs (`\t`) and new-lines (`\n`), which are two different forms of so-called *whitespace*. Show that `\t` and `\n` are interpreted as special characters inside double-quoted strings but not

inside single-quoted strings. What are their effects in each case? *Hint*: In irb, try executing commands like the ones shown in Listing 2.1.

Listing 2.1: Investigating whitespace in Ruby strings.

```
>> puts "hello\tgoodbye"
>> puts "hello\ngoodbye"
>> puts 'hello\tgoodbye'
>> puts 'hello\ngoodbye'
```

2.2 Concatenation and Interpolation

Two of the most important string operations are *concatenation* (joining strings together) and *interpolation* (putting variable content into strings). We'll start with concatenation, which we can accomplish using the `+` operator:[1]

[1] This use of `+` for string concatenation is common in programming languages, but in one respect it's an unfortunate choice, because addition is the canonical commutative operation in mathematics: $a + b = b + a$. (In contrast, multiplication is in some cases non-commutative; for example, when multiplying matrices it's often the case that $AB \neq BA$.) In the case of string concatenation, though, `+` is most definitely *not* a commutative operation, since, e.g., `"foo" + "bar"` is `"foobar"`, whereas `"bar" + "foo"` is `"barfoo"`. Partially for this reason, some languages (such as PHP) use a different symbol for concatenation, such as a dot `.` (yielding `"foo" . "bar"`).

Click here to view code image

```
$ irb
>> "foo" + "bar"            # String concatenation
=> "foobar"
>> "ant" + "bat" + "cat"    # Multiple strings can be concatenated at once.
=> "antbatcat"
```

Here the result of evaluating `"foo"` plus `"bar"` is the string `"foobar"`. (The meaning of the odd names "foo" and "bar" is discussed in *Learn Enough Command Line to Be Dangerous*.)

Let's take another look at string concatenation in the context of *variables*, which you can think of as named boxes that contain some value (as mentioned in *Learn Enough CSS & Layout to Be Dangerous* and discussed further in Box 2.1).[2]

[2] Image courtesy of Africa Studio/Shutterstock.

Box 2.1. Variables and Identifiers

If you've never programmed a computer before, you may be unfamiliar with the term *variable*, which is an essential idea in computer science. You can think

of a variable as a named box that can hold different (or “variable”) content.

As a concrete analogy, consider the labeled boxes that many elementary schools provide for students to store clothing, books, backpacks, etc. (Figure 2.1). The variable is the location of the box, the label for the box is the variable name (also called an *identifier*), and the content of the box is the variable value.

Figure 2.1: A concrete manifestation of computer variables.

In practice, these different definitions are frequently conflated, and “variable” is often used for any of the three concepts (location, label, or value).

As a concrete example, we can create variables for a first name and a last name using the `=` sign, as shown in Listing 2.2.

Listing 2.2: Using `=` to assign variables.

```
>> first_name = "Michael"
>> last_name = "Hartl"
```

Here `=` associates the identifier `first_name` with the string `"Michael"` and the identifier `last_name` with the string `"Hartl"`.

The identifiers `first_name` and `last_name` in Listing 2.2 are written in so-called snake case, whose name origins are obscure but which is probably the most common convention for Ruby variable names (Figure 2.2).[3] (In contrast, Ruby classes use the CamelCase convention discussed in *Learn Enough Java-Script to Be Dangerous* and described in more detail in Chapter 7.)

[3] Image courtesy of Kopytin Georgy/Shutterstock.

Figure 2.2: Snake case is the default for Ruby variable names.

Having defined the variable names in Listing 2.2, we can use them to concatenate the first and last names, while also inserting a space in between (Listing 2.3).

Listing 2.3: Concatenating string variables (and a string literal).

Click here to view code image

```
>> first_name + " " + last_name
=> "Michael Hartl"
```

Another way to build up strings is via *interpolation* using the number-sign curly-brace notation `#{...}`:

Click here to view code image

```
>> "#{first_name} is my first name."
=> "Michael is my first name."
```

Here Ruby automatically inserts, or *interpolates*, the value of the variable `first_name` into the string at the appropriate place.[4] Indeed, any code inside the curly braces will simply be evaluated by Ruby and inserted in place.

[4] Programmers familiar with Perl or PHP should compare this to the automatic interpolation of dollar sign variables in expressions like `"Michael $last_name"`.

We can use interpolation to replicate the result of Listing 2.3, as shown in Listing 2.4.

Listing 2.4: Concatenation review, then interpolating.

Click here to view code image

```
>> first_name + " " + last_name      # Concatenation, with a space in between
=> "Michael Hartl"
>> "#{first_name} #{last_name}"      # The equivalent interpolation
=> "Michael Hartl"
```

The two expressions shown in Listing 2.4 are equivalent, but I generally prefer the interpolated version because having to add the single space `" "` in between strings feels a bit awkward.

2.2.1 Single-Quoted Strings

All the examples so far have used *double-quoted* strings, but Ruby also supports *single-quoted* strings. For many uses, the two types of strings are effectively identical:

Click here to view code image

```
>> 'foo'          # A single-quoted string
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

There's an important difference, though; single-quoted strings are what are known as *raw strings*. For example, Ruby won't interpolate into single-quoted strings:

Click here to view code image

```
>> '#{first_name} #{last_name}'      # No interpolation!
=> "\#{first_name} \#{last_name}"
```

Note how irb returns values using double-quoted strings, which requires a backslash to *escape* special character combinations such as `#{`.

If double-quoted strings can do everything that single-quoted strings can do, and interpolate to boot, what's the point of single-quoted strings? They are often useful because they are truly literal, containing exactly the characters you type. For example, the "backslash" character is special on most systems, as in the literal newline `\n`. If you want a variable to contain a literal backslash, single quotes make it easier:

Click here to view code image

```
>> '\n'      # A literal 'backslash n' combination
=> "\\n"
```

As with the `#{` combination in our previous example, Ruby needs to escape the backslash with an additional backslash; inside double-quoted strings, a literal backslash is represented with *two* backslashes. For a small example like this, there's not much savings, but if there are lots of things to escape it can be a real help:

Click here to view code image

```
>> 'Newlines (\n) and tabs (\t) both use the backslash character: \.'
=> "Newlines (\\n) and tabs (\\t) both use the backslash character: \\."
```

This practice of escaping out characters is unnecessary inside single-quoted strings, *except* for single quotes themselves. For example, if you enter this in irb the REPL just hangs (Figure 2.3):[5]

Figure 2.3: Sometimes it's not easy when the REPL just hangs.

[5]Image courtesy of LorraineHudgins/Shutterstock.

Click here to view code image

```
$ irb
>> 'It's not easy being green'
```

This is because irb interprets `'It'` as a string, and the final `'` as the *opening* of a *second* string, and it just waits because it's looking for a closing `'`. (Another result, as seen above, is that the syntax highlighting looks odd—a side effect that is frequently useful as a visual hint of a syntax error.)

As noted in *Learn Enough Command Line to Be Dangerous*, the way to get out of this trouble is to hit `Ctrl-C`; we can then put in the string correctly by escaping out the apostrophe in `It's` with a backslash:

Click here to view code image

```
$ irb
>> 'It's not easy being green'
^C
>> 'It\'s not easy being green'
=> "It's not easy being green"
```

Finally, it's worth noting that, in the common case that both single and double quotes work just fine, you'll often find that the source code switches between the two without any apparent pattern. Don't worry—you'll get used to it soon enough.

2.2.2 Exercises

1. Assign variables `city` and `state` to your current city and state of residence. (If residing outside the U.S., substitute the analogous quantities.) Using interpolation, print a string consisting of the city and state separated by a comma and a space, as in "Los Angeles, CA".
2. Repeat the previous exercise but with the city and state separated by a tab character.

2.3 Printing

As we saw in Section 1.2 and subsequent sections, the Ruby way to print a string to the screen is to use the `puts` function:

Click here to view code image

```
>> puts "hello, world!"     # Print output
hello, world!
=> nil
```

This function operates as a *side effect*, which refers to anything a function does other than returning a value. In particular, the expression

```
puts "hello, world!"
```

prints the string to the screen and then returns nothing. This is why irb displays `nil` after the printed value: *Nil* is a contraction of the Latin *nihil*, which literally means "nothing".

> We'll generally omit `nil` when showing results in the REPL, but it's good to distinguish between functions that return values (almost all of them) and those like `puts` that operate using side effects.
>
> It's also worth noting that, as a Ruby function, `puts` can be called with parentheses, like this:

```
>> puts("hello, world!")
"hello, world!"
=> nil
```

This is perfectly valid Ruby, but conventionally `puts` is typically called without them.

> A closely related function is `print`, which will be familiar if you solved the exercise in Section 1.2.1. It works just like `puts`, but without adding an automatic newline:[6]
>
> [6] If it had been up to me, I probably would have swapped the roles of `puts` and `print`. This would have made `print "hello, world!"` the standard "hello, world!" program, which IMHO is much clearer, whereas the current method requires explaining what `puts` means. But I'm not Matz, so that's just how it goes.

```
>> print "hello, world!"
hello, world!=> nil
```

We can replicate the behavior of `puts` by appending a newline:

```
>> print "hello, world! \n"
hello, world!
=> nil
```

2.3.1 Exercises

1. What is the effect of giving `puts` multiple arguments? How about for `print`?

2.4 Attributes, Booleans, and Control Flow

Everything in Ruby, including strings, is an object. This means that we can get useful information about strings and do useful things with them using the same *dot notation* used in many object-oriented languages (e.g., JavaScript, as seen in *Learn Enough JavaScript to Be Dangerous*).

We'll start by accessing a string *attribute*, which is a piece of data attached to an object. In particular, in the console we can use the `length` attribute to find the number of characters in a string:

Click here to view code image

```
$ irb
>> "badger".length    # Accessing the "length" property of a string
=> 6
>> "".length          # The empty string has zero length.
=> 0
```

The `length` attribute is especially useful in comparisons, such as checking the length of a string to see how it compares to a particular value (note that the REPL supports "up arrow" to retrieve previous lines, just like the command-line terminal):

```
>> "badger".length > 3
=> true
>> "badger".length > 6
=> false
>> "badger".length >= 6
=> true
>> "badger".length < 10
=> true
>> "badger".length == 6
=> true
```

The last line uses the equality comparison operator `==`, which Ruby shares with many other languages. (Note that, like JavaScript, Ruby supports `===`, and indeed has several comparison operators in general, but `==` works in almost all cases of relevance.)

The return values in the comparisons above, which are always either `true` or `false`, are known as *boolean* values, after mathematician and logician George Boole (Figure 2.4).[7]

[7] Image courtesy of Yogi Black/Alamy Stock Photo.

Figure 2.4: True or false? This is a picture of George Boole.

Boolean values are especially useful for *control flow*, which lets us take actions based on the result of a comparison (Listing 2.5).

Listing 2.5: Control flow with `if`.

```
>> password = "foo"
=> "foo"
>> if (password.length < 6)
>> "Password is too short."
>> end
=> "Password is too short."
```

Note in Listing 2.5 that the comparison after `if` is in parentheses, and the `if` statement is terminated by the `end` keyword. The latter is required, but in Ruby (unlike many other languages) the parentheses are optional, and it's common to leave them off (Listing 2.6).

Listing 2.6: Control flow with `if` and no parentheses.

```
>> if password.length < 6
>>   "Password is too short."
>> end
=> "Password is too short."
```

Listing 2.5 and Listing 2.6 also follow a consistent indentation convention, which is irrelevant to Ruby but is important for human readers of the code (Box 2.2).

Box 2.2. Code Formatting

The code samples in this tutorial, including those in the REPL, are designed to show how to format Ruby in a way that maximizes readability and code comprehension. The programs executing Ruby programs, whether irb or Ruby itself, don't care about these aspects of the code, but human developers do.

While exact styles differ, here are some general guidelines for good code formatting:

- *Indent code to indicate block structure.* Pretty much every time you see an opening curly brace `{`, you'll end up indenting the subsequent line. (Some text editors even do this automatically.)
- *Use two spaces (typically via* emulated tabs*) for indentation.* Many developers use four or even eight spaces, but I find that two spaces are enough to indicate block structure visually while conserving scarce horizontal space.

- *Add newlines to indicate logical structure.* One thing I particularly like to do is add an extra newline after a series of variable assignments, in order to give a visual indication that the setup is done and the real coding can begin. An example appears in Listing 4.9.
- *Limit lines to 80 characters (also called "columns").* This is an old constraint, one that dates back to the early days of 80-character-width terminals. Many modern developers routinely violate this constraint, considering it outdated, but in my experience the 80-character limit is a good source of discipline, and will save your neck when using command-line programs like `less` (or when using your code in a document with more stringent width requirements, such as a book). A line that breaks 80 characters is a hint that you should introduce a new variable name, break an operation into multiple steps, etc., to make the code clearer for anyone reading it.

We'll see several examples of more advanced code-formatting conventions as we proceed throughout the rest of this tutorial.

We can add a second behavior using `else`, which serves as the default result if the first comparison is `false` (Listing 2.7).

Listing 2.7: Control flow with `if` and `else`.

Click here to view code image

```
>> password = "foobar"
>> if password.length < 6
>>   "Password is too short."
>> else
>>   "Password is long enough."
>> end
=> "Password is long enough."
```

The first line in Listing 2.7 *redefines* `password` by assigning it a new value. After reassignment, the `password` variable has length 6, so `password.length < 6` is `false`. As a result, the `if` part of the statement (known as the `if` *branch*) doesn't get evaluated; instead, Ruby evaluates the `else` branch, resulting in a message indicating that the password is long enough.

Unusually among programming languages, Ruby has a special `elsif` keyword meaning "else if", as shown in Listing 2.8 (Figure 2.5).[8]

[8] Image courtesy of Jessie Willcox Smith/Alamy Stock Photo.

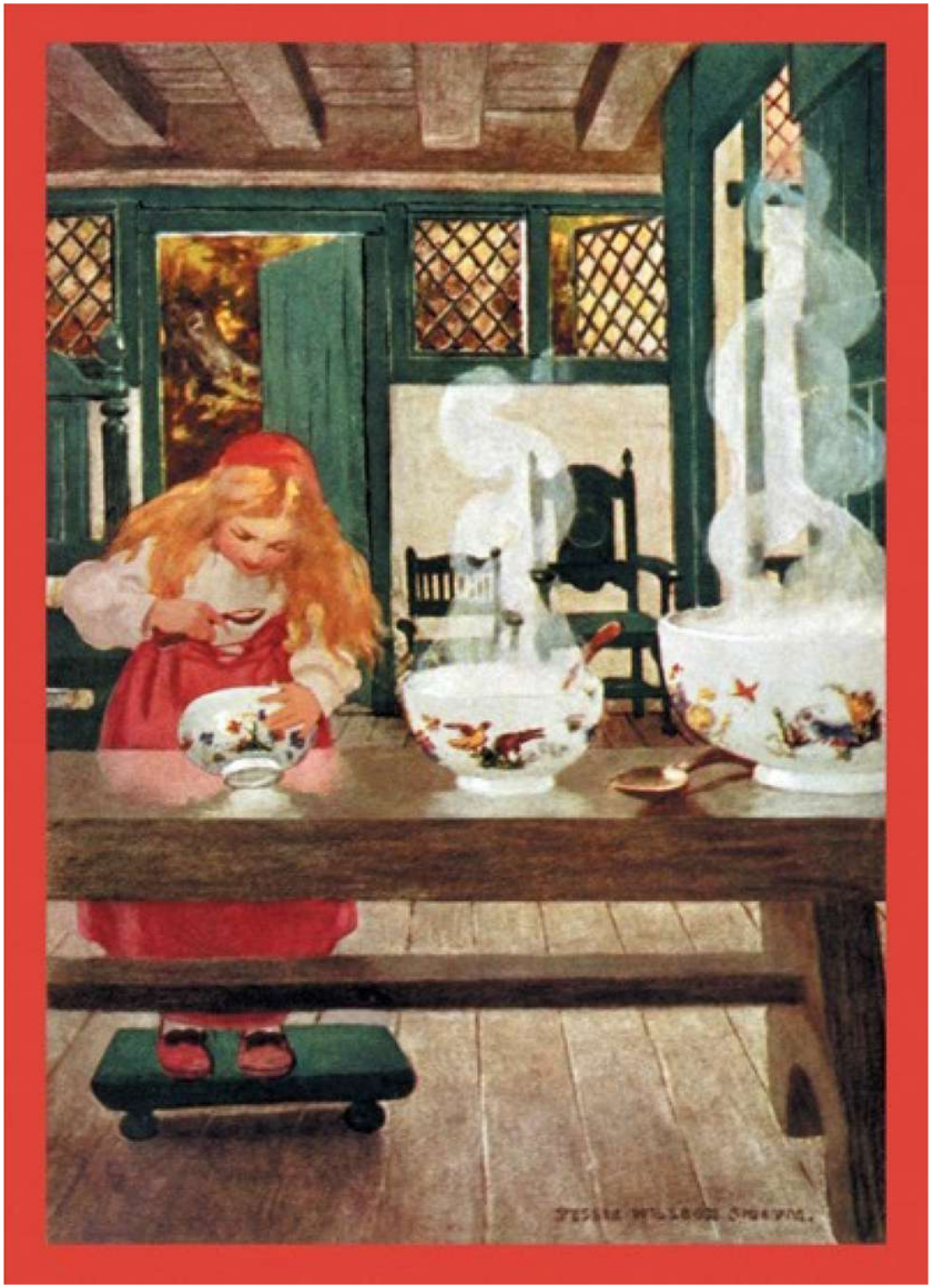


Figure 2.5: Goldilocks chooses control flow that is just right.

Listing 2.8: Control flow with `elsif`.

Click here to view code image

```
>> password = "goldilocks"
>> if password.length < 6
>>   "Password is too short."
>> elsif password.length < 50
>>   "Password is just right!"
>> else
>>   "Password is too long."
>> end
=> "Password is just right!"
```

As a final example, it's worth noting that Ruby allows us to place the `if` part *after* the statement when there's only one line:

Click here to view code image

```
>> password = "foo"
>> "Password is too short." if password.length < 6
=> "Password is too short."
```

The `if` here can be negated using `unless` instead, with the opposite comparison as well:

Click here to view code image

```
>> "Password is too short." unless password.length >= 6
=> "Password is too short."
```

It's essentially never wrong to use `if`, but in some cases the conditional sounds better using `unless`. I suggest pronouncing the conditional as if it were English and choosing whichever variant sounds more natural.

2.4.1 Combining and Inverting Booleans

Booleans can be combined or inverted using the `&&` ("and"), `||` ("or"), and `!` ("bang" or "not") operators.

Let's start with `&&`. When comparing two booleans with `&&`, *both* have to be `true` for the combination to be `true`. For example, if I said I wanted both french fries *and* a baked potato, the only way the combination could be true is if I could answer "yes" (true) to both of the questions "Do you want french fries?" and "Do you want a baked potato?" If my answer to either of those is false, then the combination must be false as well. The resulting combinations of possibilities are collectively known as a *truth table*; the truth table for `&&` appears in Listing 2.9.

Listing 2.9: The truth table for `&&` ("and").

```
>> true && false
=> false
>> false && true
=> false
>> false && false
=> false
>> true && true
=> true
```

We can apply this to a conditional as shown in Listing 2.10.

Listing 2.10: Using the `&&` operator in a conditional.

Click here to view code image

```
>> x = "foo"
>> y = ""
>> if x.length == 0 && y.length == 0
>>   "Both strings are empty!"
>> else
>>   "At least one of the strings is nonempty."
>> end
=> "At least one of the strings is nonempty."
```

In Listing 2.10, `y.length` is in fact `0`, but `x.length` isn't, so the combination is `false` (in agreement with Listing 2.9), and Ruby evaluates the `else` branch.

In contrast to `&&`, `||` lets us take action if *either* comparison (or both) is true (Listing 2.11).

Listing 2.11: The truth table for `||` ("or").

```
>> true || false
=> true
>> false || true
=> true
>> true || true
=> true
>> false || false
=> false
```

We can use `||` in a conditional as shown in Listing 2.12.

Listing 2.12: Using the `||` operator in a conditional.

Click here to view code image

```
>> if x.length == 0 || y.length == 0
>>   "At least one of the strings is empty!"
>> else
>>   "Neither of the strings is empty."
```

```
>> end
=> "At least one of the strings is empty!"
```

Note from Listing 2.11 that `||` isn't *exclusive*, meaning that the result is true even when *both* statements are true. This stands in contrast to colloquial usage, where a statement like "I want fries or a baked potato" implies that you want either fries *or* a baked potato, but you don't want both (Figure 2.6).[9]

Figure 2.6: Turns out I only wanted fries.

[9] Image courtesy of Rikaphoto/Shutterstock.

In addition to `&&` and `||`, Ruby supports *negation* via the "not" operator `!` (often pronounced "bang"), which just converts `true` to `false` and `false` to `true` (Listing 2.13).

Listing 2.13: The truth table for `!`.

```
>> !true
=> false
>> !false
=> true
```

We can use `!` in a conditional as shown in Listing 2.14. Note that parentheses *are* required in this case, because otherwise we're asking if `!x.length` is equal to `0`.

Listing 2.14: Using the `!` operator in a conditional.

```
>> if !(x.length == 0)
>>   "x is not empty."
>> else
>>   "x is empty."
>> end
=> "x is not empty."
```

The code in Listing 2.14 is valid Ruby, as it simply negates the test `x.length == 0`, yielding `true`:

```
>> !(x.length == 0)
=> true
```

In this case, though, it's more common to use `!=` ("not equals"):

```
>> if x.length != 0
>>   "x is not empty."
>> else
>>   "x is empty."
>> end
=> "x is not empty"
```

Because we're no longer negating the entire expression, we can omit the parentheses as before.

2.4.2 Bang Bang

Not all booleans are the result of comparisons, and in fact every Ruby object has a value of either `true` or `false` in a boolean context. We can force Ruby to use such a boolean context with `!!` (pronounced "bang bang"); because `!` converts between `true` and `false`, using *two* exclamation points returns us back to the original boolean:

```
>> !!true
=> true
```

```
>> !!false
=> false
```

Using this trick allows us to see that a string like `"foo"` is `true` in a boolean context:

```
>> !!"foo"
=> true
```

As it happens, the empty string is also `true` in a boolean context:[10]

[10] This is the sort of detail that varies from language to language.

```
>> !!""
=> true
```

In fact, even `0` is `true` in Ruby:

```
>> !!0
=> true
```

The only Ruby object that's false in a boolean context (other than `false` itself) is `nil`:

```
>> !!nil
=> false
```

2.4.3 Exercises

1. If `x` is `"foo"` and `y` is `""` (the empty string), what is the value of `x && y`? Verify using the "bang bang" notation that `x && y` is true in a boolean context. *Hint*: When applying `!!` to a compound expression, wrap the whole thing in parentheses.
2. What is `x || y`? What is it in a boolean context? Rewrite Listing 2.15 to use `x || y`, ensuring that the result is the same. (*Hint*: Switch the order of the strings.)

2.5 Methods

As noted in Section 2.4, Ruby string objects have an attribute called `length`, but in fact Ruby makes no fundamental distinction between attributes and *methods*, which can be

thought of as “messages” that get passed to objects, prompting the object to respond with some value.

In the language of object-oriented programming, a particular string, or *string instance*, is said to “respond to” a particular method. For example, a string instance responds to the `length` instance method by returning its length.

One important class of methods is *boolean methods*, which return `true` or `false`. Unusually among programming languages, Ruby allows punctuation in method names, and Ruby boolean methods conventionally end in a question mark `?`:

```
>> "badger".empty?
=> false
>> "".empty?
=> true
```

Here we see that `empty?` returns `true` for the empty string and `false` otherwise.

We can use the `empty?` method to rewrite code like Listing 2.10 more naturally using boolean methods, as shown in Listing 2.15.

Listing 2.15: Using boolean methods.

Click here to view code image

```
>> if x.empty? && y.empty?
>>   "Both strings are empty!"
>> else
>>   "At least one of the strings is nonempty."
>> end
=> "At least one of the strings is nonempty."
```

Strings also respond to a wealth of methods that return transformed versions of the string’s content. For example, strings respond to the instance method `downcase`, which (surprise!) converts the string to all lowercase letters (Figure 2.7):[11]

[11] Image courtesy of Pavel Kovaricek/Shutterstock.

Figure 2.7: This honey badger used to be a HONEY BADGER, but he don't care.

```
>> "HONEY BADGER".downcase
=> "honey badger"
```

Note that the `downcase` method returns a *new* string, without changing (or *mutating*) the original:

```
>> animal = "HONEY BADGER"
>> animal.downcase
=> "honey badger"
>> animal
=> "HONEY BADGER"
```

This is the sort of method that could be useful, for example, when standardizing on lowercase letters in an email address:[12]

[12] If you've exited and re-entered irb, `first_name` might no longer be defined, as such definitions don't persist from session to session. If this is the case, apply your technical sophistication (Box 1.1) to figure out what to do.

Click here to view code image

```
>> first_name = "Michael"
>> username = first_name.downcase
>> "#{username}@example.com"    # Sample email address
=> "michael@example.com"
```

As you might be able to guess, Ruby supports the opposite operation as well; before looking at the example below, see if you can guess the method for converting a string to uppercase (Figure 2.8).[13]

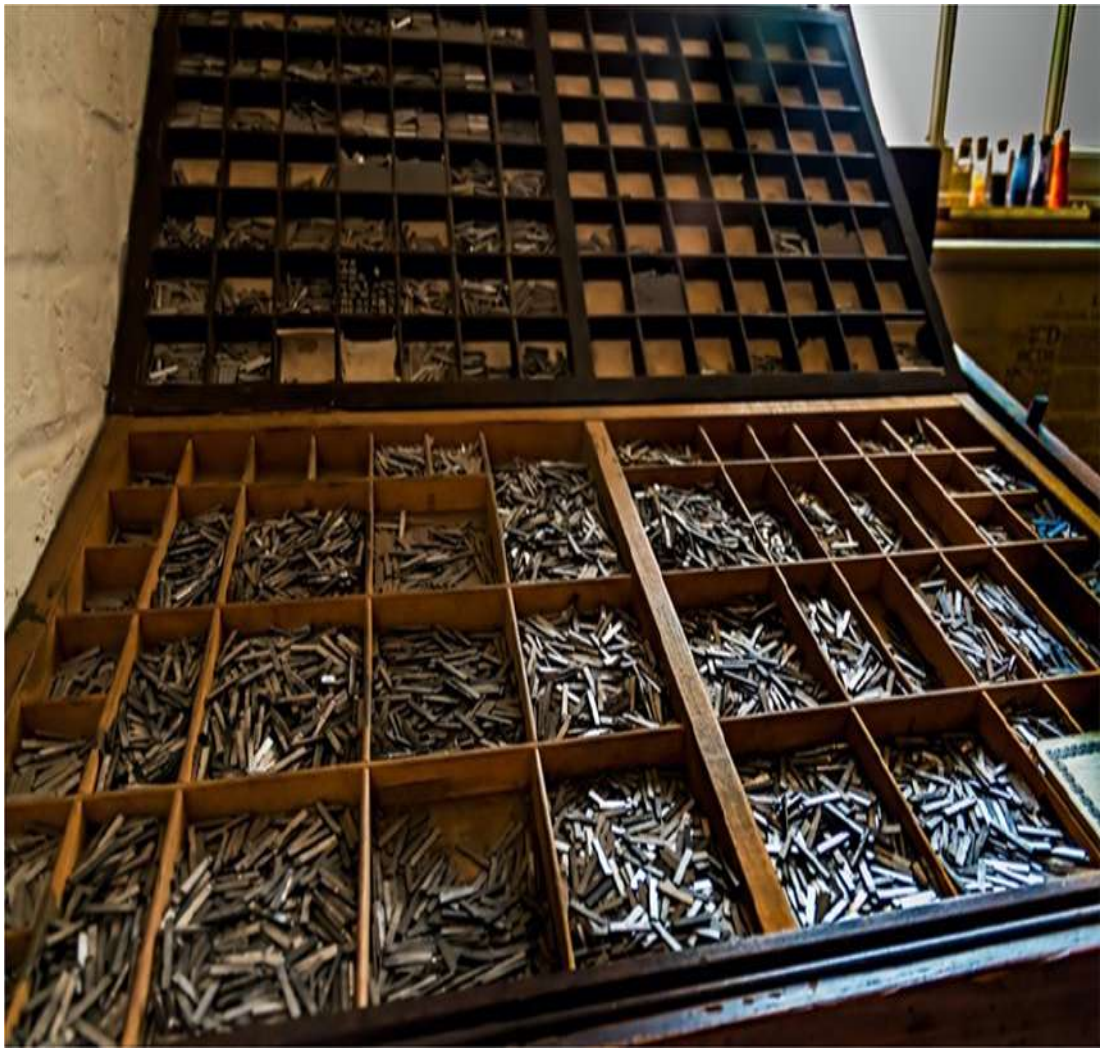


Figure 2.8: Early typesetters kept large letters in the “upper case” and small letters in the “lower case”.

[13] Image courtesy of arco1/123RF.

I’m betting you got the right answer (or at least came close):

```
>> last_name.upcase
=> "HARTL"
```

Being able to guess answers like this is a hallmark of technical sophistication, but as noted in Box 1.1 another key skill is being able to use the documentation. In particular, the Ruby documentation page on `String` has a long list of useful string instance methods.[14] Let’s take a look at some of them (Figure 2.9).

[14] You can find such pages by going directly to the official Ruby documentation, but the truth is that I nearly always find such pages by Googling things like “ruby string”. Be mindful of the version number—although Ruby is quite stable at this point, if you notice any discrepancies make sure you’re using documentation compatible with your own version of Ruby. (The links in this tutorial use version 2.5.0, but it shouldn’t ever matter.)

include? other_str → true or false

Returns `true` if *str* contains the given string or character.

```
"hello".include? "lo"   #=> true
"hello".include? "ol"   #=> false
"hello".include? ?h     #=> true
```

index(substring [, offset]) → integer or nil

index(regexp [, offset]) → integer or nil

Returns the index of the first occurrence of the given *substring* or pattern (*regexp*) in *str*. Returns `nil` if not found. If the second parameter is present, it specifies the position in the string to begin the search.

```
"hello".index('e')             #=> 1
"hello".index('lo')            #=> 3
"hello".index('a')             #=> nil
"hello".index(?e)              #=> 1
"hello".index(/[aeiou]/, -3)   #=> 4
```

replace(other_str) → str

Replaces the contents and taintedness of *str* with the corresponding values in *other_str*.

```
s = "hello"         #=> "hello"
s.replace "world"   #=> "world"
```

Figure 2.9: Some Ruby string methods.

Inspecting the methods in Figure 2.9, we see one that looks like this:

Click here to view code image

```
include? other_str → true or false
```

This indicates that the `include?` method takes an *argument*, `other_str`, and returns `true` or `false`. As with `empty?`, the ending question mark is an indication that `include?` is a boolean method.

As seen in Figure 2.9, parentheses are optional when calling methods in Ruby:

```
>> "hello".include? "lo"
=> true
>> "hello".include? "ol"
=> false
>> "hello".include? ?h
=> true
```

We see that, true to the description in Figure 2.9, `include?` returns `true` if the string instance contains the given string or character.

By the way, if you find the `?h` in the third line above confusing, so do I! I might have seen that syntax before, but I didn't remember it offhand, and I had to confirm in irb that `?h` is the same as `"h"`. You learn something every day!

Although the examples above omit parentheses, in general I prefer to use them apart from certain rare exceptions (such as the `puts` command first encountered in Listing 1.4). Let's take a look at a more extended example using parentheses, a longer quote (Figure 2.10),[15] and a greater variety of substrings shown in Listing 2.16.

Figure 2.10: Hamlet, Prince of Denmark, asks: “To be, or not to be, that is the question.”

[15]Image courtesy of Everett Collection/Shutterstock.

Listing 2.16: Include or does not include? That is the question.

Click here to view code image

```
>> soliloquy = "To be, or not to be, that is the question:"
>> soliloquy.include?("To be")        # Does it include the substring "To be"?
=> true
>> soliloquy.include?("question")     # What about "question"?
=> true
>> soliloquy.include?("nonexistent")  # This string doesn't appear.
=> false
>> soliloquy.include?("TO BE")        # String inclusion is case-sensitive.
=> false
```

We see from Listing 2.16 that `String#include?` (so-written to indicate that `include?` is a `String` instance method) can be called with arbitrary substrings, handles spaces just fine, and is case-sensitive.

2.5.1 Exercises

1. Write the Ruby code to test whether the string “hoNeY BaDGer” includes the string “badger” without regard to case.
2. Using the documentation, figure out how to capitalize a string. What happens if you capitalize a string that is already all-caps?
3. See if you can guess the Ruby boolean method to test if an object is `nil` (equivalent to `object == nil`). Use it to show that the empty string isn’t `nil`.

2.6 String Iteration

Our final topic on strings is *iteration*, which is the practice of repeatedly stepping through an object one element at a time. Iteration is a common theme in computer programming, and we’ll get plenty of practice in this tutorial. We’ll also see how one sign of your growing power as a developer is learning how to *avoid* iteration entirely (as discussed in Chapter 6 and Section 8.5).

In the case of strings, we’ll be learning how to iterate one *character* at a time. There are two main prerequisites to this: First, we need to learn how to access a particular character in a string, and second, we need to learn how to make a *loop*.

We can figure out how to access a particular string character by consulting the list of String methods, which includes the following entry:

> **`str[index]` → `new_str or nil`**
>
> Element Reference—If passed a single `index`, returns a substring of one character at that index.

Looking at the first example in the documentation and applying it to the `soliloquy` string from Section 2.5 lets us see how it works, as shown in Listing 2.17.

Listing 2.17: Investigating the behavior of `str[fixnum]`.

Click here to view code image

```
>> puts soliloquy   # Just a reminder of what the string is
To be, or not to be, that is the question:
>> soliloquy[0]
=> "T"
>> soliloquy[1]
=> "o"
>> soliloquy[2]
=> " "
```

We see in Listing 2.17 that Ruby supports a bracket notation for accessing string elements, so that `[0]` returns the first character, `[1]` returns the second, and so on. (We'll discuss this possibly counter-intuitive numbering convention, called "zero-offset", further in Section 3.1.) Each number `0`, `1`, `2`, etc., is called an *index* (plural *indexes* or *indices*).

Now let's look at our first example of a loop. In particular, we'll use a `for` loop that defines an index value `i` and does an operation for each value in the *range* `0..4` (Listing 2.18).[16]

[16] According to the official Ruby documentation, a `for` loop can include an optional `do` at the end, and this was included in earlier versions of this tutorial. Some readers reported problems with the `do`, though, so it has now been removed.

Listing 2.18: A simple `for` loop.

```
>> for i in 0..4
?>   puts i
>> end
0
1
2
3
4
```

This is Ruby's version of the classic "for loop" that is exceptionally common across an astonishing variety of programming languages, from C and C++ to JavaScript, Perl, and PHP. Unlike those languages, though, which explicitly increment a counter variable, Ruby defines a range of values directly via a special `Range` data type.

Listing 2.18 is arguably a little more elegant than the equivalent "classic" `for` loop seen in *Learn Enough JavaScript to Be Dangerous* (Listing 2.19), but it's still not very good Ruby.

Listing 2.19: A `for` loop in JavaScript.

```
> for (i = 0; i < 5; i++) {
  console.log(i);
}
0
1
2
3
4
```

As a language and as a community, Ruby is especially vigilant about avoiding plain `for` loops. As computer scientist (and personal friend) Mike Vanier (Figure 2.11) once put it in an email to Paul Graham:

Figure 2.11: Just a few more `for` loops and Mike Vanier will be a millionaire.

> This [tedious repetition] grinds you down after a while; if I had a nickel for every time I've written "for (i = 0; i < N; i++)" in C I'd be a millionaire.

In order to avoid getting ground down, we'll learn how to use a special `each` method (Section 3.5) that is especially characteristic of Ruby as compared to other languages. We'll also see how Ruby lets us avoid loops entirely using functional programming (Chapter 6 and Section 8.5).

For now, though, let's build on Listing 2.18 to iterate through all the characters in the first line of Hamlet's famous soliloquy. The only new thing we need is the index for when the loop should stop. In Listing 2.18, we hard-coded the upper limit (`4`), and we could do the same here if we wanted. The `soliloquy` variable is a bit long to count the characters by hand, though, so let's ask Ruby to tell us using the `length` property (Section 2.4):

```
>> soliloquy.length
=> 42
```

This exceptionally auspicious result suggests writing code like this:

```
for i in 0..41
  puts soliloquy[i]
end
```

This code will work, and it is in perfect analogy with Listing 2.18, but it also raises a question: Why hard-code the length when we can just use the `length` property in the loop itself?

The answer is that we shouldn't, and when looping it's common practice to use the `length` property whenever possible. The resulting improved `for` loop (with result) appears in Listing 2.20.

Listing 2.20: Combining `length` and a `for` loop.

Click here to view code image

```
>> for i in 0..(soliloquy.length - 1)
?>   puts soliloquy[i]
>> end
T
o

b
e
.
.
.
```

```
t
i
o
n
:
```

As noted above, `for` loops are best avoided if at all possible, but this less elegant style of looping is still an excellent place to start. As we'll see in Chapter 8, one powerful technique is to write a *test* for the functionality we want, then get it passing any way we can, and then *refactor* the code to use a more elegant method. The second step in this process (called *test-driven development*, or TDD) often involves writing inelegant but easy-to-understand code—a task at which the humble `for` loop excels.

2.6.1 Exercises

1. Write a `for` loop that prints out the characters of `soliloquy` in reverse order. *Hint*: Inside the loop, what is `soliloquy.length - i`?

Chapter 3

Arrays

In Chapter 2, we saw that strings can be thought of as sequences of characters in a particular order. In this chapter, we'll learn about the *array* data type, which is the general Ruby container for a list of arbitrary elements in a particular order. We'll start by explicitly connecting strings and arrays via the `String#-split` method (Section 3.1), and then learn about various other array methods throughout the rest of the chapter.

3.1 Splitting

So far we've spent a lot of time understanding strings, and there's a natural way to get from strings to arrays via the `split` method:

Click here to view code image

```
>> "ant bat cat".split(" ")      # Split a string into a three-element array.
=> ["ant", "bat", "cat"]
```

We see from this result that `split` returns a list of the strings that are separated from each other by a space in the original string.

Splitting on space is one of the most common operations, but we can split on nearly anything else as well:

Click here to view code image

```
>> "ant,bat,cat".split(",")
=> ["ant", "bat", "cat"]
>> "ant, bat, cat".split(", ")
=> ["ant", "bat", "cat"]
>> "antheybatheycathey".split("hey")
=> ["ant", "bat", "cat"]
```

We can even split a string into its component characters by splitting on the empty string:

Click here to view code image

```
>> "badger".split("")
=> ['b', 'a', 'd', 'g', 'e', 'r']
```

We'll put this basic technique to good use in Section 5.3.

Perhaps the most common use of `split` is with *no* arguments; in this case, the default behavior is to split on *whitespace* (such as spaces, tabs, or newlines):

Click here to view code image

```
>> "ant bat cat".split
=> ["ant", "bat", "cat"]
>> "ant     bat\t\tcat\n    duck".split
=> ["ant", "bat", "cat", "duck"]
```

We'll investigate this case more closely when discussing *regular expressions* in Section 4.3.

3.1.1 Exercises

1. Assign `a` to the result of splitting the string "A man, a plan, a canal, Panama" on comma-space. How many elements does the resulting array have?
2. Can you guess the method to reverse `a` in place? (Google around if necessary.)

3.2 Array Access

Having connected strings with arrays via the `split` method, we'll now discover a second close connection as well. Let's start by assigning a variable to an array of characters created using `split`:

Click here to view code image

```
>> a = "badger".split("")
=> ["b", "a", "d", "g", "e", "r"]
```

We can access particular elements of `a` using the same bracket notation we first encountered in Section 2.6, as seen in Listing 3.1.

Listing 3.1: Array access with the bracket notation.

```
>> a[0]
=> "b"
>> a[1]
=> "a"
>> a[2]
=> "d"
```

We see from Listing 3.1 that, as with strings, arrays are *zero-offset*, meaning that the "first" element has index `0`, the second has index `1`, and so on. This convention can be confusing, and in fact it's common to refer to the initial element for zero-offset arrays as the "zeroth" element as a reminder that the indexing starts at `0`. This convention can also be confusing when using multiple languages (some of which start array indexing at `1`), as illustrated in the xkcd comic strip "Donald Knuth".[1]

[1] This particular xkcd strip takes its name from renowned computer scientist Donald Knuth (pronounced "kuh-NOOTH"), author of *The Art of Computer Programming* and creator of the TEX typesetting system used to prepare many technical documents, including this one.

So far we've dealt exclusively with arrays of characters, but Ruby arrays can contain all kinds of elements:

Click here to view code image

```
>> soliloquy = "To be, or not to be, that is the question:"
>> a = ["badger", 42, soliloquy.include?("To be")]
=> ["badger", 42, true]
>> a[2]
=> true
>> a[3]
=> nil
```

We see here that the square bracket access notation works as usual for an array of mixed types, which shouldn't come as a surprise. We also see that trying to access an array index outside of the defined range returns `nil` (a value which we saw before in the context of `puts` (Listing 1.4)). This might be a surprise if you have previous programming experience, since many languages raise an error if you try to access an element that's out of range, but Ruby is more tolerant in this regard.

Another convenient feature of Ruby bracket notation is supporting *negative* indices, which count from the end of the array:

```
>> a[-2]
=> 42
```

Among other things, negative indices give us a compact way to select the *last* element in an array. Because arrays, like strings, respond to a `length` method, we

could do it directly:

```
>> a[a.length - 1]
=> true
```

But it's even easier like this:

```
>> a[-1]
=> true
```

This is such a common operation that Ruby supplies a special `last` method just for doing it:

```
>> a.last
=> true
```

A final common case is where we want to access the final element and remove it at the same time. We'll cover the method for doing this in Section 3.4.2.

3.2.1 Exercises

1. Write a `for` loop to print out the characters obtained from splitting "honey badger" on the empty string.
2. See if you can guess the value of `a[100]` in a boolean context. Use `!!` to confirm.

3.3 Array Slicing

In addition to supporting the bracket notation described in Section 3.2, Ruby supports a technique known as *array slicing* for accessing multiple elements at a time. In anticipation of learning to *sort* in Section 3.4, let's redefine our array `a` to have purely numerical elements:

```
>> a = [42, 8, 17, 99]
=> [42, 8, 17, 99]
```

One way to slice an array is to provide two arguments, an index and a number of elements, which returns the given number of elements starting at the given index. For example, for an array with four elements, `slice(2, 2)` returns the "second" and "third" ones (recall that the "first" or zeroth element has index `0`):

```
>> a.slice(2, 2)
=> [17, 99]
```

Another technique (and my favored one) is to use the `Range` data type we met briefly in Listing 2.18, which returns the elements with index between the beginning and end of the range:

```
>> a.slice(1..3)
=> [8, 17, 99]
```

Unlike most languages, Ruby lets us perform array slicing directly with the bracket notation:

```
>> a[2, 2]
=> [17, 99]
>> a[1..3]
=> [8, 17, 99]
```

The second example above, using a range, is probably the most common way to slice arrays in real-world Ruby code.

Finally, it's worth noting that ranges can easily be converted to arrays using the "to array" method, `to_a`:

Click here to view code image

```
>> (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> ('a'..'z').to_a
=> ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
    "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
```

We see in the second example above that ranges work on letters as well as numbers.

3.3.1 Exercises

1. Define an array with the numbers 1 through 10. Use slicing and `length` to select the third element through the third-to-last. Accomplish the same task using a negative index.

2. Show that strings also support the `slice` method by selecting just `bat` from the string `"ant bat cat"`. (You might have to experiment a little to get the indices just right.)

3. By combining a range, `to_a`, and array slicing, create an array containing the first 13 letters in the alphabet.

3.4 More Array Methods

In addition to `last`, `length`, and `slice`, arrays respond to a wealth of other methods. As usual, the documentation is a good place to go for details.

As with strings, arrays respond to an `include?` method to test for element inclusion:

Click here to view code image

```
>> a = [42, 8, 17, 99]
=> [42, 8, 17, 99]
>> a.include?(42)       # Test for element inclusion.
=> true
>> a.include?("foo")
=> false
```

3.4.1 Sorting and Reversing

You can also sort an array in place—an excellent trick that in ye olden days of C often required a custom implementation.[2] In Ruby, we just call `sort`:

[2] This isn't entirely fair to C: Ruby itself is written in C, so `Array#sort` actually is just such a "custom implementation"!

Click here to view code image

```
>> a.sort
=> [8, 17, 42, 99]
>> a                    # `a` hasn't changed as the result of `sort`.
=> [42, 8, 17, 99]
```

As you might expect for an array of integers, `a.sort` sorts the array numerically (unlike, e.g., JavaScript, which confusingly sorts them "alphabetically", so that 17 comes before 8). We also see that (again unlike JavaScript) sorting an array doesn't change the array itself.

Ruby includes a second sort method that *does* change the array, sorting it in place:

Click here to view code image

```
>> a.sort!
=> [8, 17, 42, 99]
>> a                    # `a` has changed as the result of `sort!`.
=> [8, 17, 42, 99]
```

Here the `sort!` (read "sort-bang") method makes use of Ruby's ability to include punctuation in method names, as we saw in Section 2.5 with boolean methods like `include?`. In this case, the exclamation point indicates that the method *mutates* (changes) the underlying object. Any time you see a Ruby method ending in `!`, it's more than likely that some sort of mutation is occurring.[3]

[3] A notable exception occurs in Ruby on Rails, which often uses exclamation points to distinguish between methods that return `false` on failure and those that raise an *exception* (Section 5.2). Thus, the Active Record method for saving to a database, called `save`, returns `false` if the save fails for any reason, while `save!` raises an exception instead.

Another useful method—one we'll put to good use in developing our palindrome theme starting in Section 5.3—is the `reverse` method:

Click here to view code image

```
>> a.reverse
=> [99, 42, 17, 8]
>> a                    # Like `sort`, `reverse` doesn't mutate the array.
=> [8, 17, 42, 99]
```

As with `sort`, `reverse` *returns* a reversed array, but doesn't change the array itself. Can you guess the method for mutating an array by reversing it in place?

3.4.2 Pushing and Popping

One useful pair of array methods is `push` and `pop`; `push` lets us append an element to the end of an array, while `pop` removes it:

Click here to view code image

```
>> a.push(6)                    # Pushing onto an array
=> [8, 17, 42, 99, 6]
>> a.push("foo")
=> [8, 17, 42, 99, 6, "foo"]
>> a.pop                        # `pop` returns the value itself
=> "foo"
>> a.pop
=> 6
>> a.pop
=> 99
>> a
=> [8, 17, 42]
```

As noted in the comments, `pop` returns the value of the final element (while removing it as a side effect), while `push` simply returns the resulting array.

We are now in a position to appreciate the comment made in Section 3.2 about obtaining the last element of the array, as long as we don't mind mutating it:

Click here to view code image

```
>> the_answer_to_life_the_universe_and_everything = a.pop
=> 42
```

Finally, Ruby supports a second (and very commonly used) notation for pushing onto arrays, the so-called "shovel operator" `<<`:

Click here to view code image

```
>> a << "badger"
=> [8, 17, "badger"]
>> a << "ant" << "bat" << "cat"
=> [8, 17, "badger", "ant", "bat", "cat"]
```

As seen in the second example, the shovel operator can be *chained*, pushing a sequence of elements onto the end of the array.

3.4.3 Undoing a Split

A final example of an array method, one that brings us full circle from Section 3.1, is `join`. Just as `split` splits a string into array elements, `join` joins array elements into a string (Listing 3.2).

Listing 3.2: Different ways to join.

Click here to view code image

```
>> a = ["ant", "bat", "cat", 42]
=> ["ant", "bat", "cat", 42]
>> a.join                          # Join on default (empty space).
=> "antbatcat42"
>> a.join(", ")                    # Join on comma-space.
=> "ant, bat, cat, 42"
>> a.join(" -- ")                  # Join on double dashes.
=> "ant -- bat -- cat -- 42"
```

Note that `42`, which is an integer, is automatically converted to a string in the join.

3.4.4 Exercises

1. Confirm your guess about the version of `reverse` that mutates an array by reversing it in place.
2. The `split` and `join` methods are almost inverse operations, but not quite. In particular, confirm using `==` that `a.join(" ").split(" ")` in Listing 3.2 is *not* the same as `a`. Why not?
3. Using the array documentation, figure out how to push onto or pop off the *front* of an array. *Hint*: The names aren't intuitive at all, so you might have to work a bit.

3.5 Array Iteration

One of the most common tasks with arrays is iterating through their elements and performing an operation with each one. This might sound familiar, since we solved the exact same problem with strings in Section 2.6, and indeed the solution is virtually the same. All we need to do is adapt the `for` loop from Listing 2.20 to arrays, i.e., replace `soliloquy` with `a`, as shown in Listing 3.3.

Listing 3.3: Combining array access and a `for` loop.

Click here to view code image

```
>> for i in 0..(a.length - 1)
?>   puts a[i]
>> end
ant
bat
cat
42
```

That's convenient, but it's not the best way to iterate through arrays, and Mike Vanier still wouldn't be happy (Figure 3.1).

Figure 3.1: Mike Vanier is still annoyed by typing out `for` loops.

Luckily, looping the Right Way™ is easier than it is in most other languages, so we can actually cover it here (unlike in, e.g., *Learn Enough JavaScript to Be Dangerous*, when we had to wait until Chapter 5). The trick is to use a special `each` method that is particularly characteristic of Ruby, as shown in Listing 3.4.

Listing 3.4: Using `each` to iterate over an array the Right Way™.

Click here to view code image

```
>> a.each do |element|
?>   puts element
```

```
>> end
ant
bat
cat
42
```

The array `a` responds to the `each` method by yielding each element in turn, which in this case we simply print to the screen. (The syntax in Listing 3.4 uses a Ruby *block*, but don't worry about it for now; we'll cover blocks in more detail in Section 5.4.) Using the `each` method, we can iterate directly through the elements in an array, thereby avoiding having to type out Mike Vanier's *bête noire*, "for (i = 0; i < N; i++)". The result is cleaner code and a happier programmer (Figure 3.2).

Figure 3.2: Using `each` has made Mike Vanier a little happier.

3.5.1 Exercises

1. Combine `reverse` and `each` to print out an array's elements in reverse order. *Hint*: You can call each method in sequence, as in `a.reverse.-each`, a technique known as *method chaining* (covered in Section 5.3).

Chapter 4

Other Native Objects

Now that we've taken a look at strings and arrays, we'll continue with a tour of some other important Ruby objects: math, dates, regular expressions, and hashes.

4.1 Math

Like most programming languages, Ruby supports a large number of mathematical operations:

```
>> 1 + 1
=> 2
>> 2 - 3
=> -1
>> 2 * 3
=> 6
>> 10/5
=> 2
```

Be especially careful with division, though; it can be counter-intuitive:

```
>> 10/4
=> 2
>> 2/3
=> 0
```

Here Ruby uses *integer division*, which returns the number of times the denominator goes into the numerator. When dividing one number by another, chances are you want *floating-point* division instead, which you can get by adding a "point zero" to at least one of the numbers:

```
>> 10/4.0
=> 2.5
>> 2/3.0
=> 0.6666666666666666
```

Many programmers, including me, find it convenient to fire up irb and use it as a simple calculator when the need arises. It's not fancy, but it's quick and

relatively powerful, and the ability to define variables often comes in handy as well.

4.1.1 More Advanced Operations

Ruby supports more advanced mathematical operations via a *module* called `Math`, which has utilities for things like mathematical constants, roots, and trigonometric functions:

```
>> Math::PI
=> 3.141592653589793
>> Math.sqrt(2)
=> 1.4142135623730951
>> Math.cos(2*Math::PI)
=> 1
```

The double colon notation and all-caps in `Math::PI` are characteristic of module *constants* (names that can't change value).

There is one gotcha for those coming from high-school (and even college) textbooks that use ln for the natural logarithm (base *e*). Like mathematicians and most other programming languages, Ruby uses log instead:

```
>> Math.log(Math::E)
1
>> Math.log(10)
2.302585092994046
```

Mathematicians typically indicate base-ten logarithms using $\log_{10}$, and Ruby follows suit with `log10`:

```
>> Math.log10(10)
1
>> Math.log10(1000000)
6
>> Math.log10(Math::E)
0.4342944819032518
```

Finally, Ruby also supports exponentiation via the `**` operator:

```
>> 2**3
=> 8
>> Math::E**100
=> 2.6881171418161212e+43
```

The final result here, using a number followed by `e+43`, is Ruby's way of expressing the scientific notation for $e^{100} \approx 2.6881171418161212 \times 10^{43}$.

The `Math` documentation includes a more comprehensive list of further operations.

4.1.2 Math to String

We discussed in Chapter 3 how to get from strings to arrays (and vice versa) using `split` and `join`. Similarly, Ruby allows us to convert between numbers and strings.

Probably the most common way to convert from a number to a string is using the `to_s` ("to string") method, as we can see with the useful definition shown in Listing 4.1 (Figure 4.1).[1]

[1] The use of τ to represent the circle constant 6.283185 . . . was proposed in a math essay I published in 2010 called *The Tau Manifesto*, which also established a math holiday called Tau Day, celebrated annually on June 28.

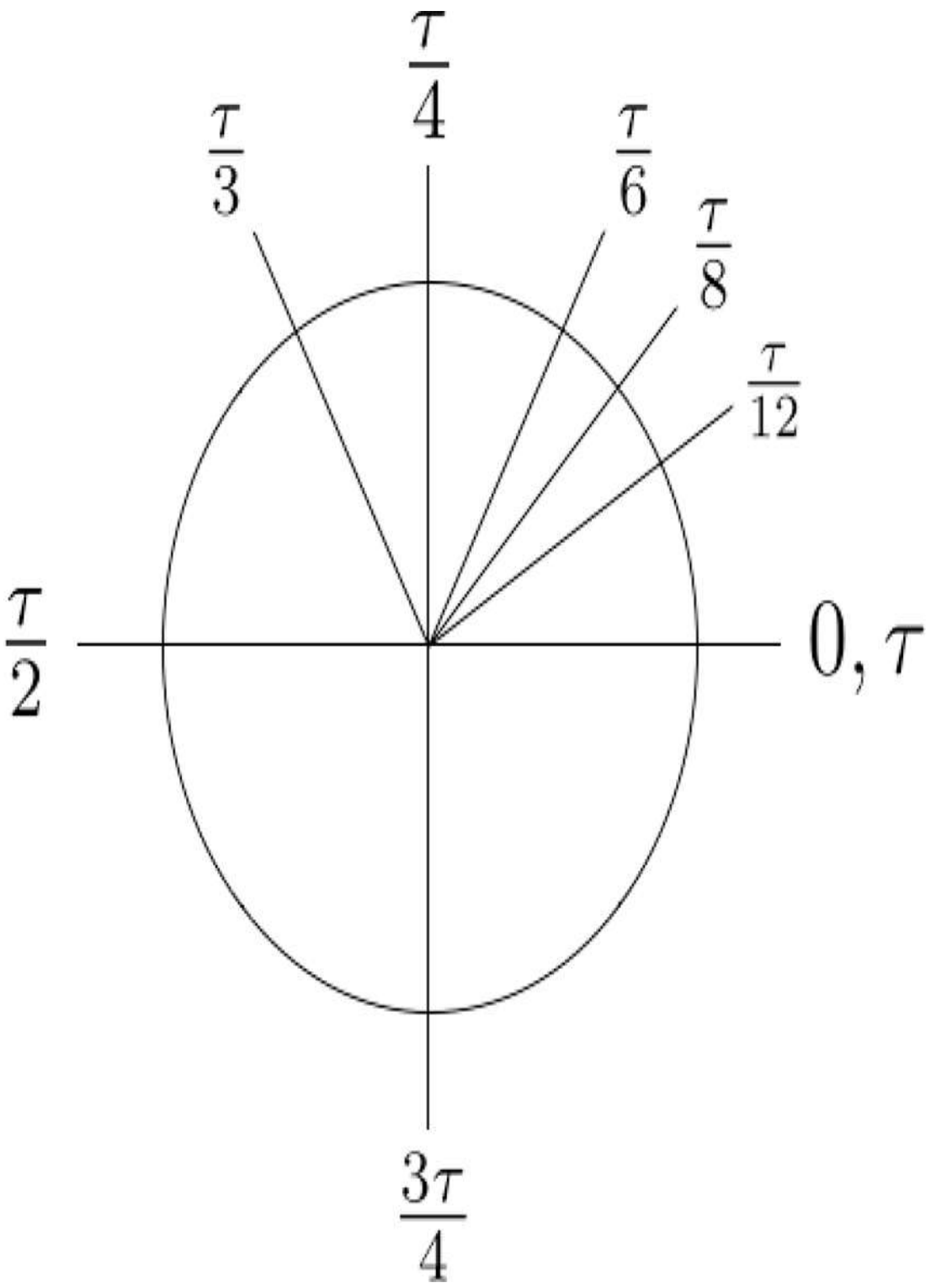


Figure 4.1: Some special angles in terms of $\tau = 2\pi$.

Listing 4.1: Using `tau` for the circle constant.

```
>> tau = 2 * Math::PI
>> tau.to_s
=> "6.283185307179586"
```

The `to_s` method also works on bare numbers:

```
>> 6.283185307179586.to_s
=> "6.283185307179586"
```

To go the other direction, we can use the `to_i` (“to integer”) and `to_f` (“to float”) methods:

```
>> "6.283185307179586".to_f
=> 6.283185307179586
>> "6".to_i
=> 6
```

4.1.3 Exercises

1. What happens when you call `to_f` on the string `"1.24e6"`? What about if you call `to_s` on the result?
2. Like most programming languages, Ruby lacks support for imaginary numbers, i.e., numbers that are a real multiple of the *imaginary unit* i (satisfying the equation $i^2 = -1$, sometimes written as $i=\sqrt{-1}$). What happens if you ask Ruby to calculate the square root of −1?
3. Ruby *constants*, such as `PI`, are written in ALL CAPS. Rewrite Listing 4.1 to use `TAU` in place of `tau`. What happens if you try reassigning some other value to `TAU`?

4.2 Time

Another frequently used built-in object is `Time` (technically a *class*, which is a special kind of object we’ll learn more about starting in Chapter 7). `Time` gives us our first chance to use the `new` method, a so-called *constructor function* that is the standard Ruby way to create a new object. So far, we’ve been able to rely on “literal constructors” like quotes and square brackets, but we can also define things like strings and arrays using `new`:[2]

[2] Some irb implementations may have trouble with the Unicode character in “canal—Panama!”. In particular, I had to follow the instructions in the article “Enable Unicode support in IRB” to get it to display properly.

Click here to view code image

```
>> s = String.new("A man, a plan, a canal—Panama!")
=> "A man, a plan, a canal—Panama!"
>> s.split(", ")
=> ["A man", "a plan", "a canal—Panama!"]
```

and

```
>> a = Array.new
>> a << 3 << 4
=> [3, 4]
>> a << "hello, world!"
=> [3, 4, 'hello, world!']
```

Unlike strings and arrays, times have no literal constructor, so we *have* to use `new` in this case:

```
>> now = Time.new
=> 2018-08-14 19:18:36 -0700
```

When called with no arguments, `Time.new` returns the current time. Ruby supplies `Time.now` as an intuitive synonym for this important special case:

```
>> now = Time.now
=> 2018-08-14 19:18:55 -0700
```

As with other Ruby objects, `Time` objects respond to a variety of methods:

```
>> now.year
=> 2018
>> now.day
=> 14
>> now.month
=> 8
>> now.hour
=> 19
```

It's also possible to initialize `Time` objects with specific dates and times:

Click here to view code image

```
>> moon_landing = Time.new(1969, 7, 20, 20, 17, 40)
=> 1969-07-20 20:17:40 -0700
>> moon_landing.day
=> 20
```

By default, `Time` uses the local time zone, but this introduces weird location dependence to the operations, so it's a good practice to use UTC instead:[3]

[3] For most practical purposes, Coordinated Universal Time (UTC) is the same as Greenwich Mean Time. But why call it UTC? From the NIST Time and Frequency FAQ: **Q:** Why is UTC used as the acronym for Coordinated Universal Time instead of CUT? **A:** In 1970 the Coordinated Universal Time system was devised by an international advisory group of technical experts within the International Telecommunication Union (ITU). The ITU felt it was best to designate a single abbreviation for use in all languages in order to minimize confusion. Since unanimous agreement could not be achieved on using either the English word order, CUT, or the French word order, TUC, the acronym UTC was chosen as a compromise.

Click here to view code image

```
>> moon_landing = Time.utc(1969, 7, 20, 20, 17, 40)
=> 1969-07-20 20:17:40 UTC
```

We can also get the current time in UTC by appending `.utc` to `Time.now`:[4]

[4] This is an example of *method chaining*, which we'll learn more about in Section 5.3.

```
>> now = Time.now.utc
=> 2018-08-15 02:20:31 UTC
```

Finally, `Time` instances can be subtracted from each other:

```
>> now - moon_landing
=> 1548482571.0
```

The result here is the number of seconds since the day and time of the first Moon landing (Figure 4.2).[5] (Your results, of course, will vary, because time marches on, and your value for `Time.now` will differ.)

[5] Image courtesy of Castleski/Shutterstock.

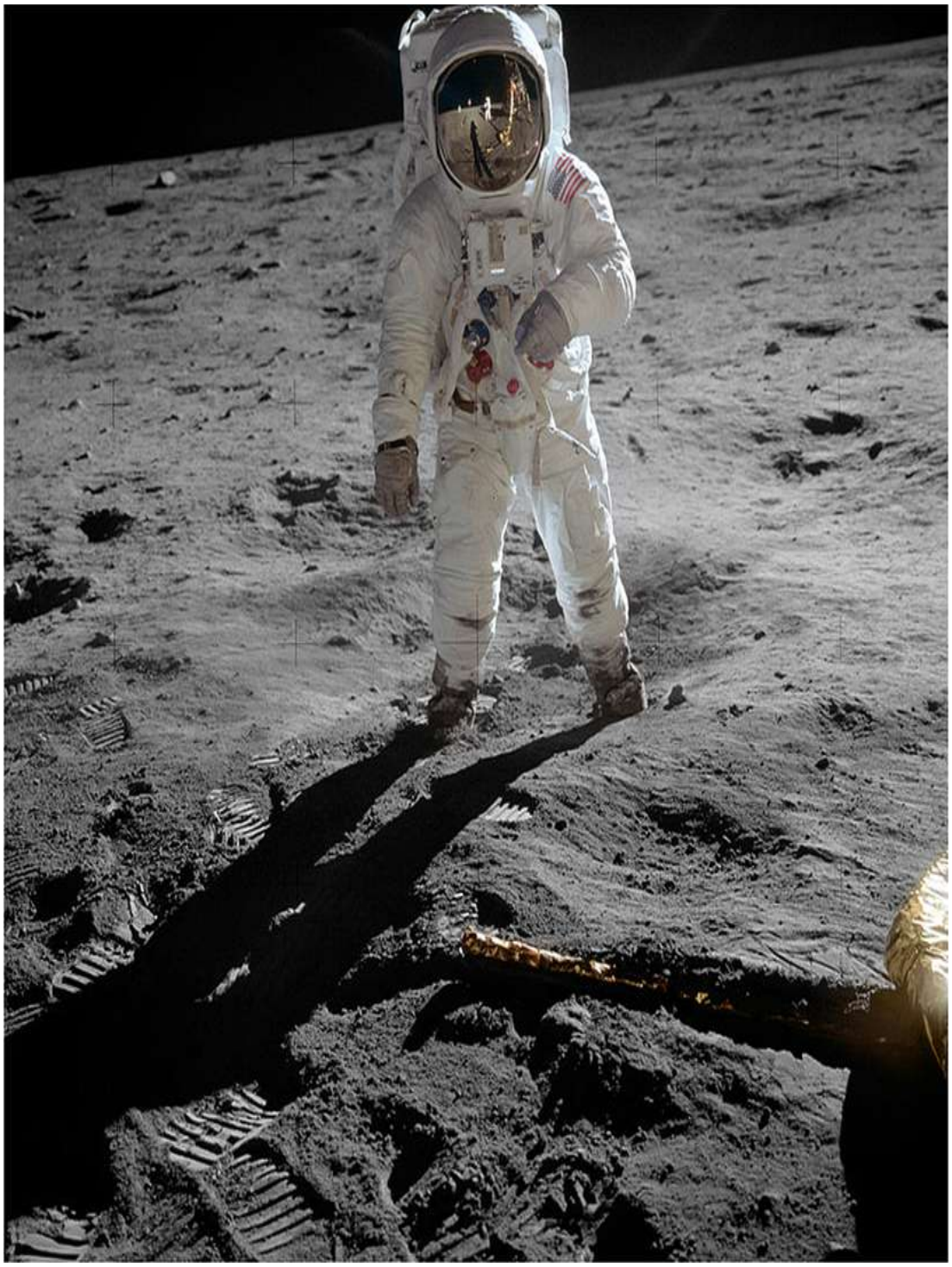


Figure 4.2: Buzz Aldrin and Neil Armstrong somehow got to the Moon (and back!) without Ruby.

You may have noticed that the month and day are returned as *unit-offset* values, which differs from the zero-offset indexing used for arrays (Section 3.2). For example, in the eighth month (August), the return value of `now.month` is `8` rather

than `7` (as it would be if months were being treated like indices of a zero-offset array). There is one important value that is returned as a zero-offset index, though:

Click here to view code image

```
>> moon_landing.wday        # wday = weekday
=> 0
```

Here `wday` is the "weekday" method, and the `0` index indicates that the Moon landing happened on the "zeroth" (first) day of the week.

Even though the official international standard is that Monday is the first day, Ruby follows the American convention of using Sunday instead. We can get the name of the day by making an array of strings for the days of the week (assigned to an ALL CAPS identifier, indicating a constant), and then using `wday` as an index in the array with the square bracket notation (Section 3.1):

Click here to view code image

```
>> DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
?>             "Thursday", "Friday", "Saturday"]
>> DAYNAMES[moon_landing.wday]
=> "Sunday"
>> DAYNAMES[Time.now.wday]
=> "Tuesday"
```

(These day names are actually available as part of the `Date` class via `Date::-DAYNAMES`. You just have to run `require 'date'` to load the class.) Your results for the last line will vary, of course, unless you happen to be reading this on a Tuesday.

As a final exercise, let's update our Sinatra hello app from Listing 1.8 with a greeting including the day of the week. The code appears in Listing 4.2, with the result as shown in Figure 4.3.

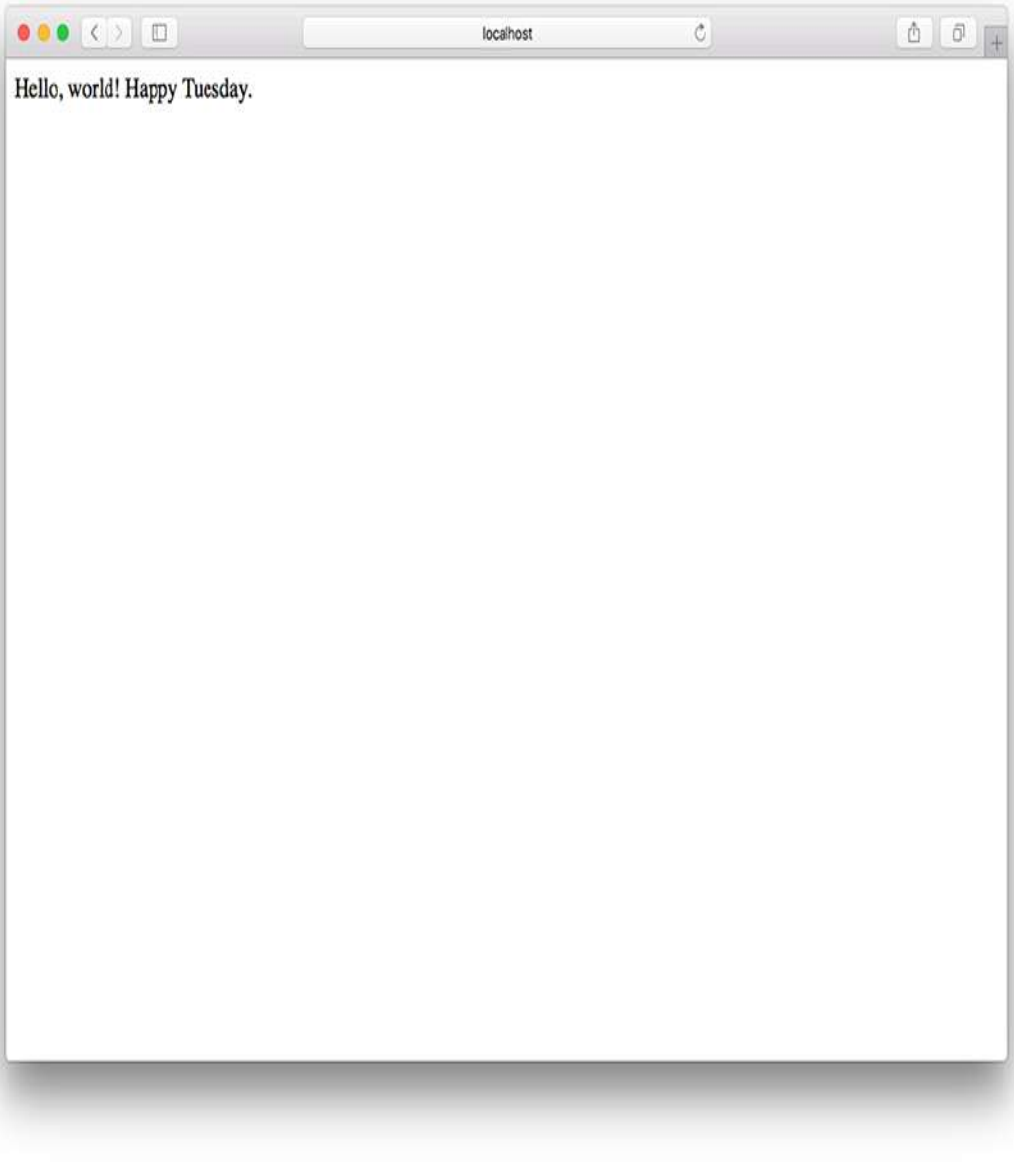


Figure 4.3: A greeting customized just for today.

Listing 4.2: Adding a greeting customized to the day of the week.
hello_app.rb

Click here to view code image

```
require 'sinatra'

get '/' do
  DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"]
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

Note that we've switched the greeting in Listing 4.2 from a single- to a double-quoted string so that we can interpolate the `dayname` value into the greeting.

4.2.1 Exercises

1. Use Ruby to calculate how many seconds after the Moon landing you were born. (Or maybe you were even born *before* the Moon landing—in which case, lucky you! I hope you got to watch it on TV.)
2. Show that Listing 4.2 works even if you pull `DAYNAMES` out of the `get`, as shown in Listing 4.3. (Remember to restart the server.) Then use the `Date` class to eliminate the variable entirely (Listing 4.4). (Note that `date` is automatically required by `sinatra`, so there's no need to include it separately.)

Listing 4.3: Pulling `DAYNAMES` out of `get`.
hello_app.rb

Click here to view code image

```
require 'sinatra'

DAYNAMES =["Sunday", "Monday", "Tuesday", "Wednesday",
           "Thursday", "Friday", "Saturday"]

get '/' do
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

Listing 4.4: Using the built-in `DAYNAMES`.
hello_app.rb

Click here to view code image

```
require 'sinatra'

get '/' do
  dayname = Date::DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

4.3 Regular Expressions

Ruby has full support for *regular expressions*, often called *regexes* or *regexps* for short, which are a powerful mini-language for matching patterns in text. A full mastery of regular expressions is beyond the scope of this book (and perhaps beyond the scope of human ability), but the good news is that there are many resources available for

learning about them incrementally. (Some such resources are mentioned in "Grepping" in *Learn Enough Command Line to Be Dangerous* and "Global find and replace" in *Learn Enough Text Editor to Be Dangerous*.) The most important thing to know about is the general idea of regular expressions; you can fill in the details as you go along.

> Regexes are notoriously terse and error-prone; as programmer Jamie Zawinski famously said:
>
> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Luckily, this situation is greatly ameliorated by web applications like Rubular, which let us build up regexes interactively (Figure 4.4). Moreover, such resources typically include a quick reference to assist us in finding the code for matching particular patterns (Figure 4.5).

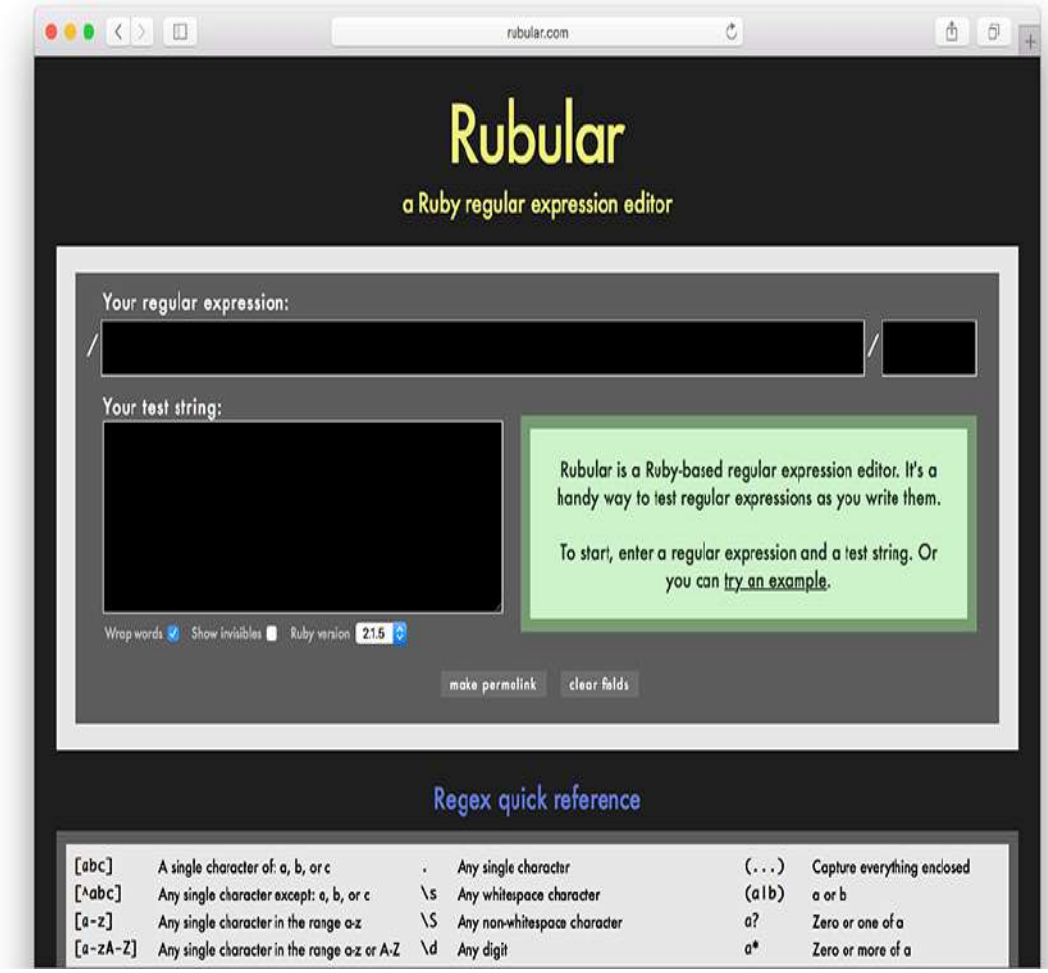


Figure 4.4: An online regex builder.

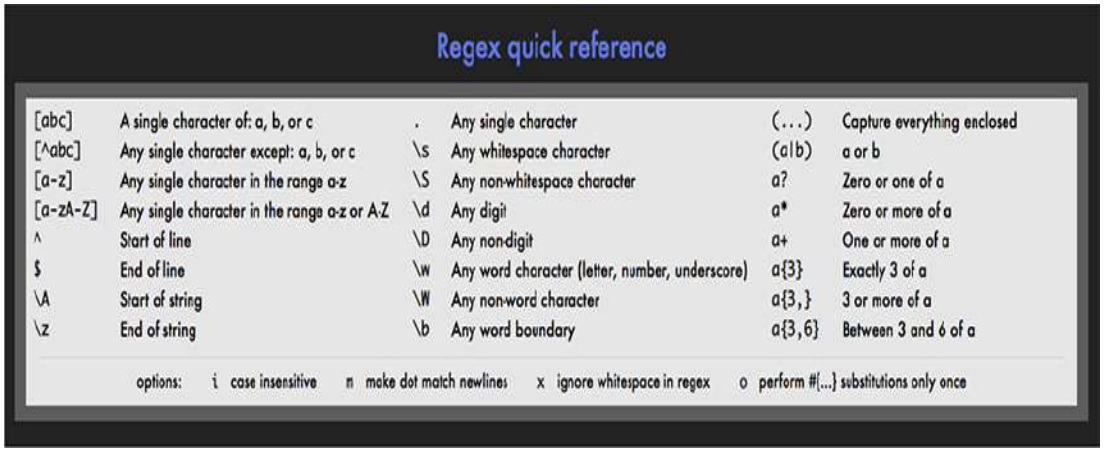


Figure 4.5: A close-up of the regex reference.

Rubular is Ruby-specific, but you can also use a site like the regex101 regex tester used in *Learn Enough JavaScript to Be Dangerous*. Regex101 doesn't have Ruby-specific support, but the "PHP" option should be good enough for most applications. In practice, languages differ little in their implementation of regular expressions, but it's wise to use the correct language-specific settings when available, and always to double-check when moving a regex to a different language.

Let's take a look at some simple regex matches in Ruby. A basic regex consists of a sequence of characters that matches a particular pattern. We can create a new regex using the `new` function (Section 4.2) on the Regexp object, but it's far more common to use the literal constructor `/.../`. For example, here's a regex that matches standard American ZIP codes (Figure 4.6),[6] consisting of five digits in a row:

[6] Image courtesy of 4kclips/123RF.

Figure 4.6: 90210 (Beverly Hills) is one of the most expensive ZIP codes in America.

```
>> zip_code = /\d{5}/
```

If you use regular expressions a lot, eventually you'll memorize many of these rules, but you can always look them up in a quick reference (Figure 4.5).

Now let's see how to tell if a string matches a regex. Strings come with a `match` method that lets us match to a regex:[7]

[7] The same method exists on `Regex`, so in fact you can reverse the order of the regex and string and it will still work. In my experience, the vast majority of Ruby developers use the `String#match` method most of the time.

Click here to view code image

```
>> "no match".match(zip_code)
=> nil
```

```
>> "Beverly Hills 90210".match(zip_code)
=> #<MatchData "90210">
```

The second result here is a somewhat cryptic "`MatchData`" object. In practice, its main use is in boolean contexts, like this:

Click here to view code image

```
>> s = "Beverly Hills 90210"
>> puts "It's got a ZIP code!" if s.match(zip_code)
It's got a ZIP code!
```

Another common and instructive regex operation involves creating an array of *all* the matches. We'll start by defining a longer string, one with two ZIP codes (Figure 4.7):[8]

[8] Image courtesy of kitleong/123RF.

Figure 4.7: 91125 is a dedicated ZIP code for the campus of the California Institute of Technology (Caltech).

Click here to view code image

```
>> s =  "Beverly Hills 90210 was a '90s TV show set in Los Angeles."
>> s += " 91125 is another ZIP code in the Los Angeles area."
=> "Beverly Hills 90210 was a '90s TV show set in Los Angeles. 91125 is another
 ZIP code in the Los Angeles area."
```

You should be able to use your technical sophistication (Box 1.1) to infer what the `+=` operator does here if you haven't seen it before (which might involve doing a quick Google search).

To find out whether the string matches the regex, we can use the `String#-scan` method to find an array of matches:

```
>> s.scan(zip_code)
=> ["90210", "91125"]
```

It's also easy to use a literal regex directly, such as this `scan` to find all multi-letter words that are in ALL CAPS:

```
>> s.scan(/[A-Z]{2,}/)
=> ["TV", "ZIP"]
```

See if you can find the rules in Figure 4.5 used to make the regex above.

4.3.1 Splitting on Regexes

Our final example of regexes combines the power of pattern matching with the `split` method we saw in Section 3.1. In that section, we saw how to split on spaces, like this:

Click here to view code image

```
>> "ant bat cat duck".split(" ")
=> ['ant', 'bat', 'cat', 'duck']
```

We can obtain the same result in a more robust way by splitting on whitespace. Consulting the quick reference (Figure 4.5), we find that the regex for whitespace is `\s`, and the way to indicate "one or more" is the plus sign `+`. Thus, we can split on whitespace as follows:

Click here to view code image

```
>> "ant bat cat duck".split(/\s+/)
=> ["ant", "bat", "cat", "duck"]
```

The reason this is so nice is that now we can get the same result if the strings are separated by multiple spaces, tabs, newlines, etc.:

Click here to view code image

```
>> "ant    bat\tcat\nduck".split(/\s+/)
=> ["ant", "bat", "cat", "duck"]
```

As we saw in Section 3.1, this pattern is so useful that it's actually the default behavior for `split`. When we call `split` with zero arguments, Ruby splits on whitespace automatically:

Click here to view code image

```
>>  "ant    bat\tcat\nduck".split
=> ["ant", "bat", "cat", "duck"]
```

4.3.2 Exercises

1. Write a regex that matches the extended-format ZIP code consisting of five digits, a hyphen, and a four-digit extension (such as 10118-0110).

   Confirm that it works using `String#match` and the caption in Figure 4.8.[9]

   [9] Image courtesy of jordi2r/123RF.

Figure 4.8: ZIP code 10118-0110 (the Empire State Building).

2. Write a regex that splits only on newlines. Such regexes are useful for splitting a block of text into separate lines. In particular, test your regex by pasting the poem in Listing 4.5 into the console and using `sonnet.-split(/your regex/)`. What is the length of the resulting array?

Listing 4.5: Some text with newlines.

Click here to view code image

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
```

```
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."
```

4.4 Hashes

Our final example of a simple Ruby data type is a *hash*, also called an *associative array*. You can think of hashes as being like regular arrays but with generic labels rather than integers as indices, so instead of `array[0] = 0` we could have `hash["name"] = "Michael"`. Each element is thus a pair of values: a label (the *key*) and an element of any type (the *value*). These elements are also known as *key–value pairs*.

The most familiar choice for key labels is strings (Chapter 2); indeed, this is by far the most common choice in languages that support associative arrays. We'll thus start by creating hashes using string keys, but be alert—we'll make a change in short order. As a simple example, let's create an object to store the first and last names of a user, such as we might have in a web application:

Click here to view code image

```
>> user = {}                              # {} is an empty hash.
=> {}
>> user["first_name"] = "Michael"         # Key "first_name", value "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"            # Key "last_name", value "Hartl"
=> "Hartl"
```

As you can see, an empty hash is represented by curly braces, and we can assign values using the same square bracket syntax as for arrays. We can retrieve values in the same way:

Click here to view code image

```
>> user["first_name"]         # Element access is like arrays
=> "Michael"
>> user["last_name"]
=> "Hartl"
>> user["nonexistent"]
=> nil
```

Note in the last example that hashes return `nil` when the key doesn't exist. This behavior is especially convenient in a boolean context, as we'll see in Section 4.5.

Instead of defining hashes one item at a time using square brackets, it's easy to use a literal representation with keys and values separated by `=>`, called a

“hashrocket”:

Click here to view code image

```
>> user
=> {"first_name"=>"Michael", "last_name"=>"Hartl"}
>> moonman = { "first_name" => "Buzz", "last_name" => "Aldrin" }
=> {"first_name"=>"Buzz", "last_name"=>"Aldrin"}
```

Here I’ve used the usual Ruby convention of putting an extra space at the two ends of the hash—a convention ignored by the console output. (I don’t know why the spaces are conventional; probably some early influential Ruby programmer liked the look of the extra spaces, and the convention stuck.)

4.4.1 Symbols

So far we’ve used strings as hash keys, but nowadays it is more common to use *symbols* instead. Symbols look kind of like strings, but are prefixed with a colon instead of surrounded by quotes. For example, `:name` is a symbol. You can think of symbols as being strings without all the extra baggage:[10]

[10] As a result of having less baggage, symbols are easier to compare to each other; strings need to be compared character by character, while symbols can be compared all in one go. This makes them ideal for use as hash keys.

Click here to view code image

```
>> "name".split(' ')
=> ["n", "a", "m", "e"]
>> :name.split(' ')
undefined method `split' for :name:Symbol (NoMethodError)
```

Symbols are a special Ruby data type shared with very few other languages, so they may seem weird at first, but Ruby uses them a lot, so you’ll get used to them fast. Unlike with strings, not all characters are valid, though you can get them to work using quotes if you’re willing to have them kind of look like strings:

Click here to view code image

```
>> :foo-bar
undefined local variable or method `bar' for main:Object (NameError)
>> :2foo
syntax error, unexpected integer literal, expecting literal content or
terminator or tSTRING_DBEG or tSTRING_DVAR (SyntaxError)
>> :"foo-bar"
=> :"foo-bar"
```

```
>> :"2foo"
=> :"2foo"
```

As long as you start your symbols with a letter and stick to normal word characters, you should be fine.

In terms of symbols as hash keys, we can define a `user` hash as follows:

Click here to view code image

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> user[:name]              # Access the value corresponding to :name.
=> "Michael Hartl"
>> user[:password]          # Access the value of an undefined key.
=> nil
```

Because it's so common for hashes to use symbols as keys, Ruby supports a custom syntax just for this special case:

Click here to view code image

```
>> user = { name: "Michael Hartl", email: "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
```

The second syntax replaces the symbol/hashrocket combination with the name of the key followed by a colon and a value:

Click here to view code image

```
{ name: "Michael Hartl", email: "michael@example.com" }
```

This construction more closely follows the hash notation in other languages (such as JavaScript) and enjoys growing popularity in the Ruby community. Because both hash syntaxes are still in common use, though, it's essential to be able to recognize both of them. Unfortunately, this can be confusing, especially since `:name` is valid on its own (as a standalone symbol) but `name:` has no meaning by itself. The bottom line is that `:name =>` and `name:` are effectively the same *only inside literal hashes*, so that

```
{ :name => "Michael Hartl" }
```

and

```
{ name: "Michael Hartl" }
```

are equivalent, but otherwise you need to use `:name` (with the colon coming first) to denote a symbol.

4.4.2 Nested Hashes

Hash values can be virtually anything, even other hashes, as seen in Listing 4.6.

Listing 4.6: Nested hashes.

Click here to view code image

```
>> params = {}        # Define a hash called 'params' (short for 'parameters').
=> {}
>> params[:user] = { name: "Michael Hartl", email: "mhartl@example.com" }
=> {:name=>"Michael Hartl", :email=>"mhartl@example.com"}
>> params
=> {:user=>{:name=>"Michael Hartl", :email=>"mhartl@example.com"}}
>> params[:user][:email]
=> "mhartl@example.com"
```

These sorts of hashes-of-hashes, or *nested hashes*, are heavily used in Ruby web development (as seen in the *Ruby on Rails Tutorial*).

Hash Iteration

As with arrays (Chapter 3), hashes respond to the `each` method. For example, consider a hash named `flash` with keys for two conditions, `:success` and `:danger`:[11]

[11] The `flash` hash is a commonly used part of the Rails framework.

Click here to view code image

```
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", :danger=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :danger has value "It failed."
```

Note that, while the `each` method for arrays takes a block with only one variable, `each` for hashes takes two, a *key* and a *value*. Thus, the `each` method for a hash iterates through the hash one key-value *pair* at a time.

The last example uses the useful `inspect` method, which returns a string with a literal representation of the object it's called on:

Click here to view code image

```
>> puts (1..5).to_a              # Put an array as a string.
1
2
3
4
5
>> puts (1..5).to_a.inspect      # Put a literal array.
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"
```

By the way, using `inspect` to print an object is common enough that there's a shortcut for it, the `p` function:[12]

[12] There's actually a subtle difference, which is that `p` returns the object being printed while `puts` always returns `nil`. (Thanks to reader Katarzyna Siwek for pointing this out.)

Click here to view code image

```
>> p :name                # Same output as 'puts :name.inspect'
:name
```

4.4.3 Exercises

1. Using symbols, define a hash for a `user` with three attributes (keys): `username`, `password`, and `password_confirmation`. How would you test if the password matches the confirmation?

4.5 Application: Unique Words

Let's apply the hashes from Section 4.4 to a challenging exercise, consisting of our longest program so far. Our task is to extract all the unique words in a fairly long piece of text, and count how many times each word appears.

Because the sequence of commands is rather extensive, our main tool will be a Ruby file (Section 1.3), executed using the `ruby` command. (We're not going to make it a self-contained shell script as in Section 1.4 because we don't intend this to be a general-purpose utility program.) At each stage, I suggest using irb to

execute the code interactively if you have any question about the effects of a given command.

Let’s start by creating our file:

```
$ touch count.rb
```

Now fill it with a string containing the text, which we’ll choose to be Shake-speare’s Sonnet 116[13] (Figure 4.9),[14] as borrowed from Listing 4.5 and shown again in Listing 4.7.

[13] Note that in the original pronunciation used in Shakespeare’s time, words like “love” and “remove” rhymed, as did “come” and “doom”.

[14] Image courtesy of psychoshadowmaker/123RF.

Figure 4.9: Sonnet 116 compares love's constancy to the guide star for a wandering bark (ship).

Listing 4.7: Adding some text.
count.rb

Click here to view code image

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."
```

Next, we'll initialize our hash, which we'll call `uniques` because it will have an entry for each unique word in the text:

```
uniques = {}
```

For the purposes of this exercise, we'll define a "word" as a run of one or more *word characters* (i.e., letters or numbers, though there are none of the latter in the present text). This match can be accomplished with a regular expression (Section 4.3), which includes a pattern (`\w`) for exactly this case (Figure 4.5):

```
words = sonnet.scan(/\w+/)
```

This uses the `scan` method from Section 4.3 to return an array of all the strings that match "one or more word characters in a row". (Extending this pattern to include apostrophes (so that it matches, e.g., "wand'ring" as well) is left as an exercise (Section 4.5.1).)

At this point, the file should look like Listing 4.8.

Listing 4.8: Adding an object and the matching words.
count.rb

Click here to view code image

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
```

```
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."

uniques = {}
words = sonnet.scan(/\w+/)
```

Now for the heart of our program. We're going to iterate through the `words` array (using the `each` method seen in Listing 3.4) and do the following:

1. If the word already has an entry in the `uniques` object, increment its count by `1`.
2. If the word doesn't have an entry yet in `uniques`, initialize it to `1`.

The result, using the `+=` operator we met briefly in Section 4.3, looks like this:

```
words.each do |word|
  if uniques[word]
    uniques[word] += 1
  else
    uniques[word]  = 1
  end
end
```

Note that we're relying on `uniques[word]` being `nil` (`false` in a boolean context) if `word` isn't yet a valid key.

Finally, we'll print out the result to the terminal:

```
puts uniques
```

The full program (with added comments) appears as in Listing 4.9.

Listing 4.9: A program to count words in text.
count.rb

Click here to view code image

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
```

```
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."

# Unique words
uniques = {}
# All words in the text
words = sonnet.scan(/\w+/)

# Iterate through `words` and build up a hash of unique words.
words.each do |word|
  if uniques[word]
    uniques[word] += 1
  else
    uniques[word]  = 1
  end
end

puts uniques
```

The result of running `count.rb` in the terminal looks something like this:

Click here to view code image

```
$ ruby count.rb
{"Let"=>1, "me"=>2, "not"=>4, "to"=>4, "the"=>4, "marriage"=>1,
"of"=>2, "true"=>1, "minds"=>1, "Admit"=>1, "impediments"=>1,
"Love"=>3, "is"=>4, "love"=>1, "Which"=>1, "alters"=>2, "when"=>1,
```

Click here to view code image

```
"it"=>3, "alteration"=>1, "finds"=>1, "Or"=>1, "bends"=>1, "with"=>2,
"remover"=>1, "remove"=>1, "O"=>1, "no"=>2, "an"=>1, "ever"=>2,
"fixed"=>1, "mark"=>1, "That"=>1, "looks"=>1, "on"=>1, "tempests"=>1,
"and"=>4, "never"=>2, "shaken"=>1, "It"=>1, "star"=>1, "every"=>1,
"wand"=>1, "ring"=>1, "bark"=>1, "Whose"=>1, "worth"=>1, "s"=>4,
"unknown"=>1, "although"=>1, "his"=>3, "height"=>1, "be"=>2,
"taken"=>1, "time"=>1, "fool"=>1, "though"=>1, "rosy"=>1,
"lips"=>1, "cheeks"=>1, "Within"=>1, "bending"=>1, "sickle"=>1,
"compass"=>1, "come"=>1, "brief"=>1, "hours"=>1, "weeks"=>1,
"But"=>1, "bears"=>1, "out"=>1, "even"=>1, "edge"=>1, "doom"=>1,
"If"=>1, "this"=>1, "error"=>1, "upon"=>1, "proved"=>1, "I"=>1,
"writ"=>1, "nor"=>1, "man"=>1, "loved"=>1}
```

4.5.1 Exercises

1. Extend the regex used in Listing 4.9 to include an apostrophe, so it matches, e.g., “wand’ring”. *Hint*: Combine the first reference regex at Rubular (Figure 4.10) with `\w`, an apostrophe, and the plus operator `+`.

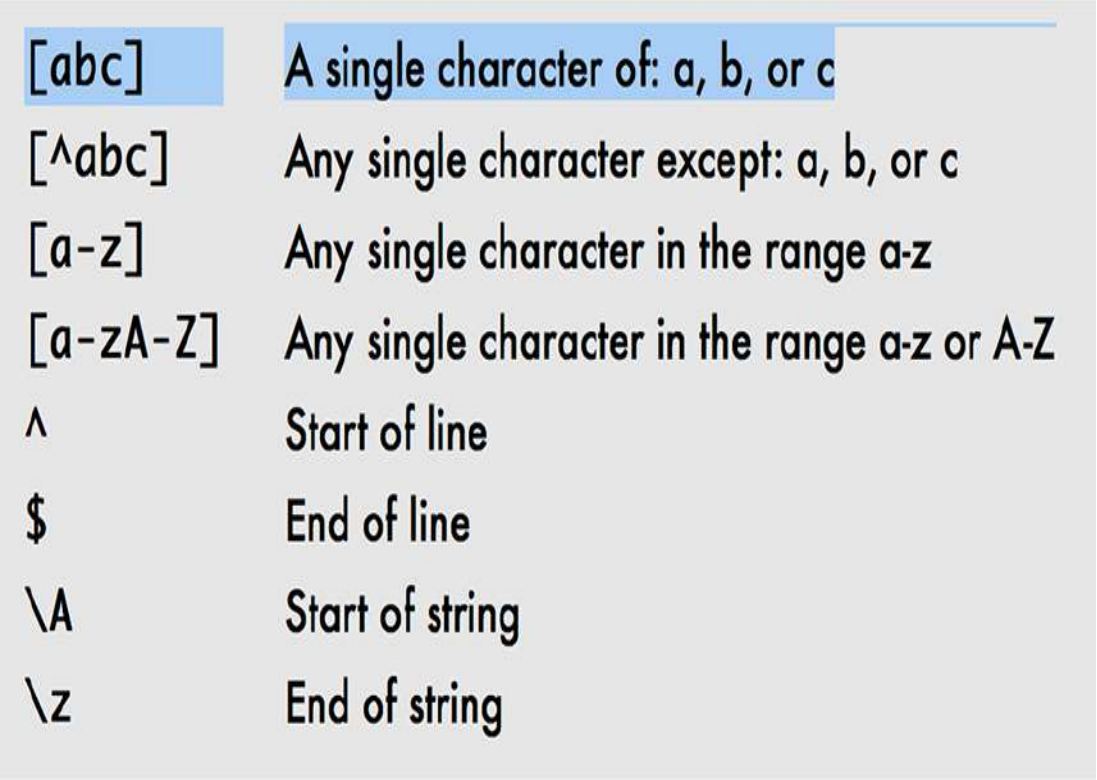


Figure 4.10: An exercise hint.

Chapter 5

Functions and Blocks

So far in this tutorial, we've seen several examples of Ruby functions, blocks, and methods. In this chapter, we'll learn how to define our own functions, we'll deepen our understanding of blocks, and we'll learn how to *chain* methods. As we'll see, the three are closely related, and in fact all of them are variations on the same basic functional theme—one of the most important ideas in Ruby, and indeed in all of computing (Figure 5.1).

Figure 5.1: Time to level up.

5.1 Function Definitions

As we saw in Section 1.2, function calls in Ruby consist of a *name* and zero or more arguments:

Click here to view code image

```
>>puts "hello, world!"
hello, world!
```

Although it's conventional with `puts` to omit parentheses, as with all Ruby functions it works either way:

Click here to view code image

```
>> puts("hello, world!")
hello, world!
```

As discussed in Section 2.5, functions attached to objects (such as `empty?` in `"badger".empty?`) are also called *methods*. We'll learn how to define methods of our own in Chapter 7.

One of the most important tasks in programming involves defining our own functions, which in Ruby can be done using the `def` keyword. Let's take a look at a simple example in the REPL. We'll define a function called `day_of_-the_week` that takes a single `Time` argument (Section 4.2) and returns the day of the week represented by the time.

Recall from Section 4.2 that a `Time` object has a method called `wday` representing the (zero-offset) index of the day of the week:

Click here to view code image

```
>> now = Time.now
>> now.wday
=> 4
```

In that same section, we mentioned briefly that the closely related `Date` library defines a constant for the days of the week:

Click here to view code image

```
>> require 'date'
>> Date::DAYNAMES
=> ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```

This allows us to find the day of the week as follows:

Click here to view code image

```
>> Date::DAYNAMES[now.wday]
=> "Thursday"
```

It would be convenient to *encapsulate* this definition and logic in a `day_-of_the_week` function, so that we could write

```
day_of_the_week(Time.now)
```

By combining the elements above, we can accomplish this as follows:

Click here to view code image

```
>> def day_of_the_week(time)
?>   return Date::DAYNAMES[time.wday]
>> end
```

We see here that a Ruby function starts with the `def` keyword followed by the function name and any arguments; then, there's the function *body*, which determines the return value of the function; finally, the function is ended by the `end` keyword (roughly the equivalent of a closing curly brace `}` in a C-like language like JavaScript). We can test it as follows:

Click here to view code image

```
>> day_of_the_week(Time.now)
=> "Thursday"
```

This might not seem like a big improvement, but note that it's conceptually simpler because we don't have to think about the implementation (i.e., finding the element of an array corresponding to the value of `wday`). This sort of *abstraction layer* between function name and implementation is useful even if the function definition is only a single line. (Indeed, I consider single-line functions to be a sign of good program design.)

In the definition above, we see that, as in many other languages, Ruby supports returning a value via the `return` keyword, but in fact the `return` is unnecessary. Far more common is using Ruby's *implicit return*, whereby the last expression evaluated in the body of the function is returned automatically. This means we can rewrite the `day_of_the_week` function as follows:

Click here to view code image

```
>> def day_of_the_week(time)
?>   Date::DAYNAMES[time.wday]
>> end
```

We'll put this function to good use in Section 5.2 to simplify the customized greeting in our hello app (Listing 4.2).

5.1.1 Exercises

1. Using `def`, define a `square` function that returns the square of a number.

5.2 Functions in a File

Although defining functions in a REPL is convenient for demonstration purposes, it's a bit cumbersome, and a better practice is to put them in a file (as we did with the script in Section 4.5). We'll start by moving the function defined in Section 5.1 into `hello_app.rb`, and we'll then move it to an even more convenient external file.

Let's recall the current state of our hello application, which looks like Listing 5.1.

Listing 5.1: The current state of our hello app. *hello_app.rb*

Click here to view code image

```
require 'sinatra'
get '/' do
  DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"]
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

Our first step is to put the function definition from Section 5.1 into this file, as shown in Listing 5.2.

Listing 5.2: Adding a function for the day of the week. *hello_app.rb*

Click here to view code image

```
require 'sinatra'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end

get '/' do
  DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"]
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

Then, we can edit the body of `get` down to a single line, as shown in Listing 5.3.

Listing 5.3: Replacing the greeting. *hello_app.rb*

Click here to view code image

```
require 'sinatra'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end

get '/' do
  "Hello, world! Happy #{day_of_the_week(Time.now)}."
end
```

At this point, you should be able to confirm that the app is working:

```
$ ruby hello_app.rb
```

The result appears in Figure 5.2.

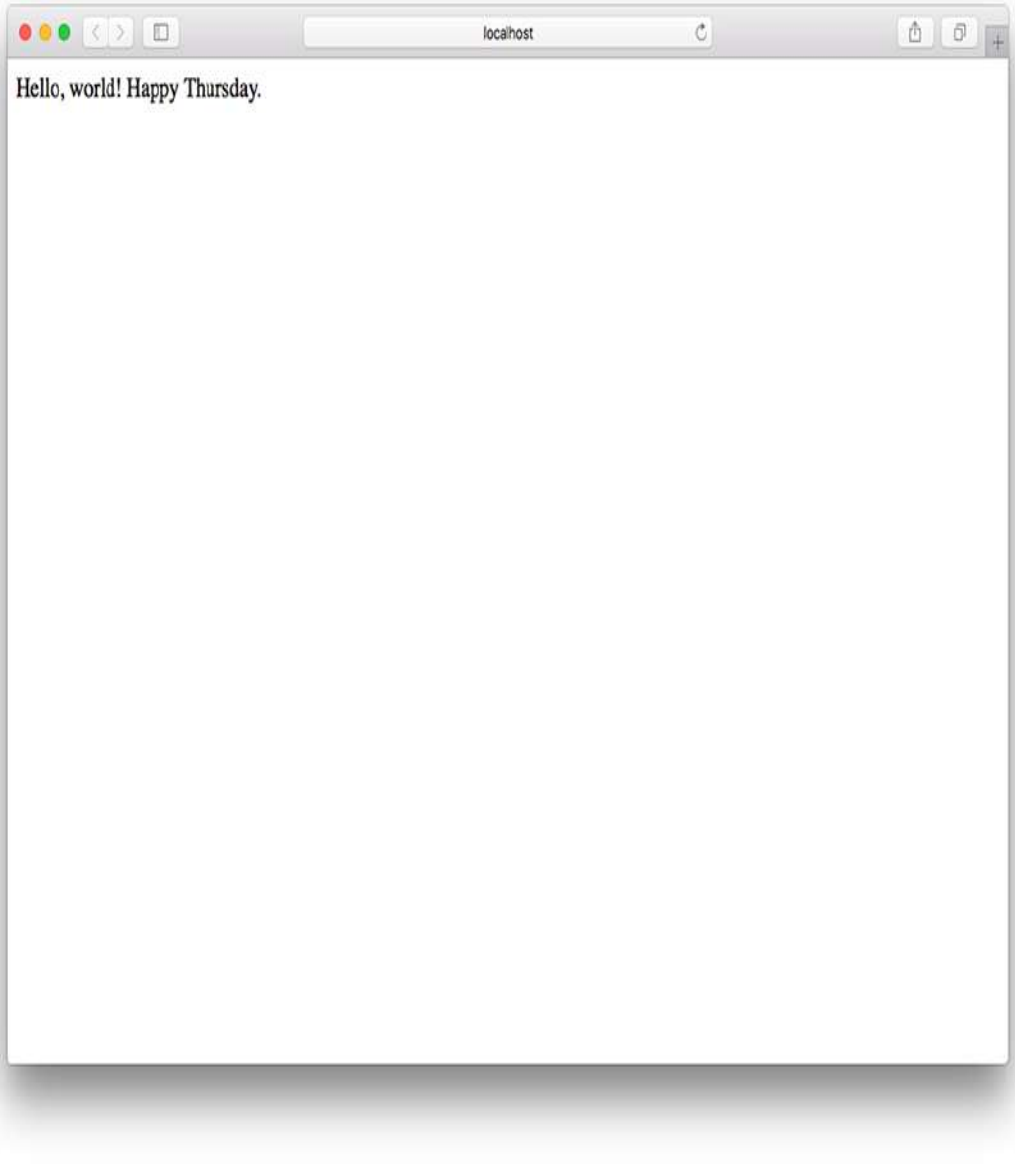


Figure 5.2: The result of a functional greeting.

We can make the code in Listing 5.3 even cleaner by factoring the `day_of_the_week` function into a separate file and then including it into our app. We'll start by cutting the function and pasting it into a new file, `day.rb`:

```
$ touch day.rb
```

The resulting files appear as in Listing 5.4 and Listing 5.5.[1] Note that we've slightly updated the greeting in Listing 5.5 so that we can tell our new code is actually

working. In particular, recall from Section 1.5 that you have to restart the Sinatra server to see updates to the site.

[1] In some editors, you can use Shift-Command-V to paste in a selection using the local indentation level, which saves us the trouble of dedenting it by hand.

Listing 5.4: The `day_of_the_week` function in a file. *day.rb*

Click here to view code image

```
require 'date'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end
```

Listing 5.5: Our greeting with the function cut. *hello_app.rb*

Click here to view code image

```
require 'sinatra'

get '/' do
  "Hello, world! Happy #{day_of_the_week(Time.now)}—now from a file!"
end
```

As you can verify by restarting the server and reloading the browser, the app doesn't work—it crashes immediately, and all we get is the Sinatra error page (Figure 5.3), which indicates that there was an *exception* of type `NoMethod-Error`. (Exceptions are simply a standardized way of indicating particular kinds of errors in a program.) We can find out more about what went wrong by looking at the error message, which indicates that the `day_of_the_week` method isn't defined; zooming in on the message, we see that it even tells us the exact line that has the problem (Figure 5.4).

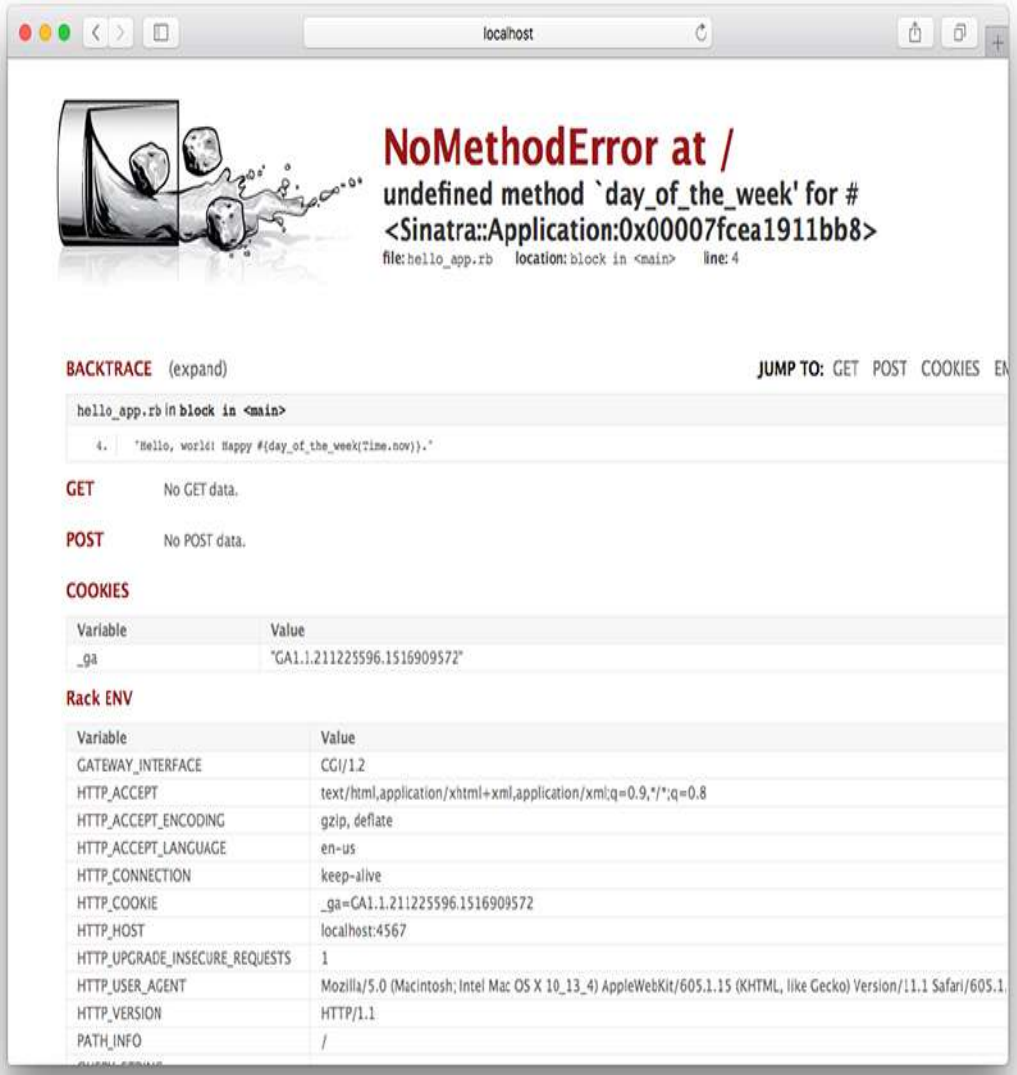


Figure 5.3: A sure sign our app isn't working.

Figure 5.4: Using the Sinatra crash page to find an error.

This practice is a powerful debugging technique: If your Ruby program crashes, inspecting the error message should be your method of first resort. Moreover, if you can't see right away what went wrong, Googling the error message will often yield useful results (Box 5.1).

Box 5.1. Debugging Ruby

One skill that's an essential part of technical sophistication is *debugging*: the art of finding and correcting errors in computer programs. While there's no substitute for experience, here are some techniques that should give you a leg up when tracking down the inevitable glitches in your code:

- *Trace the execution with* `puts`. When trying to figure out why a particular program is going awry, it's often helpful to display variable values with temporary `puts` statements, which can be removed when the bug is fixed. This technique is especially useful when combined with the `inspect` method, which returns a literal representation of the object (Section 4.4.2), as in `puts array.inspect` (which can actually be written using the `p` function as `p array`).
- *Stop the execution with* `raise`. The `raise` function, which raises an exception, is an especially heavy-handed technique because it stops the program execution entirely—but this is sometimes exactly what you want. Even more than with `puts`, using `inspect` is highly recommended, as in `raise array.inspect`.
- *Comment out code.* It's sometimes a good idea to comment out code you suspect is unrelated to the problem to allow you to focus on the code that isn't working.
- *Use the REPL.* Firing up irb and pasting in the problematic code is frequently an excellent way to isolate the problem. (A more advanced version of this technique is *Pry*, which lets you run irb right inside your program.)
- *Google it.* Googling error messages or other search terms related to the bug (which often leads to helpful threads at Stack Overflow) is an essential skill for every modern software developer (Figure 5.5).

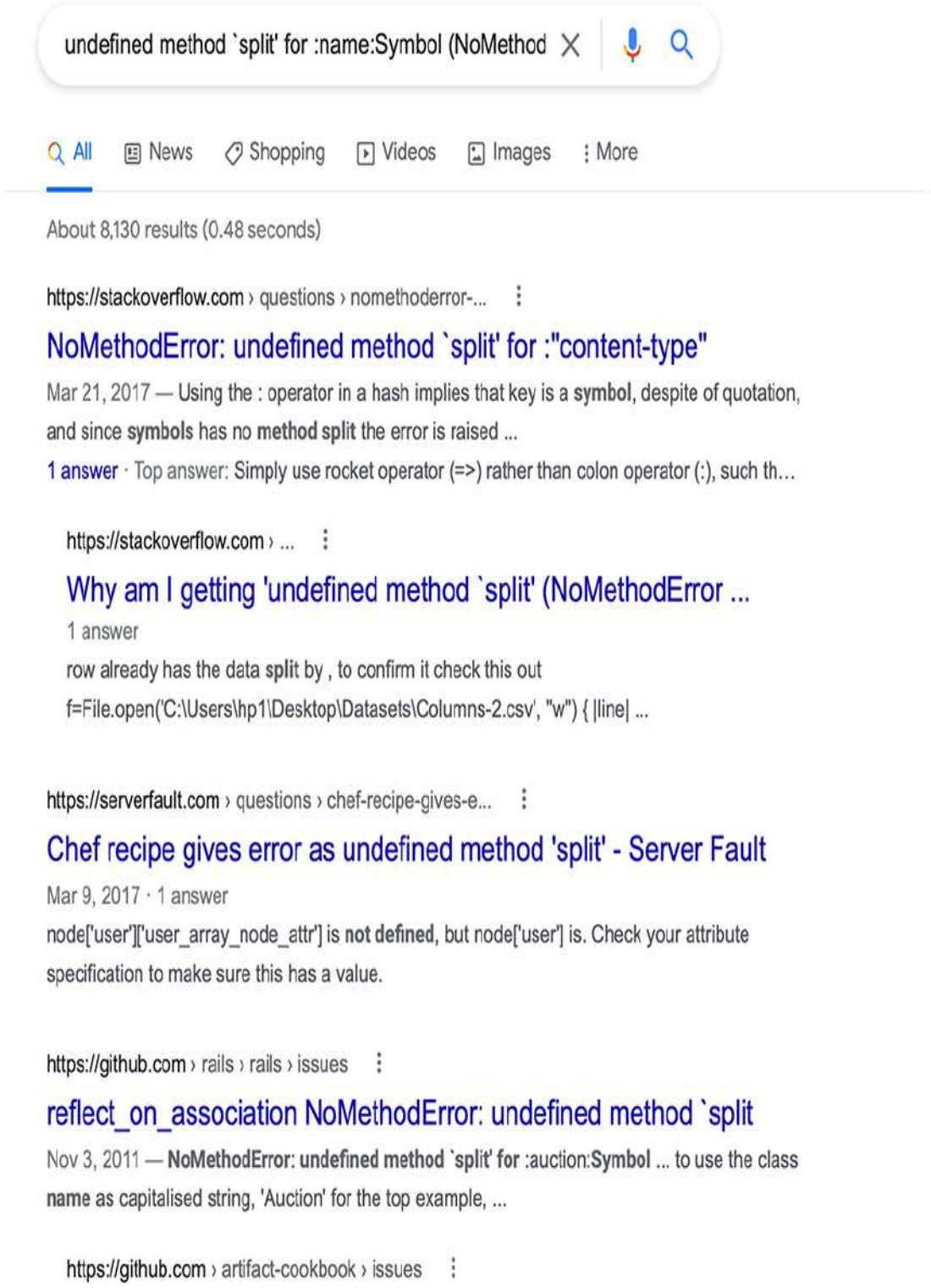


Figure 5.5: How did people ever debug before Google?

The reason for the crash is that we've removed `day_of_the_week` from `hello_app.rb`, so naturally our app has no idea what it is. The solution is to `require` it in much the

same way that we required `sinatra`, as shown in Listing 5.6. Note that the `require` in Listing 5.5 includes the path to the file relative to the current directory (`./day`), which is necessary because our project directory isn't on the Ruby include path by default.[2]

[2] How would you figure out how to add the current directory to the load path? Here's how I'd do it: ruby add to load path.

Listing 5.6: Using a function from an external file. *hello_app.rb*

Click here to view code image

```
require 'sinatra'
require './day'

get '/' do
  "Hello, world! Happy #{day_of_the_week(Time.now)}—now from a file!"
end
```

At this point, the app is working! The result should look something like Figure 5.6.

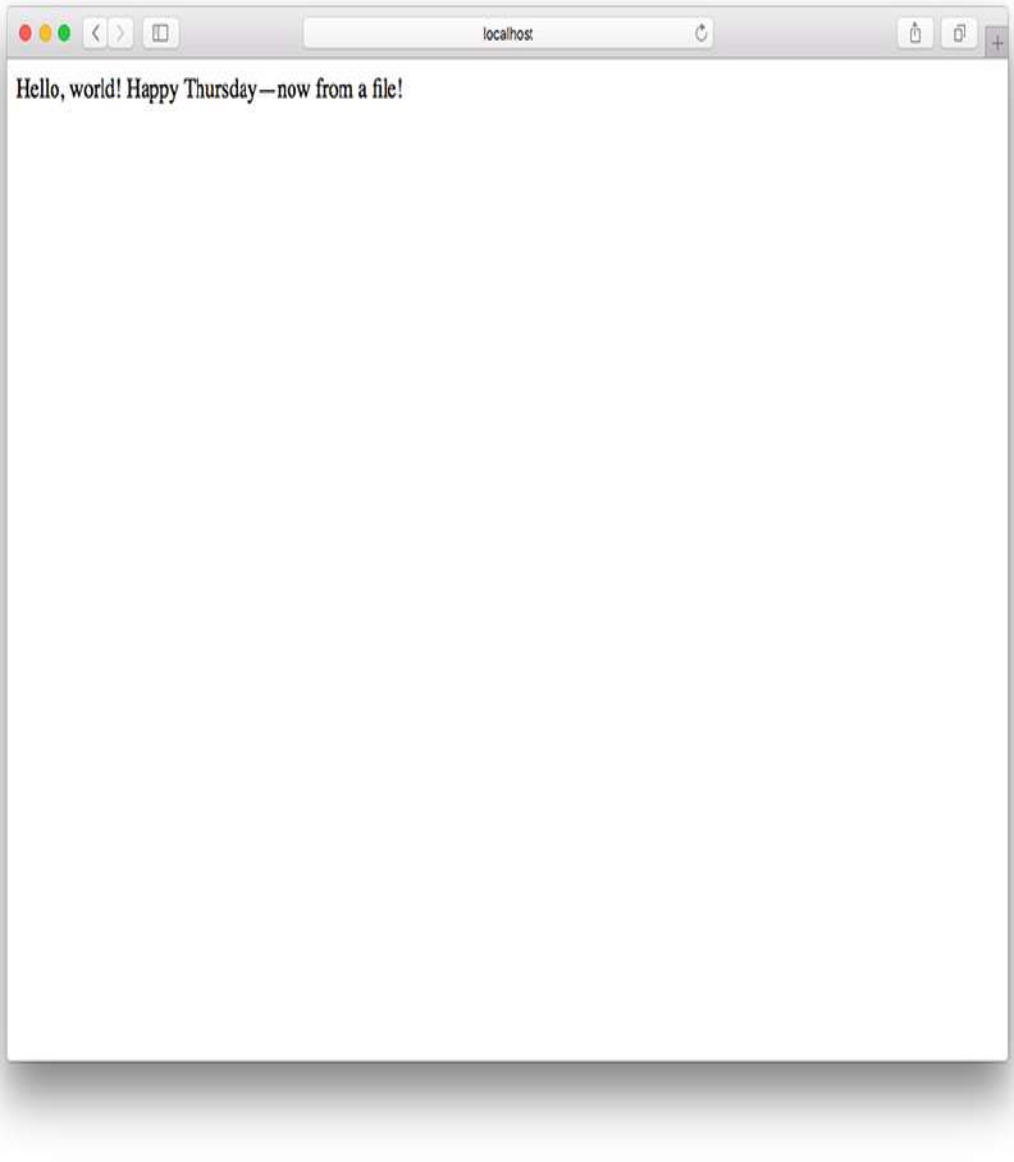


Figure 5.6: An updated greeting.

5.2.1 Exercises

1. Let's replace the interpolated string in Listing 5.5 with a `greeting` function in `day.rb`. Fill in the code in Listing 5.7 to get Listing 5.8 to work.

Listing 5.7: Defining a `greeting` function.*day.rb*

Click here to view code image

```
require 'date'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end

# Returns a friendly greeting.
def greeting(time)
  # FILL IN
end
```

Listing 5.8: Using the `greeting` function.*hello_app.rb*

Click here to view code image

```
require 'sinatra'
require './day'

get '/' do
  greeting(Time.now)
end
```

5.3 Method Chaining

In this section, we'll start developing the palindrome theme mentioned in the introduction. Our goal is to write a function called `palindrome?` that returns `true` if its argument is the same forward and backward, and `false` otherwise.

We can express the simplest possible definition of a palindrome as "a string and the string reversed are the same." (We'll steadily expand this definition over time.) In order to do this, we need to be able to reverse a string.

One way to do this is with *method chaining*, which involves calling a series of messages on a particular object. We saw in Section 3.1 that we can form an array of characters from a string by splitting on the empty string:

Click here to view code image

```
>> "racecar".split("")
=> ["r", "a", "c", "e", "c", "a", "r"]
```

We saw in Section 3.4.1 how to reverse an array:

Click here to view code image

```
>> a = [ 17, 42, 8, 99 ]
>> a.reverse
=> [99, 8, 42, 17]
```

Finally, we learned in Section 3.4.3 that the `join` method effectively undoes a split:

Click here to view code image

```
>> ["r", "a", "c", "e", "c", "a", "r"].join
=> "racecar"
```

This discussion suggests reversing a string with method chaining as follows:

Click here to view code image

```
>> "Racecar".split("").reverse.join
=> "racecaR"
```

We can also simplify it slightly by using the `String#chars` method, which simply returns a string's characters as an array:

Click here to view code image

```
>> "Racecar".chars.reverse.join
=> "racecaR"
```

This method is perfectly valid, and indeed is necessary in languages (like JavaScript) that don't have native support for reversing strings. As it happens, though, Ruby *does* have native support for string reversal, via the `String#-reverse` method:

Click here to view code image

```
>> "Racecar".reverse
=> "racecaR"
```

We can still benefit from method chaining, though, because we'd like to detect palindromes like "Racecar", so we need to combine reversing and *downcasing* strings (seen before in Section 2.5):

Click here to view code image

```
>> "Racecar".downcase.reverse
=> "racecar"
```

With this in our toolkit, we're ready to write the first version of our palindrome method.

Let's put our function for detecting palindromes into its own file, which we'll call `palindrome.rb`:

```
$ touch palindrome.rb
```

What should we call the palindrome-detecting function? Well, the palindrome detector should take in a string and return `true` when the string is a palindrome and `false` otherwise. This makes it a boolean method. Recall from Section 2.5 that boolean methods in Ruby generally end in question marks, which suggests the definition in Listing 5.9. (We'll add the `downcase` refinement in a moment.)

Listing 5.9: Our initial `palindrome?` function.*palindrome.rb*

Click here to view code image

```
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  string == string.reverse
end
```

The code in Listing 5.9 uses the `==` comparison operator (Section 2.4) and Ruby's implicit return (Section 5.1) to automatically return the right boolean value.

We can test the code in Listing 5.9 by *loading* the palindrome file into irb:

```
>> load './palindrome.rb'
```

(Using `load` is better than using `require` as in Listing 5.6 because it allows us to *re*-load files if we make changes, which `require` doesn't let you do.) This makes `palindrome?` available in the REPL:

Click here to view code image

```
>> palindrome?("racecar")
=> true
>> palindrome?("Racecar")
=> false
```

As seen in the second example, our palindrome detector says "Racecar" isn't a palindrome, so to make our detector more general we can `downcase` the string

before the comparison. A working version appears in Listing 5.10.

Listing 5.10: Detecting palindromes independent of case. *palindrome.rb*

Click here to view code image

```
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  string.downcase == string.downcase.reverse
end
```

Returning to irb, we can reload the detector and apply it as follows:

Click here to view code image

```
>> load './palindrome.rb'
>> palindrome?("Racecar")
=> true
```

Success!

As a final refinement, let's follow the Don't Repeat Yourself (or "DRY") principle and eliminate the duplication in Listing 5.10. Inspecting the code, we see that `string.downcase` gets used twice, which suggests declaring a variable (which we'll called `processed_content`) to represent the actual string that gets compared to its own reverse (Listing 5.11).

Listing 5.11: Eliminating some duplication. *palindrome.rb*

Click here to view code image

```
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  processed_content = string.downcase
  processed_content == processed_content.reverse
end
```

Listing 5.11 eliminates one call to `downcase` at the cost of an extra line, so it's not obviously better than Listing 5.10, but we'll see starting in Chapter 8 that having a separate variable gives us much greater flexibility in detecting more complex palindromes.

As a final step, we should check that the `palindrome?` method is still working as advertised:

Click here to view code image

```
>> load './palindrome.rb'
>> palindrome?("Racecar")
=> true
>> palindrome?("Able was I ere I saw Elba")
=> true
```

As you might guess, confirming such things by hand quickly gets tedious, and we'll see in Chapter 8 how to write *automated tests* to check our code's behavior automatically.

5.3.1 Exercises

1. Using method chaining, write a function `email_parts` (Listing 5.12) to return an array of the username and domain for a standard email address of the form `username@example.com`. *Note*: Make sure your function returns the same result for `USERNAME@EXAMPLE.COM`.
2. Using irb, determine whether or not your system supports using the `palindrome?` function from Listing 5.10 on emojis. (You may find the Emojipedia links to the racing car and fox face emojis helpful.) If your system supports emojis in this context, the result should look something like Figure 5.7. (Note that an emoji is a "palindrome" if it's the same when you flip it horizontally, so the fox-face emoji is a palindrome but the racecar emoji isn't, even though the *word* "racecar" *is* a palindrome.)

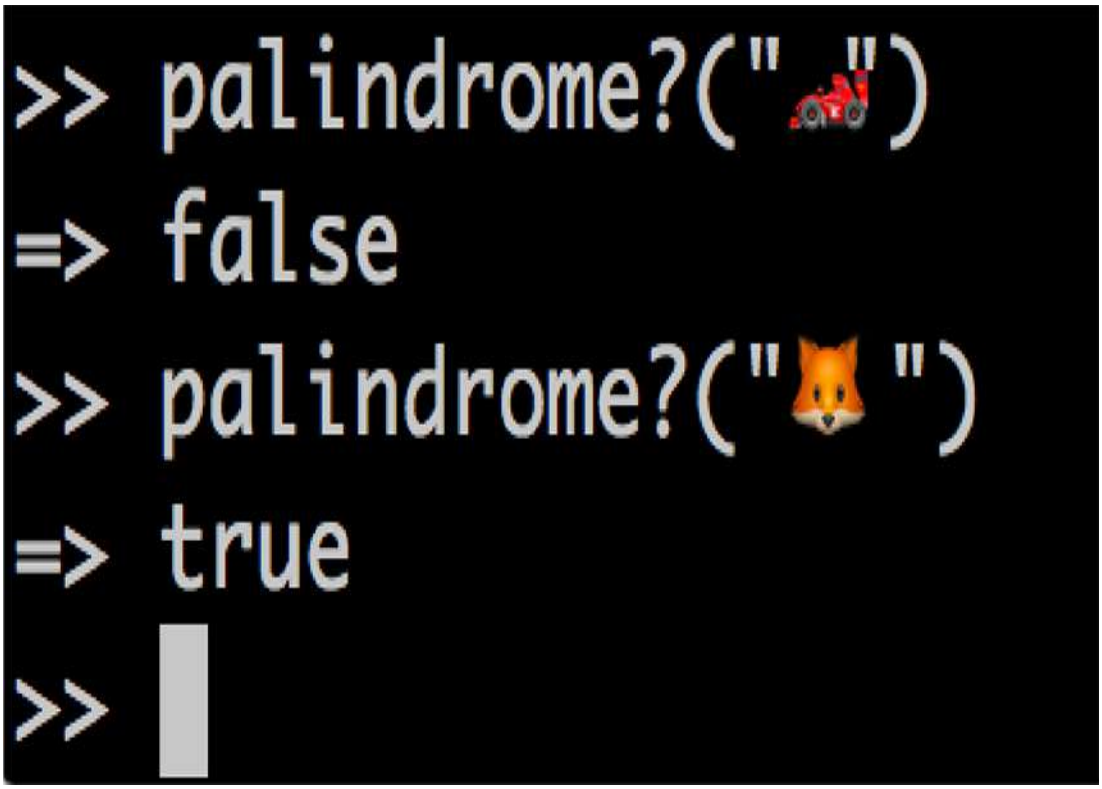


Figure 5.7: Detecting palindromic emojis.

Listing 5.12: Returning the parts of an email.

Click here to view code image

```
>> def email_parts(email)
?>   # FILL IN
>> end
```

5.4 Blocks

Blocks are one of the most characteristic and useful constructs in Ruby. So far we've seen examples in Section 1.5, Section 3.5, and Section 4.4; in this section, we'll look at blocks in more detail and gain a deeper understanding of how they work.

Let's start with a simple block that prints out the first five powers of 2:

Click here to view code image

```
>> (1..5).each { |i| puts 2**i }
2
4
8
```

```
16
32
```

This code calls the `each` method on the range `(1..5)` and passes it the block `{ |i| puts 2**i }`. The vertical bars around the variable name in `|i|` are the Ruby syntax for a *block variable*, and it's up to the method to know what to do with the block. In this case, the range's `each` method can handle a block with a single local variable, which we've called `i`, and it just executes the block for each value in the range.

Curly braces are one way to indicate a block, but there is a second way as well:

Click here to view code image

```
>> (1..5).each do |i|
?>   puts 2**i
>> end
2
4
8
16
32
=> 1..5
```

This is the version of the block syntax we've seen before, in things like

Click here to view code image

```
get '/' do
  'hello, world!'
end
```

(Listing 1.8) and

Click here to view code image

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

(Listing 3.4).

Blocks can be more than one line, and often are. In this tutorial, we'll follow the common convention of using curly braces only for short one-line blocks and the `do..end` syntax for some one-liners and for all multiline blocks:

Click here to view code image

```
>> (1..5).each do |number|
?>   puts 2 ** number
>>   puts '--'
>> end
2
--
4
--
8
--
16
--
32
--
```

Here I've used `number` in place of `i` just to emphasize that any variable name will do.

One way to think about blocks is that they are *anonymous (unnamed) functions* that we can create right when they're needed. If you've studied JavaScript (as in *Learn Enough JavaScript to Be Dangerous*), you might recognize this usage in code like Listing 5.13, which includes an explicit unnamed `function` to print out the elements of a JavaScript array.

Listing 5.13: A `forEach` loop and an anonymous function in JavaScript.

Click here to view code image

```
> ["ant", "bat", "cat", 42].forEach(function(element) {
  console.log(element);
});
ant
bat
cat
42
```

A Ruby block can be seen as a way to do the same thing as Listing 5.13 without having to include an explicit function, as seen in Listing 5.14. The result is code that feels more like a natural and integrated part of the language (or, at least, that is the practical experience of many Ruby developers).

Listing 5.14: An `each` loop with a block.

Click here to view code image

```
>> ["ant", "bat", "cat", 42].each do |element|
?>   puts element
>> end
ant
bat
cat
42
```

Although `each` is probably the most commonly used method that takes a block, there are many others. For example, the `times` method repeats the given block a certain number of times:

Click here to view code image

```
>> 3.times { puts "Betelgeuse!" }    # `times` takes a block with no variables.
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
```

We'll see several more examples in Chapter 6.

5.4.1 Yield

We can develop our understanding of blocks by using them in some of our own functions. The key observations are that (1) *every* Ruby method can take a block and (2) we can invoke the block using the `yield` keyword. Let's take a look at some concrete examples to see how this works.[3]

[3] Several of the examples in this section are adapted from the excellent article "Mastering Ruby blocks in less than 5 minutes" by Cezar Halmagean. (Unfortunately, as of this writing, the article appears to have been deleted.)

For convenience, we'll work in a file called `blocks.rb`:

```
$ touch blocks.rb
```

We'll start by defining a function called `sandwich` that yields a block between two `puts` statements (Listing 5.15).

Listing 5.15: Defining a `sandwich` function. *`blocks.rb`*

Click here to view code image

```
def sandwich
  puts "top bread"
  yield
  puts "bottom bread"
end
```

What does it mean to "yield a block"? The best way to see is to pass `sandwich` a block (Listing 5.16).

Listing 5.16: Passing a block to `sandwich`.*blocks.rb*

Click here to view code image

```
def sandwich
  puts "top bread"
  yield
  puts "bottom bread"
end

sandwich do
   puts "mutton, lettuce, and tomato"
end
```

Running the file then gives us a nice MLT (mutton, lettuce, and tomato sandwich):

Click here to view code image

```
$ ruby blocks.rb
top bread
mutton, lettuce, and tomato
bottom bread
```

One of the advantages of Ruby blocks is that they don't get evaluated until the `yield` keyword appears. This is why in Listing 5.16 we can pass a function that operates using a side effect, which wouldn't work with a normal function parameter. We'll take a closer look at exactly what this means in the exercises (Section 5.4.2), and we'll see how to use it to our advantage when we learn how to write to a file in Chapter 9.

Our final example of blocks involves a `tag` function that wraps some text in a given HTML tag, which gives us a chance to see how to use a block variable. We'll start by using interpolation (Section 2.2) to make HTML code based on the tag name and text, and then yield the result to the block (Listing 5.17).

Listing 5.17: Defining a `tag` function.*blocks.rb*

Click here to view code image

```
.
.
.
def tag(tagname, text)
  html = "<#{tagname}>#{text}</#{tagname}>"
  yield html
end
```

Note that Listing 5.17 shows how to define a function with *multiple* parameters: They're simply separated by commas (and, conventionally, a space).

With the definition in Listing 5.17, we can now pass the `tag` function both its two required parameters (`tagname` and `text`) and a block representing the HTML markup (Listing 5.18).

Listing 5.18: Passing a block (with block variable) to `tag`.*blocks.rb*

Click here to view code image

```
.
.
.
def tag(tagname, text)
  html = "<#{tagname}>#{text}</#{tagname}>"
  yield html
end

# Wrap some text in a paragraph tag.
```

Click here to view code image

```
tag("p", "Lorem ipsum dolor sit amet") do |markup|
  puts markup
end
```

Running the program at the command line gives us the expected result:

Click here to view code image

```
$ ruby blocks.rb
top bread
mutton, lettuce, and tomato
bottom bread
<p>Lorem ipsum dolor sit amet</p>
```

5.4.2 Exercises

1. Use the `downto` method to print out the strings `"99 bottles of beer on the wall"` down to `"1 bottle of beer on the wall"`. (Take care to get the $n = 1$ case right, so that `"bottle"` is singular.)
2. Using the code in Listing 5.19, confirm that a regular function can't replicate the behavior achieved with a block in Listing 5.16.

Listing 5.19: Trying to make a sandwich with a regular function.*blocks.rb*

Click here to view code image

```
.
.
.
def bad_sandwich(contents)
  puts "top bread"
  contents
  puts "bottom bread"
end

bad_sandwich(puts "mutton, lettuce, and tomato")
```

Chapter 6

Functional Programming

Having learned how to define functions and apply them in a couple of different contexts, now we're going to take our programming to the next level by learning the basics of *functional programming*, a style of programming that emphasizes—you guessed it—functions. As we'll see, functional programming also heavily uses blocks, which gives us a good opportunity to reinforce the material from Section 5.4.

This is a challenging chapter, and you may have to get in some reps to fully grok it (Box 6.1), but the rewards are rich indeed.

Box 6.1. Getting in Your Reps

In contexts ranging from martial arts to chess to language learning, practitioners will reach a point where no amount of analysis or reflection will help them improve—they just need to get in some more repetitions, or "reps".

It's amazing how much you can improve by trying something, kinda-sorta (but maybe not quite) getting it, and then just *doing it again*. In the context of a tutorial like this one, sometimes that means rereading a particularly tricky section or chapter. Some people (including yours truly) will even reread an entire book.

One important aspect of getting in your reps is *suspending self-judgment*—allow yourself not to be good right away. (Many people—including, again, yours truly—often require practice to get good at being okay with not being good right away. Meta-reps, as it were.)

Give yourself a break, get in your reps, and watch your technical sophistication grow by the day.

Functional programming de-emphasizes things like mutation and side effects, focusing instead on applying functions to manipulate and transform arguments to functions. This definition is rather abstract, and the subject itself is vast, so we'll make things concrete and manageable by focusing on a classic triumvirate of methods commonly used in functional programming: `map`, `select`, and `reduce` (Figure 6.1).[1]

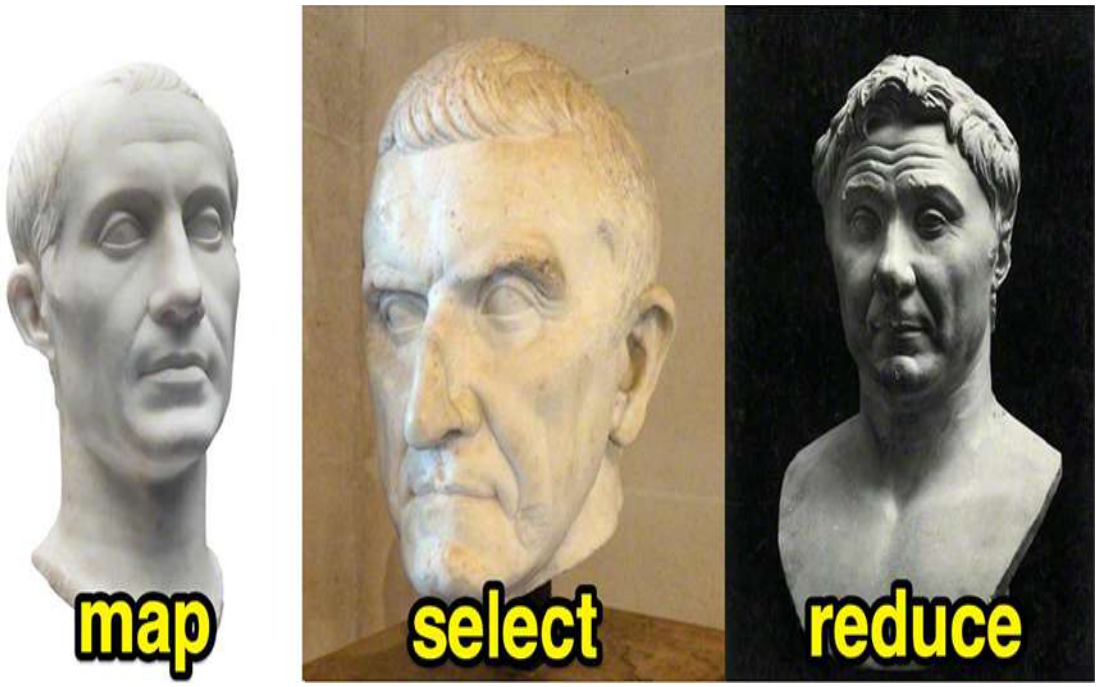


Figure 6.1: A triumvirate of functional methods.

[1]Images courtesy of Kamira/Shutterstock (left), World History Archive/Alamy Stock Photo (center), and colaimages/Alamy Stock Photo (right).

In each case, our technique will be to perform a task involving an `each` method and a sequence of commands (called "imperative programming",[2] which is what we've mostly been doing so far), and then show how to do the same thing using functional programming.

[2]Such programs are written as a series of commands; thus, "imperative," from Latin *imperātīvus*, "proceeding from a command."

For convenience, we'll create a file for our explorations, rather than typing everything at the REPL:

```
$ touch functional.rb
```

6.1 Map

The first of our triumvirate is the `map` function (Figure 6.2),[3] which lets us map a function over an array of elements. It's often a powerful alternative to looping.

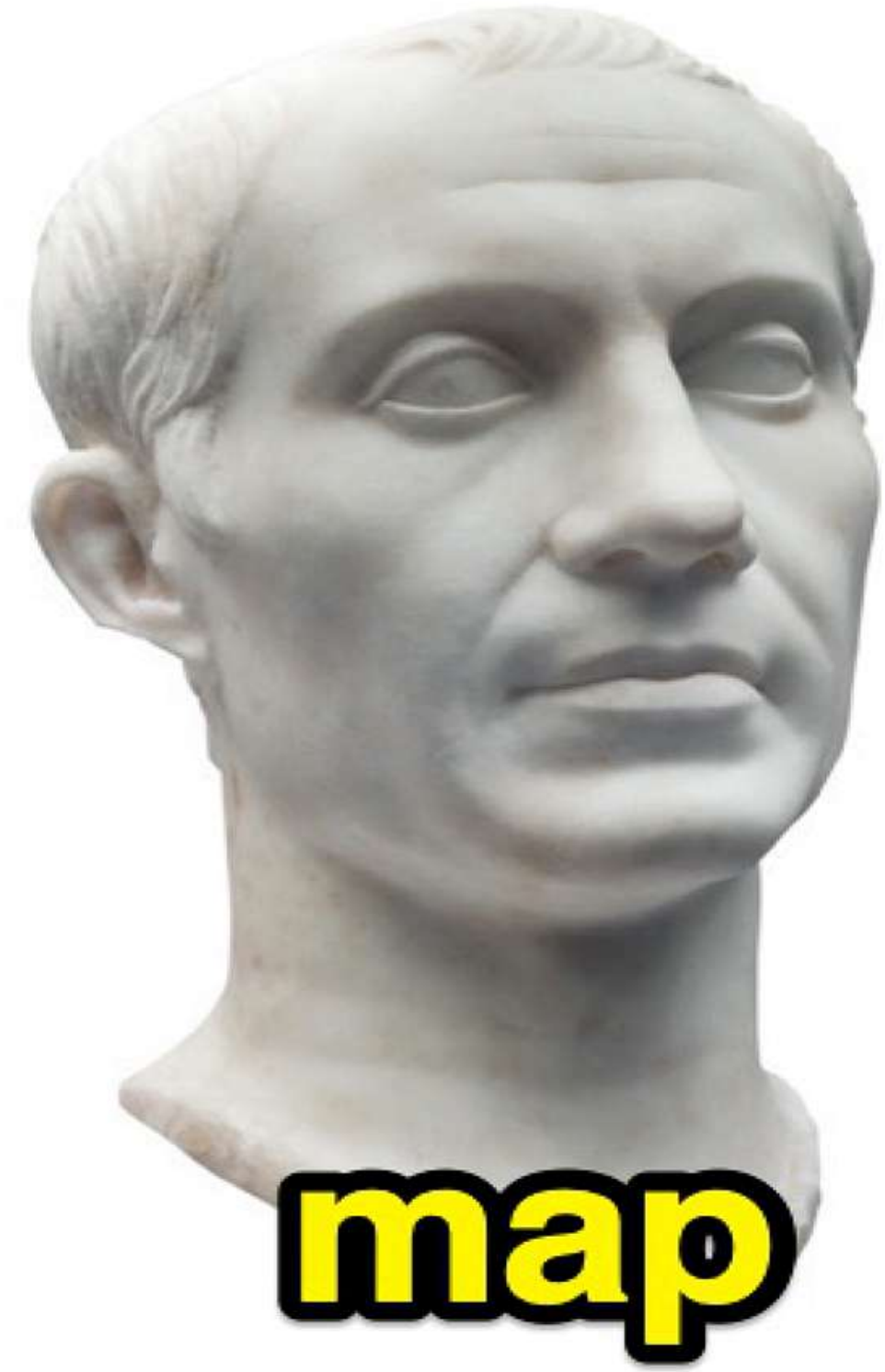


Figure 6.2: The first triumvir, Gāius Iūlius Caesar (Julius Caesar).

[3]Image courtesy of Kamira/Shutterstock. The overbars in *Gāius Iūlius Caesar* and other Latin words are *macrons*, which indicate long vowels.

For example, suppose we had an array of mixed-case strings, and we wanted to create a corresponding array of lowercase strings joined on a hyphen (making the result appropriate for use in URLs), like this:

Click here to view code image

```
"North Dakota" -> "north-dakota"
```

Using previous techniques from this tutorial, we could do this as follows:

1. Define a variable containing an array of strings.
2. Define a second variable (initially empty) for the URL-friendly array of strings.
3. For each item in the first array, `push` (Section 3.4.2) a lowercase version (Section 2.5) that's been split on whitespace (Section 4.3) and then joined (Section 3.4.3) on hyphens. (You could split on a single space `" "` instead, but splitting on whitespace is so much more robust that it's a good practice to use it by default.)

The result appears in Listing 6.1. Note that we've used the shovel operator `<<` in place of an explicit call to the `push` method, although either would work. Also note that we've used `p <obj>` to print a literal representation of the array (where `p` is equivalent to `puts <obj>.inspect`, as discussed in Box 5.1).

Listing 6.1: Making URL-appropriate strings from an array.
functional.rb

Click here to view code image

```
states_map = ["Kansas", "Nebraska", "North Dakota", "South Dakota "]

# urls: Imperative version
def imperative_urls(states)
  urls = []
  states.each do |state|
    urls << state.downcase.split.join("-")
  end
  urls
end
p imperative_urls(states)
```

This is fairly complicated code, so being able to read Listing 6.1 is a good test of your growing technical sophistication. (If it isn't easy to read, firing up irb and running it interactively is a good idea.)

The result of running Listing 6.1 looks like this:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
```

Now let's see how we can do the same thing using `map`, which operates by applying the same function to every element in an array. For example, to square every element in an array of numbers, we can map the function `n*n` over the array, as seen here in the REPL:

Click here to view code image

```
>> [1, 2, 3, 4].map { |n| n*n }
=> [1, 4, 9, 16]
```

Here we've passed a block to `map`, yielding the square of each element.

Similarly, we can downcase every element in an array of strings like this:

Click here to view code image

```
>> ["ALICE", "BOB", "CHARLIE"].map { |name| name.downcase }
=> ["alice", "bob", "charlie"]
```

By the way, the case where a method is called on every element in the sequence is so common that there's a special syntax for it (called, somewhat cryptically, "symbol to proc"):

Click here to view code image

```
>> ["ALICE", "BOB", "CHARLIE"].map(&:downcase)
=> ["alice", "bob", "charlie"]
```

This syntax was actually added dynamically by the Ruby on Rails web framework—a good example of Ruby's unusual flexibility as a language—and people liked it so much it got put into Ruby itself.

Returning to our main example, we can think of the transformation "convert to lowercase then split then join" as a single operation, and use `map` to apply that operation in sequence to each element in the array. The result is so compact that it easily fits in the REPL:

Click here to view code image

```
>> states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
>> states.map { |state| state.downcase.split.join('-') }
=> ["kansas", "nebraska", "north-dakota", "south-dakota"]
```

Pasting into `functional.rb`, we see just how much shorter it is, as shown in Listing 6.2.

Listing 6.2: Adding a functional technique using `map`.
functional.rb

Click here to view code image

```
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]

# urls: Imperative version
def imperative_urls(states)
  urls = []
  states.each do |state|
    urls << state.downcase.split.join("-")
  end

  urls
end
p imperative_urls(states)

# urls: Functional version
def functional_urls(states)
  states.map { |state| state.downcase.split.join('-') }
end
puts functional_urls(states).inspect
```

We can confirm at the command line that the results are the same:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
```

Our functional program has really put the `map` on those states (Figure 6.3).[4]

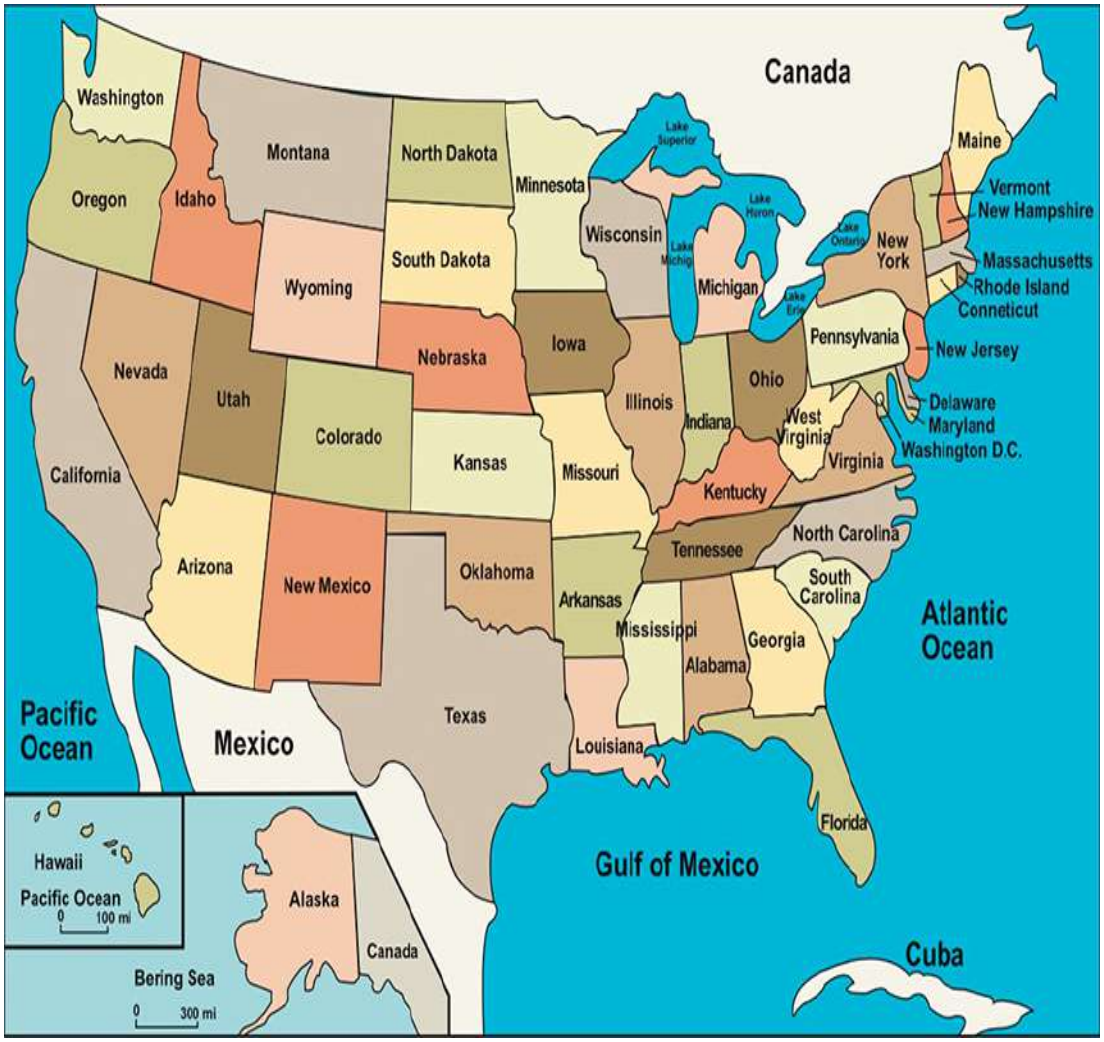


Figure 6.3: Putting some states on the `map`.

[4]Image courtesy of Creative Jen Designs/Shutterstock.

As a final refinement, let's factor the method chain responsible for making the strings URL-compatible into a separate auxiliary function called `urlify`:

Click here to view code image

```
# Returns a URL-friendly version of a string.
#   Example: "North Dakota" -> "north-dakota"
def urlify(string)
  string.downcase.split.join('-'))
end
```

Defining this function in `functional.rb` and using it in the imperative and functional versions gives the code in Listing 6.3.

Listing 6.3: Defining an auxiliary function.
functional.rb

Click here to view code image

```
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]

# Returns a URL-friendly version of a string.
#   Example: "North Dakota" -> "north-dakota"
def urlify(string)
  string.downcase.split.join('-')
end

# urls: Imperative version
def imperative_urls(states)
  urls = []
  states.each do |state|
    urls << urlify(state)
  end
  urls
end
puts imperative_urls(states).inspect

# urls: Functional version
def functional_urls(states)
  states.map { |state| urlify(state) }
end
puts functional_urls(states).inspect
```

As before, the results are the same:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
```

Compared to the imperative version, the functional version is a fifth as many lines (1 instead of 5), doesn't mutate any variables (often an error-prone step in imperative programming), and indeed eliminates the intermediate array (`urls`) entirely. This is the sort of thing that makes Mike Vanier very happy (Figure 6.4).[5]

Figure 6.4: Functional programming makes Mike Vanier happiest of all.

[5]Last I checked, Mike's favorite language was a "purely functional" language called Haskell.

6.1.1 Exercises

1. Using `map`, write a function that takes in the `states` variable and returns an array of URLs of the form `https://example.com/<urlifiedform>`.

6.2 Select

Our second triumvir is `select` (Figure 6.5),[6] which allows us to select elements from our data based on some boolean criterion.

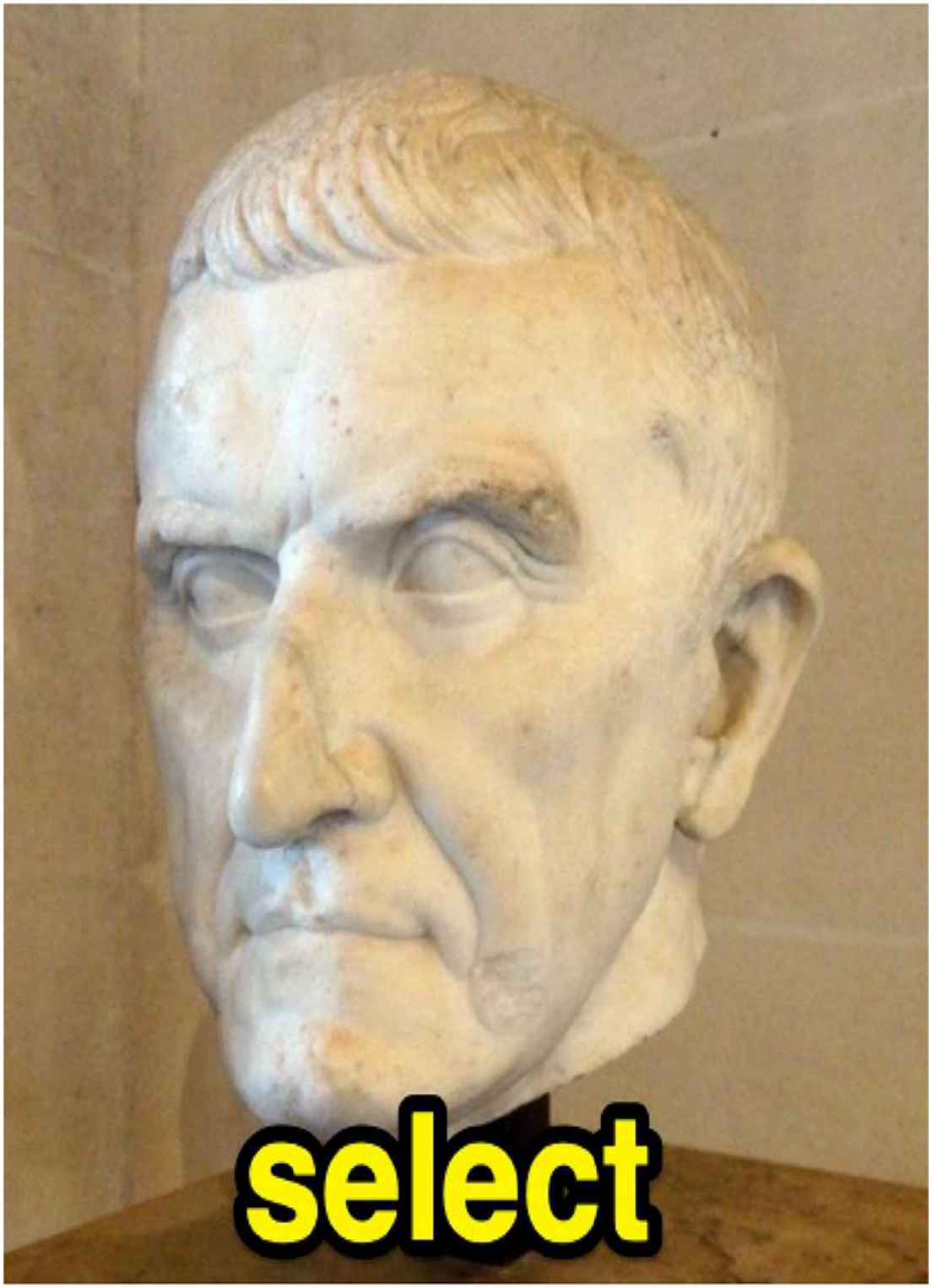


Figure 6.5: The second triumvir, Mārcus Licinius Crassus (at one point the richest man in Rome).

[6]Image courtesy of World History Archive/Alamy Stock Photo.

Suppose, for example, we wanted to select the strings in our `states` array that consist of more than one word, keeping the names that have only one. As in Section 6.1, we'll write an imperative version first:

1. Define an array to store single-word strings.
2. For each element in the list, `push` it to the storage array if splitting it on whitespace yields an array with length 1.

The result looks like Listing 6.4.

Listing 6.4: Solving a filtering problem imperatively.
functional.rb

Click here to view code image

```
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
.
.
.
# singles: Imperative version
def imperative_singles(states)
  singles = []
  states.each do |state|
    if state.split.length == 1
      singles << state
    end
  end
  singles
end
puts imperative_singles(states).inspect
```

Note in Listing 6.4 the familiar pattern from Listing 6.1: We first define an auxiliary variable in order to maintain state (no pun intended); then loop over the original array, mutating the variable as necessary; then return the mutated result. It's not particularly pretty, but it works:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
```

Now let's see how to do the same task using filter. As in Section 6.1, we'll start with a simple numerical example in the REPL.

We'll begin by looking at the *modulo operator* `%`, which returns the remainder after dividing an integer by another integer. In other words, `17 % 5` (read "seventeen mod five") is `2`, because 5 goes into 17 three times (giving 15), with a

remainder of 17 − 15 = 2. In particular, considering integers modulo 2 divides them into two *equivalence classes*: even numbers (remainder 0 (mod 2)) and odd numbers (remainder 1 (mod 2)). In code:

```
>> 16 % 2;  # even
0
>> 17 % 2;  # odd
1
>> 16 % 2 == 0;  # even
=> true
>> 17 % 2 == 0;  # odd
=> false
```

We can combine `%` and `select` to process an array of numbers and select just the even ones:

Click here to view code image

```
>> [1, 2, 3, 4, 5, 6, 7, 8].select { |n| n % 2 == 0}
=> [2, 4, 6, 8]
```

The syntax is almost exactly the same as `map`: we give `select` a variable (`n`) and then perform a *test* that returns `true` or `false`. Incidentally, Ruby integers have an `even?` method that performs the same operation:

Click here to view code image

```
>> [1, 2, 3, 4, 5, 6, 7, 8].select { |n| n.even? }
=> [2, 4, 6, 8]
```

We can do even better by combining a range and the “symbol-to-proc” notation:

```
>> (1..8).select(&:even?)
=> [2, 4, 6, 8]
```

This sort of compactness is characteristic of idiomatically correct (and functional) Ruby.

Using this idea, we see that the functional version is much cleaner—indeed, as with `map`, the `select` version is a single line (a common occurrence in functional programming), as we can see in the REPL:

Click here to view code image

```
>> states.select { |state| state.split.length == 1 }
```

Placing the result in our example file again underscores how much more compact the functional version is (Listing 6.5).

Listing 6.5: Solving a selection problem functionally.
functional.rb

Click here to view code image

```
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
.
.
.
# singles: Imperative version
def imperative_singles(states)
  singles = []
  states.each do |state|
    if (state.split.length == 1)
       singles << state
    end
  end
  singles
end
puts imperative_singles(states).inspect

# singles: Functional version
def functional_singles(states)
  states.select { |state| state.split.length == 1 }
end
puts functional_singles(states).inspect
```

As required, the result is the same:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
["Kansas", "Nebraska"]
```

6.2.1 Exercises

1. Write two `select` functions that return the Dakotas: one using `String#include?` (Section 2.5) to test for the presence of the string “Dakota” and one that tests for the length of the split array being `2`.

6.3 Reduce

We reach finally the third member of our triumvirate, the mighty `reduce` (Figure 6.6)[7]—by far the most complicated of the three.

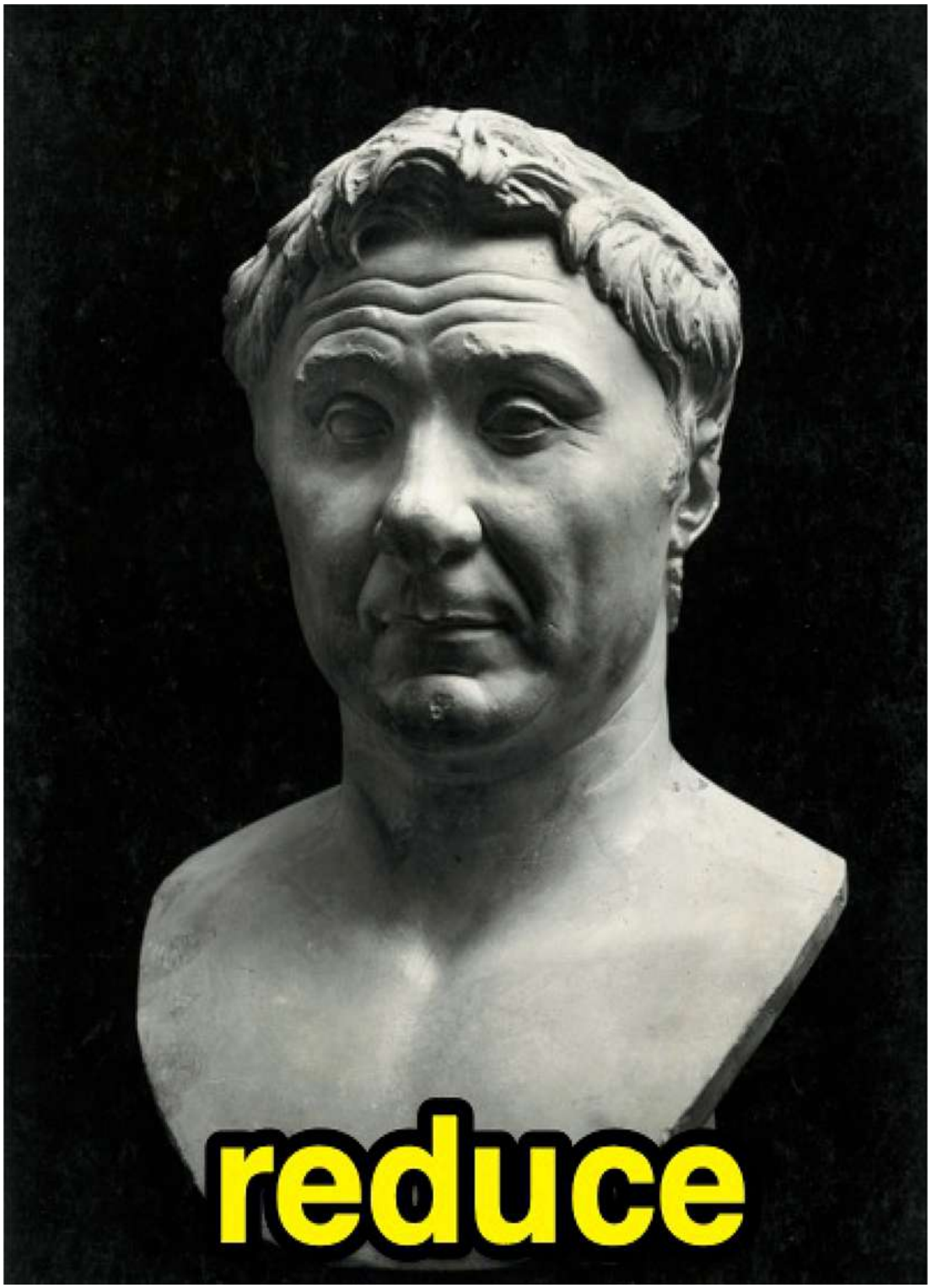


Figure 6.6: The third triumvir, Gnaeus Pompēius Magnus (Pompey the Great).

[7] Image courtesy of colaimages/Alamy Stock Photo.

Because `reduce` is particularly challenging, we'll cover two examples. First, we'll make imperative and functional versions of a `sum` operation on arrays of integers. Second, we'll make a hash (Section 4.4) that maps state names to the length of each name, with a result that will look like this:

```
{ "Kansas" => 6,
  "Nebraska" => 8,
  .
  .
  .
}
```

6.3.1 Reduce, Example 1

We'll begin with an imperative solution for the `sum` function, which involves (as usual) an `each` loop and an auxiliary variable (`total`), which we use to accumulate the result. The result appears in Listing 6.6.

Listing 6.6: An imperative solution for summing integers.
functional.rb

```
.
.
.
numbers = 1..10

# sum: Imperative solution
def imperative_sum(numbers)
  total = 0
  numbers.each do |n|
    total += n
  end
  total
end
puts imperative_sum(numbers)
```

Again we see the familiar pattern: Initialize an auxiliary variable (`total`) and then loop through the collection, accumulating the result by adding each number to the total.

The result is 55 as required:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
["Kansas", "Nebraska"]
55
```

Now for the `reduce` solution. It's a bit tricky, so let's work in the REPL:

Click here to view code image

```
>> numbers = 1..10
>> numbers.reduce(0) do |total, n|
?> total += n
?> total
?> end
55
```

You can see what I meant by "tricky". The `reduce` method takes a function of *two* arguments, the first of which is an *accumulator* for the result, and the second of which is the array element itself. The return value of the (anonymous) function gets passed back to `reduce` as the starting value for the next element in the array. The argument to `reduce` is the initial value of the accumulator (in this case, `0`).

There are three refinements we can make. First, the `+=` operator returns its value, so we can actually increment a value while simultaneously assigning it (or returning it):

```
>> i = 0
>> j = i += 1
>> i
1
>> j
1
```

This means we can return `total += n` directly:

Click here to view code image

```
>> numbers.reduce(0) { |total, n| total += n }
55
```

Second, the initial value is `0` by default, so in this case it can be left off:

Click here to view code image

```
>> numbers.reduce { |total, n| total += n }
55
```

Finally, the last value in the block (which in this case is the only value in the block) is automatically put into the variable being used to accumulate the result, so instead of modifying `total` using `+=` we can just add `n` to it:

Click here to view code image

```
>> numbers.reduce { |total, n| total + n }
55
```

Putting the result into our example file shows, as usual, a marked improvement over the imperative version (Listing 6.7).

Listing 6.7: A functional solution for summing integers.
functional.rb

Click here to view code image

```
.
.
.
numbers = 1..10

# sum: Imperative solution
def imperative_sum(numbers)
  total = 0
  numbers.each do |n|
    total += n
  end
  total
end
puts imperative_sum(numbers)

# sum: Functional solution
def functional_sum(numbers)
  numbers.reduce { |total, n| total + n }
end
puts functional_sum(numbers)
```

The result of the functional sum should be the same as for the imperative version:

Click here to view code image

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
["Kansas", "Nebraska"]
55
55
```

Listing 6.7 gives us a hint about the meaning of `reduce`: It's a function that takes the elements of an array and processes (*reduces*) them based on some operation (in this case, addition). This is not always the case, though, and as we'll see in a moment it's often more helpful to think of `reduce` as *accumulating* results and storing them in its first argument (`total` in Listing 6.7).[8]

[8]For this reason, `reduce` is sometimes called `accumulate` in other languages. See, e.g., "Sequence Operations" in Chapter 2 of *Structure and Interpretation of Computer Programs*.

6.3.2 Reduce, Example 2

To help reinforce `reduce`, let's take a look at a second example. As mentioned above, our task is to make a plain object (associative array) with keys equal to the state names and values equal to their lengths. We can solve this imperatively by initializing a `lengths` object and then iterating through the states, setting `lengths[state]` equal to the corresponding length:

```
lengths[state] = state.length
```

The full example appears in Listing 6.8.

Listing 6.8: An imperative solution for state/length correspondence.
functional.rb

Click here to view code image

```
.
.
.
# lengths: Imperative version
def imperative_lengths(states)
  lengths = {}
  states.each do |state|
    lengths[state] = state.length
  end
  lengths
end
puts imperative_lengths(states)
```

If we run the program at the command line, the desired associative array appears as the final part of the output:

Click here to view code image

```
$ ruby functional.rb
.
.
.
{"Kansas"=>6, "Nebraska"=>8, "North Dakota"=>12, "South Dakota"=>12}
```

The functional solution using `reduce` is trickier. As with the imperative solution, we have a plain `lengths` object, but instead of being an auxiliary variable, it's a

block parameter:

```
do |lengths, state|
  lengths[state] = state.length
  lengths
end
```

Meanwhile, the initial value of the `reduce` method, instead of being the default `0`, is the empty hash `{}`:

Click here to view code image

```
reduce({}) do |lengths, state|
  lengths[state] = state.length
  lengths
end
```

Note that these are code snippets, not REPL sessions; one disadvantage of `reduce` (and functional solutions generally) is that they are harder to build up incrementally. More on this in a moment.

Taking the above ideas together, we can use `reduce` to march through the `states` array, accumulating the desired associative array in the `lengths` parameter and then returning it, as shown in Listing 6.9.

Listing 6.9: A functional solution for state/length correspondence.
functional.rb

Click here to view code image

```
.
.
.
# lengths: Imperative version
def imperative_lengths(states)
  lengths = {}
  states.each do |state|
    lengths[state] = state.length
  end
  lengths
end
puts imperative_lengths(states)

# lengths: Functional version
def functional_lengths(states)
  states.reduce({}) do |lengths, state|
    lengths[state] = state.length
    lengths
  end
end
puts functional_lengths(states)
```

Although it is broken across multiple lines in the text editor, the functional solution in Listing 6.9 `return`s the result of a single `reduce`, in close analogy with the functional solutions for `map` (Listing 6.2) and `select` (Listing 6.5). (In fact, using the more advanced `merge` method, the `reduce` in Listing 6.9 can actually be compressed to a single line:

Click here to view code image

```
states.reduce({}) { |lengths, state| lengths.merge({state => state.length}) }
```

See the documentation on `merge` for more information.)

As required, the result is the same as the imperative solution:

Click here to view code image

```
$ ruby functional.rb
.
.
.
{"Kansas"=>6, "Nebraska"=>8, "North Dakota"=>12, "South Dakota"=>12}
{"Kansas"=>6, "Nebraska"=>8, "North Dakota"=>12, "South Dakota"=>12}
```

Comparing the imperative and functional solutions in Listing 6.9, the advantages of `reduce` are not as clear as they were in the case of `map` and `select`. Indeed, a good argument can be made that the imperative solution is clearer.

Which method to use is a matter of taste. I've found that the more you program functionally, the more you want to do it, and there's a strange sort of pleasure in using `reduce` to solve a problem in a single (logical) line. It's also worth noting that `reduce` is a common technique among more advanced programmers, and among other things plays a key role in an important technique (called MapReduce) for dealing efficiently with large datasets.

6.3.3 Functional Programming and TDD

One of the things you may have noticed when building up Listing 6.9 is that the functional solution is harder to break down into steps. The advantage is that we can often condense a functional solution into a single line, but the cost is that it can be harder to understand incrementally. Indeed, I find this to be a consistent pattern across all three functions in our triumvirate; the final destination is often beautifully succinct, but getting there can be a challenge.

My favorite technique for managing this challenge is *test-driven development* (TDD), which involves writing an *automated test* that captures the desired behavior in code. We can then get the test to pass using any method we want,

including an ugly but easy-to-understand imperative solution. At that point, we can *refactor* the code—changing its form but not its function—to use a more concise functional solution. As long as the test still passes, we can be confident that the code still works.

In Chapter 8, we'll apply this exact technique to the principal object developed in Chapter 7. In particular, we'll use TDD to implement a fancy extension to the `palindrome?` function first seen in Section 5.3, one that detects such complicated palindromes as "A man, a plan, a canal—Panama!" (Figure 6.7).[9]

Figure 6.7: Teddy Roosevelt was a man with a plan.

[9] Image courtesy of Everett Collection Historical/Alamy Stock Photo.

6.3.4 Terminology Review

Unlike most other programming languages, Ruby supports a large number of *synonyms*, which are simply different names for methods that do the same thing. For example, `Array#size` is a synonym for the `Array#length` method we met in Section 3.2:

```
>> a = [42, 8, 17, 99]
=> [42, 8, 17, 99]
>> a.length
=> 4
>> a.size
=> 4
```

In the case of functional programming, Ruby has a group of methods with similar-sounding names, several of which have synonyms:

- `collect`: Synonym for `map`
- `select`: Also called `find_all`

- `inject`: Synonym for `reduce`
- `detect`: Also called `find` (not covered in this chapter)
- `reject`: Like `select`, but with the boolean condition reversed (covered in an exercise (Section 6.3.5))

Thus, our original triumvirate of `map`, `select`, and `reduce` can also be written as `collect`, `select`, and `inject` (Figure 6.8).[10] All of these methods are part of an important Ruby *module* (Section 7.5) called `Enumerable`. For more on this subject, take a look at "Enumerable for Fun and Profit", a talk I gave at RubyConf 2014.

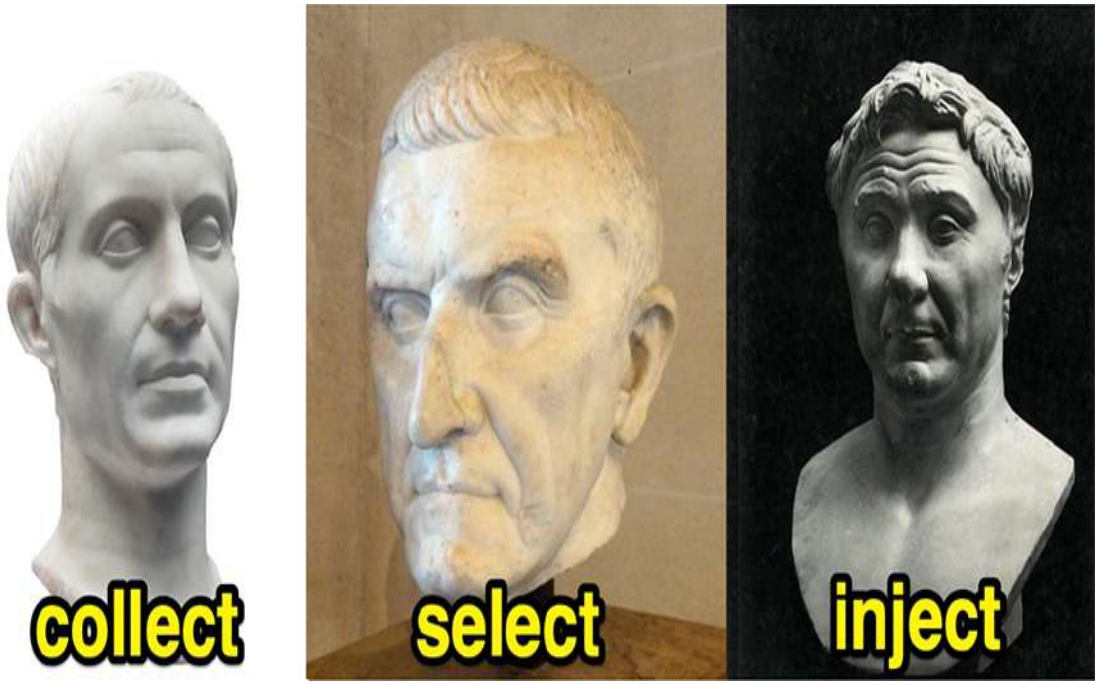


Figure 6.8: A triumvirate by any other name…

[10]Images courtesy of Kamira/Shutterstock (left), World History Archive/Alamy Stock Photo (center), and colaimages/Alamy Stock Photo (right).

Although the main names used in this chapter are generally the more common ones, context does matter. In particular, when used to accumulate an object (as in this section) rather than reducing down to a single value, `inject` is probably more common than `reduce`, so Listing 6.9 would likely be written as in Listing 6.10 instead.

Listing 6.10: Using `inject` in place of `reduce`.
functional.rb

Click here to view code image

```
.
.
.
# lengths: Functional version using `inject`
def functional_lengths(states)
  states.inject({}) do |lengths, state|
    lengths[state] = state.length
    lengths
  end
end
puts functional_lengths(states)
```

In any case, because they truly are synonyms, it's important to be able to *read* code that includes any of the versions above.

6.3.5 Exercises

1. Using `reduce`, write a function that returns the product of all the elements in an array. Start with a version analogous to using `total += n`. (*Hint*: Where `+=` adds, `*=` multiplies.) Then eliminate the assignment and return the product directly (just as the assignment was eliminated in Listing 6.7 and replaced with `total + n`).
2. Remove the newlines in the `reduce` solution from Listing 6.9 to turn it into a single long line. Does it still give the right answer? How long is the resulting line of code?
3. Rewrite all the examples from this chapter using `collect` and `inject` instead of `map` and `reduce`. Rewrite the `select` example using `reject`.

Chapter 7

Objects and Classes

So far in this tutorial we've seen many examples of Ruby objects. In this chapter, we'll learn how to use Ruby *classes* to make objects of our own, which have both attributes (data) and methods (functions) attached to them.

7.1 Defining Classes

As we saw in Section 4.2, Ruby objects can be created (or *instantiated*) using the class name and the `new` constructor method:

Click here to view code image

```
>> String.new("Madam, I'm Adam.")
=> "Madam, I'm Adam."
>> Time.new(1969, 7, 20, 20, 17, 40)
=> 1969-07-20 20:00:00 -0700
```

We can create a class of our own using three basic elements:

1. Use the `class` keyword to define the class.
2. Use the special `initialize` method to specify the behavior of `new`.
3. Use an *attribute accessor* (`attr_accessor`) to allow the getting and setting of attributes.

Our concrete example will be a `Phrase` class with a `content` attribute, which we'll put in `palindrome.rb`. Let's build it up piece by piece (for simplicity we'll omit the `palindrome?` function for the moment). The first element is the `class` itself (Listing 7.1).

Listing 7.1: Defining a `Phrase` class.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class.
class Phrase
```

```
end

phrase = Phrase.new
puts phrase
```

Here `Phrase` automatically has a `new` method because of *inheritance*, which we discuss in detail in Section 7.2.[1] Meanwhile, the final `puts` in Listing 7.1 lets us see some concrete (if not especially instructive) results at the command line:

[1] Specifically, `Phrase` automatically inherits from `Object`.

Click here to view code image

```
$ ruby palindrome.rb
#<Phrase:0x00007fa3d30a3e98>
```

This shows Ruby's abstract internal representation of a bare `Phrase` class. (Should your results match exactly?)

We'll start filling in Listing 7.1 in a moment, but before moving on we should note that, unlike variables and methods, Ruby classes use *CamelCase* (with a leading capital) instead of snake_case.

CamelCase, which is named for the resemblance of the capital letters to humps of a camel (Figure 7.1)),[2] involves separating words using capitalization rather than with underscores. It's hard to tell with `Phrase`, since it's only a single word, but we'll see the principle more clearly illustrated in Section 7.2, which defines a class called `TranslatedPhrase`.

[2] Image courtesy of Utsav Academy and Art Studio. Pearson India Education Services Pvt. Ltd.

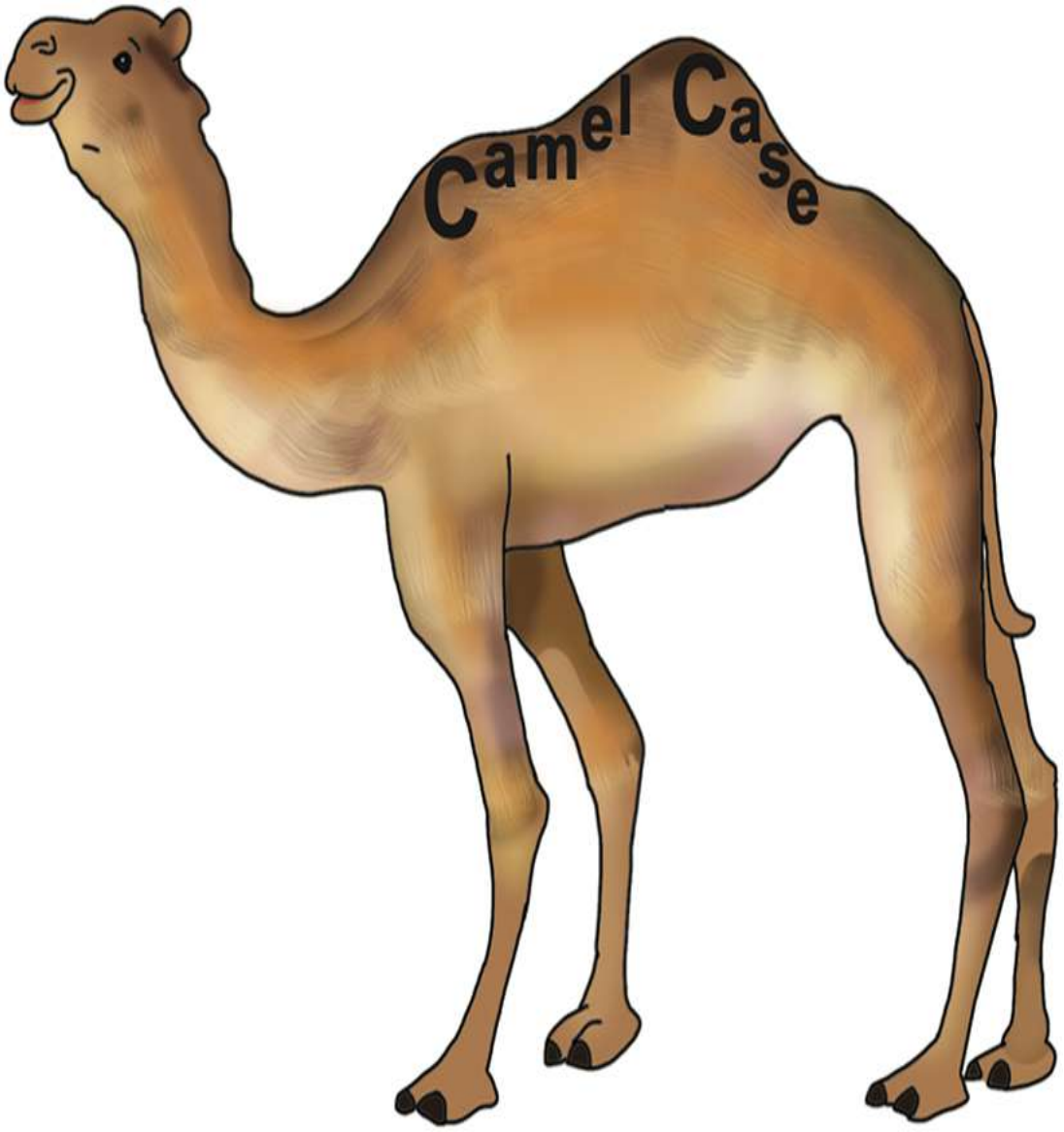


Figure 7.1: The origin of CamelCase.

Eventually, we'll use `Phrase` to represent a phrase like "Madam, I'm Adam." that can qualify as a palindrome even if it's not literally the same forward and backward. At first, though, all we'll do is define a `Phrase` constructor function that takes in an argument (the `content`) and sets the property `content`. As with the `length` property of `String` objects (Section 2.4), we can access a phrase's content using the familiar dot notation.

In order to get this to work, we first need to define the `initialize` method that gets called when we initialize an object using `Phrase.new` (Listing 7.2).[3]

[3] Why `initialize` isn't just called `new` I have no idea.

Listing 7.2: Defining `initialize`.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class.
class Phrase

  def initialize(content)
    @content = content
  end
end

phrase = Phrase.new("Madam, I'm Adam.")
puts phrase.content
```

Listing 7.2 initializes the special *instance variable* `@content`, which is distinguished by a leading `@` symbol and is pronounced "at content". As we'll see in a moment, this automatically allows us to associate the variable with an object attribute. But first, let's see the effect at the command line:

Click here to view code image

```
$ ruby palindrome.rb
Traceback (most recent call last):
palindrome.rb:15:in `<main>': undefined method `content' for
#<Phrase:0x00007fe8c70a3db0 @content="Madam, I'm Adam."> (NoMethodError)
```

Because we have only an `@content` variable, Ruby doesn't know what to make of `phrase.content`, so it raises a `NoMethodError` exception (which we saw before in Section 5.2).

The way to fix this issue is by adding special *accessor* methods (also known as getter/setter methods) that allow us to access ("get") and assign ("set") the given attribute. Ruby has a special syntax for this called an `attr_accessor` ("attribute accessor"), as seen in Listing 7.3.

Listing 7.3: Adding an attribute accessor.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class.
class Phrase
  attr_accessor :content
```

```
  def initialize(content)
   @content = content
  end
end

phrase = Phrase.new("Madam, I'm Adam.")
puts phrase.content
```

Note that the argument to `attr_accessor` is a *symbol* (Section 4.4.1), which Ruby automatically associates with the instance variable of the same name. In the case of Listing 7.3, this means `phrase.content` uses the value of the `@content` instance variable.

It's important to understand which uses of `content` in Listing 7.3 need to use that name, and which don't. For the `content` attribute to work correctly, all three of `:content`, `@content`, and `phrase.content` have to use the same word (in this case, "content"), but the argument to `initialize` can be whatever name we want. (We just used `content` for convenience.) These relationships are illustrated in Figure 7.2.

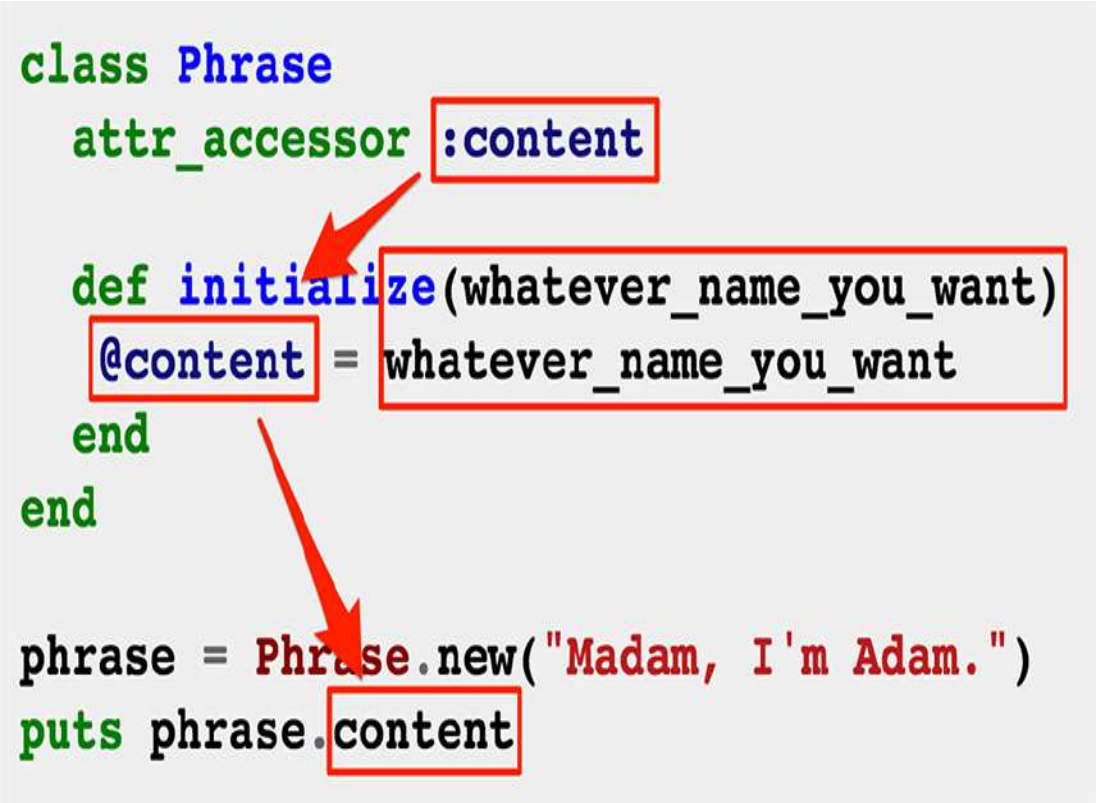


Figure 7.2: The word relationships for creating an object attribute.

With the definition in Listing 7.3, we now have a working example:

```
$ ruby palindrome.rb
Madam, I'm Adam.
```

We can also now assign directly to `content` using the dot notation, as seen in Listing 7.4.

Listing 7.4: Assigning to an object attribute.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end
end

phrase = Phrase.new("Madam, I'm Adam.")
puts phrase.content

phrase.content = "Able was I, ere I saw Elba."
puts phrase.content
```

The result is as you probably can guess:

Click here to view code image

```
$ ruby palindrome.rb
Madam, I'm Adam.
Able was I, ere I saw Elba.
```

At this point, we're ready to restore the `palindrome?` method in our initial definition of `Phrase`, as shown in Listing 7.5 (which also deletes the `puts` and related lines).

Listing 7.5: Our initial `Phrase` class definition.
palindrome.rb

Click here to view code image

```
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  processed_content = string.downcase
  processed_content == processed_content.reverse
end
```

```
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end
end
```

Just as a reality check, it's a good idea to run it in irb to catch any syntax errors, etc.:

Click here to view code image

```
>> load "./palindrome.rb"
>> phrase = Phrase.new("Racecar")
>> phrase .content
=> "Racecar"
>> palindrome?(phrase.content)
=> true
```

As a next step, we'll move the `palindrome?` function into the `Phrase` object itself, adding it as a method. The only things we need to do are (1) change the method to take zero arguments and (2) use the `Phrase` content instead of the variable `string`. Exactly how to do this second step is shown in Listing 7.6.

Listing 7.6: Moving `palindrome?` into the `Phrase` class.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.content.downcase
    processed_content == processed_content.reverse
  end
end
```

We see in Listing 7.6 that inside the `palindrome?` method we can access the value of `content` using the special `self` object, which inside the class represents the instance itself.

In other words, when we write

Click here to view code image

```
phrase = Phrase.new("Racecar")
```

we can access the attributes of `phrase` inside the `Phrase` class using `self`. (Compare with `this` in JavaScript.)

The result of Listing 7.6 is that we can now call `palindrome?` directly on a phrase, as follows:

Click here to view code image

```
>> load "./palindrome.rb"
>> phrase = Phrase.new("Racecar")
>> phrase.palindrome?
=> true
```

It worked! A `phrase` instance initialized with the string “Racecar” knows that it’s a palindrome (Figure 7.3).[4]

Figure 7.3: A Formula One palindrome.

[4]Image courtesy of msyaraafiq/Shutterstock.

The palindrome detector in Listing 7.6 is fairly rudimentary, but we now have a good foundation for building (and testing) a more sophisticated palindrome detector in Chapter 8.

7.1.1 Exercises

1. By filling in the code in Listing 7.7, add a `louder` method to the `Phrase` object that returns a LOUDER (all-caps) version of the content. Confirm in the REPL that the result appears as in Listing 7.8.

Listing 7.7: Making the content LOUDER.
palindrome.rb

Click here to view code image

```ruby
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.content.downcase
    processed_content == processed_content.reverse
  end

  # Makes the phrase LOUDER.
  def louder
    # FILL IN
  end
end
```

Listing 7.8: Using `louder` in the REPL.

Click here to view code image

```
>> load "./palindrome.rb"
>> p = Phrase.new("yo adrian!")
>> p.louder
=> "YO ADRIAN!"
```

7.2 Inheritance

When learning about Ruby classes, it's useful to investigate the *class hierarchy* using the `class` and `superclass` methods. Let's look at an example of what this means in the case of a familiar class, `String`:

Click here to view code image

```
>> s = String.new("foobar")
=> "foobar"
>> s.class                            # Find the class of s.
=> String
>> s.class.superclass                 # Find the superclass of String.
=> Object
>> s.class.superclass.superclass      # Ruby has a BasicObject base class as of 1.9
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

A diagram of this inheritance hierarchy appears in Figure 7.4. We see here that the superclass of `String` is `Object` and the superclass of `Object` is `Basic-Object`, but `BasicObject` has no superclass. This pattern is true of every Ruby object: Trace back the class hierarchy far enough and every class in Ruby ultimately inherits from `BasicObject`, which has no superclass itself. This is the technical meaning of "everything in Ruby is an object".

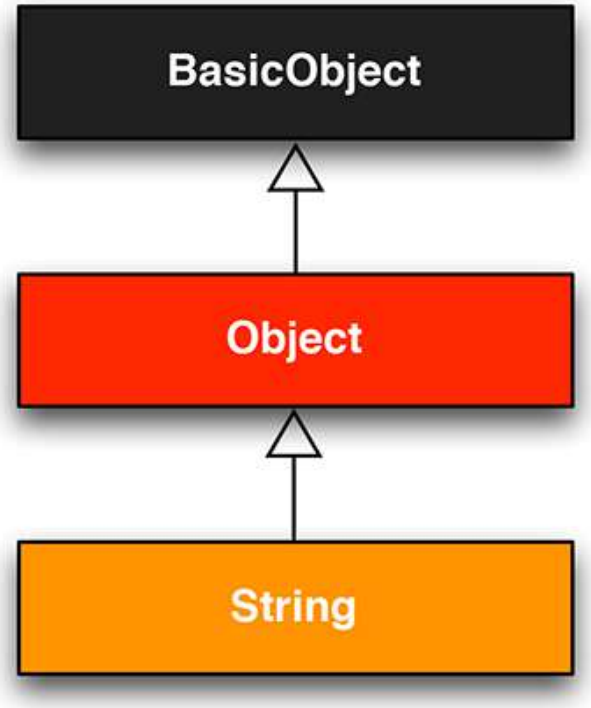


Figure 7.4: The inheritance hierarchy for the `String` class.

The way Ruby's class hierarchy works is that each class *inherits* the attributes and methods of classes further up in the hierarchy. For example, we've just seen how to find the class of a `String` object:

Click here to view code image

```
>> "honey badger".class
=> String
```

But where does the `class` method itself come from? The answer is that `String` inherits `class` from `Object`, as seen in the documentation on `Object` (Figure 7.5). Because `String` inherits from `Object`, it automatically has a `class` method (as well as all the other `Object` methods). *Note*: In older versions of Ruby, `Object` was the base of all classes, but `Object` has a lot of methods, and experience showed that using a super-stripped-down `BasicObject` class was convenient for some applications.

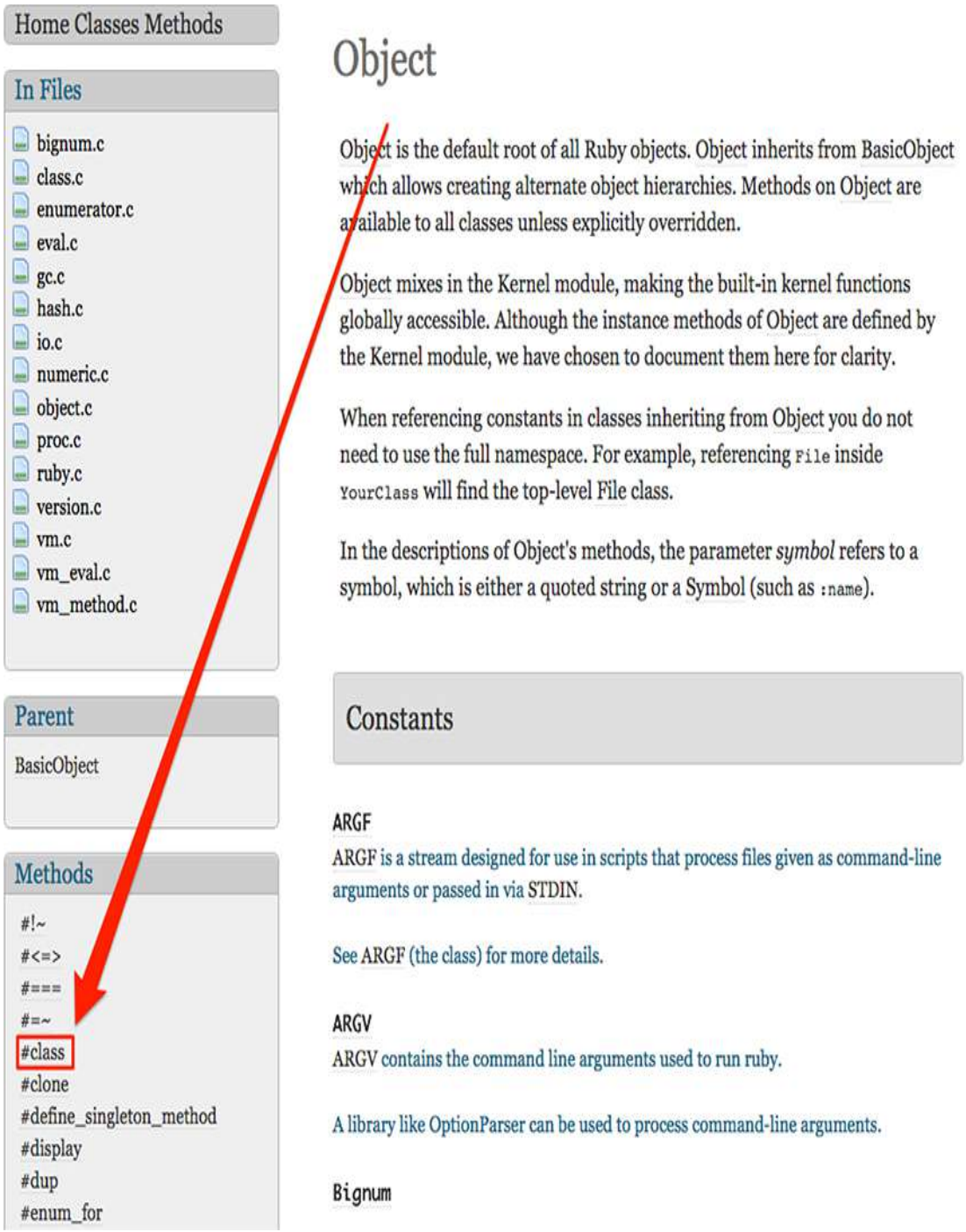


Figure 7.5: The `class` method in the `Object` documentation.

Let's return now to the `Phrase` class we defined in Section 7.1. Right now, `Phrase` has a `content` attribute, but a phrase really is a *string*, which suggests inheriting from the `String` class itself. (In the terminology of object-oriented programming, this is known as an *is-a relationship*, because "a phrase is a string.") The way to inherit in Ruby is to use the left angle bracket `<`, as shown in Listing 7.9. Note that we've replaced `self.content.downcase` from Listing 7.6 with `self.downcase`—because a

phrase *is a* string, `self` is a string, which means we can call the `downcase` method directly.

Listing 7.9: Inheriting from `String`.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class.
class Phrase < String
  attr_accessor :content

  def initialize(content)
    @content = content
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.downcase
    processed_content == processed_content.reverse
  end
end
```

Observant readers might note at this point that we're no longer using `content` at all, which means we should eliminate the attribute accessor and `initialize` method entirely. The result is the much more compact `Phrase` class definition shown in Listing 7.10.

Listing 7.10: Eliminating `initialize` and `content`.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.downcase
    processed_content == processed_content.reverse
  end
end
```

We can confirm using irb that this worked (you may have to exit and restart irb first):

Click here to view code image

```
>> load "./palindrome.rb"
>> phrase = Phrase.new("Racecar")
>> phrase.palindrome?
=> true
```

Here the `new` method is the one on `String`—we have eliminated our custom `initialize` method entirely. Because a `Phrase` is a `String`, we can also call any of the string methods on it:

```
>> phrase.empty?
=> false
>> phrase.length
=> 7
>> phrase.scan(/c\w/)
=> ["ce", "ca"]
```

The relationship between `Phrase` and the other classes in the hierarchy can be seen as follows:

Click here to view code image

```
>> phrase.class
=> Phrase
>> phrase.class.superclass
=> String
>> phrase.class.superclass.superclass
=> Object
>> phrase.class.superclass.superclass.superclass
=> BasicObject
```

A visual representation appears in Figure 7.6.

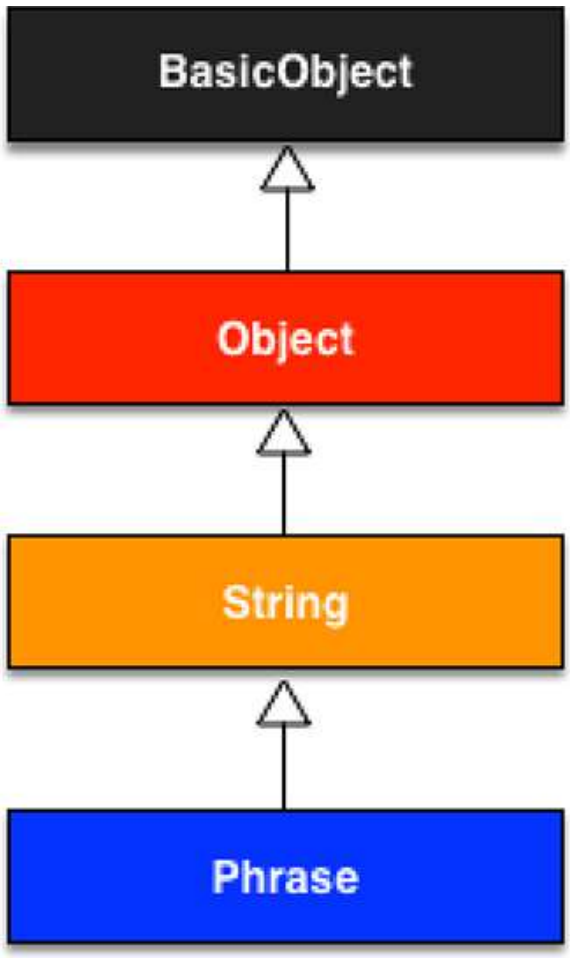


Figure 7.6: The inheritance hierarchy for the `Phrase` class.

7.2.1 Exercises

1. Inside a Ruby class, the use of `self.` is necessary when making attribute assignments, but otherwise you can leave it off. In particular, confirm that we can write `downcase` in place of `self.downcase`, as shown in Listing 7.11. This may look a little odd at first, but the practice of leaving off `self.` when possible is common in idiomatic Ruby.

Listing 7.11: Eliminating `self.` outside an assignment.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class (inheriting from String).
class Phrase < String
```

```
  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = downcase
    processed_content == processed_content.reverse
  end
end
```

7.3 Derived Classes

Let's build on the techniques in Section 7.2 to make a class that inherits from `Phrase`, which we'll call `TranslatedPhrase`. The purpose of this so-called *derived class* is to re-use as much of `Phrase` as possible while giving us the flexibility to, say, test if a *translation* is a palindrome.

We'll start by factoring `processed_content` into a separate method, as shown in Listing 7.12. We'll see in a moment why this is useful in the current context, though it's a nice refinement in any case.

Listing 7.12: Factoring `processed_content` into a method.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end
```

Now we're ready to inherit from `Phrase`. We'll start by using the inheritance operator `<` as described above:

Click here to view code image

```
# Defines a translated Phrase.
class TranslatedPhrase < Phrase

end
```

Our plan is to use `TranslatedPhrase` like this:

Click here to view code image

```
TranslatedPhrase.new("recognize", "reconocer")
```

where the first argument is the `Phrase` content and the second argument is the translation. As a result, a `TranslatedPhrase` instance needs a `translation` attribute, which we'll create using `initialize` and `attr_accessor` as with `content` in Listing 7.3:

Click here to view code image

```
# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    @translation = translation
  end
end
```

Note that `initialize` takes *two* arguments, `content` and `translation`. We've handled `translation` like a normal attribute, but what to do about content? In Listing 7.12, we could leave it off and simply delegate to the constructor for the `String` class, but how do we do this inside `TranslatedPhrase`? The answer is a special Ruby method called `super`:

Click here to view code image

```
# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end
end
```

This calls the `initialize` method for the superclass—in this case, Ruby looks for one in `Phrase`, but `Phrase` has no `initialize`, so Ruby keeps going up the class hierarchy (Figure 7.6) until it finds one in the `String` class. At that point, Ruby initializes `self` to have the value given by the `content` parameter.

Putting everything together gives the `TranslatedPhrase` class shown in Listing 7.13.

Listing 7.13: Defining `TranslatedPhrase`.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class (inheriting from String).
class Phrase < String
  .
  .
  .
end

# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end
end
```

Because `TranslatedPhrase` inherits from the `Phrase` object, an instance of `TranslatedPhrase` automatically has all the methods of a `Phrase` instance, including `palindrome?`. Let's create a variable called `frase` (pronounced "FRAH-seh", Spanish for "phrase") to see how it works (Listing 7.14).

Listing 7.14: Defining a `TranslatedPhrase`.

Click here to view code image

```
>> load "./palindrome.rb"
>> frase = TranslatedPhrase.new("recognize", "reconocer")
>> frase.palindrome?
=> false
```

We see that `frase` has a `palindrome?` method as claimed, and that it returns `false` because "recognize" isn't a palindrome.

But what if we wanted to use the *translation* instead of the content for determining whether the translated phrase is a palindrome or not? Because we factored `processed_content` into a separate method (Listing 7.12), we can do this by *overriding* the `processed_content` method in `TranslatedPhrase`, as seen in Listing 7.15.

Listing 7.15: Overriding a method.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end

# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end

  # Processes the translation for palindrome testing.
  def processed_content
    self.translation.downcase
  end
end
```

The key point in Listing 7.15 is that we're using `self.translation` in the `TranslatedPhrase` version of `processed_content`, so Ruby knows to use that one instead of the one in `Phrase`. Because the translation "reconocer" *is* a palindrome, we get a different result from the one we got in Listing 7.14, as shown in Listing 7.16.

Listing 7.16: Calling `palindrome?` after overriding `processed_content`.

Click here to view code image

```
>> load "./palindrome.rb"
>> frase = TranslatedPhrase.new("recognize", "reconocer")
>> frase.palindrome?
=> true
```

The resulting inheritance hierarchy appears as in Figure 7.7.

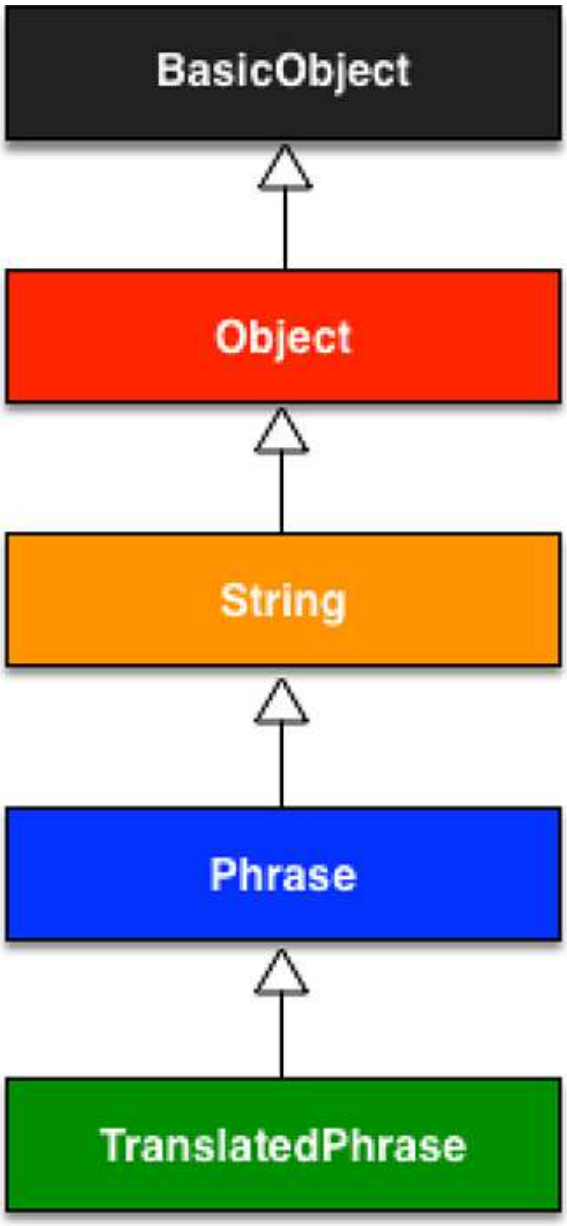


Figure 7.7: The inheritance hierarchy for the `TranslatedPhrase` class.

This practice of overriding gives us great flexibility. We can trace the execution of `frase.palindrome?` for the two different cases:

Case 1: Listing 7.13 and Listing 7.14

1. `frase.palindrome?` calls `palindrome?` on the `frase` instance, which is a `TranslatedPhrase`. Since there is no `palindrome?` method in the `TranslatedPhrase` object, Ruby uses the one from `Phrase`.

2. The `palindrome?` method in `Phrase` calls the `processed_content` method. Since there is no `processed_content` method in the `TranslatedPhrase` object, Ruby uses the one from `Phrase`.

3. The result is to compare the processed version of the `Translated-Phrase` instance with its own reverse. Since “recognize” isn’t a palindrome, the result is `false`.

Case 2: Listing 7.15 and Listing 7.16

1. `frase.palindrome?` calls `palindrome?` on the `frase` instance, which is a `TranslatedPhrase`. As in Case 1, there is no `palindrome?` method in the `TranslatedPhrase` object, so Ruby uses the one from `Phrase`.
2. The `palindrome?` method in `Phrase` calls the `processed_content` method. Since there now *is* a `processed_content` method in the `TranslatedPhrase` object, Ruby uses the one from `TranslatedPhrase` instead of the one in `Phrase`.
3. The result is to compare the processed version of `self.translation` with its own reverse. Since “reconocer” *is* a palindrome, the result is `true`.

¿Puedes «reconocer» un palındromo en español? (Can you “reconocer” [recognize] a palindrome in Spanish?) (Figure 7.8).[5]

Figure 7.8: Narciso se reconoce. (Narcissus recognizes himself.)

[5]John William Waterhouse, “Echo and Narcissus”, 1903 (detail). Image courtesy of Archivart/Alamy Stock Photo.

7.3.1 Exercises

1. After filling in the code in Listing 7.15, both `Phrase` and `TranslatedPhrase` have explicit calls to the `downcase` method. Eliminate this duplication by filling in Listing 7.17 to define an appropriate `processor` method that returns the lowercase version of the content.

Listing 7.17: Eliminating duplication with a `processor` method.
palindrome.rb

Click here to view code image

```
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Processes the string for palindrome testing.
  def processor(string)
    # FILL IN
  end
```

Click here to view code image

```
  # Returns content for palindrome testing.
  def processed_content
    processor(self)
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end

# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end

  # Processes the translation for palindrome testing.
  def processed_content
    processor(translation)
  end
end
```

7.4 Modifying Native Objects

As a final step in understanding Ruby inheritance, we're going to learn how to modify native Ruby objects. Specifically, we're going to add the `palindrome?` method from Listing 7.15 as a method on `String` objects themselves.

> The reader should be warned that what we're about to do can be somewhat controversial. As prominent Rubyist Avdi Grimm once put it, "Monkeypatching is Destroying Ruby" (links added):[6]
>
> [6] https://avdi.codes/why-monkeypatching-is-destroying-ruby/. The article uses both "monkeypatching" (no space) and "monkey patching" (with a space) to refer to the same thing.
>
> "Monkey_patching", for anyone who doesn't know, refers to the practice of extending or modifying existing code by changing classes at run-time. It is a powerful technique that has become popular in the Ruby community at least in part because the Ruby language makes it so easy. Any class can be re-opened at any time and amended in any way.
>
> I believe the term first arose in the Python community, as a derogatory term for a practice which that community tended to frown on. The Ruby community, on the other hand, has embraced the term and the practice with enthusiasm. I'm starting to think that the Pythonistas' attitude may have been justified.
>
> This advice reminds me of a scene from the movie *Troy*, in which Achilles('Αχιλλεύζ), the greatest warrior in Greece, is training his close confidant Patroclus (Πάτϱοϰλοζ, depicted in the film as Achilles' cousin). At one point in their mock swordfight, Achilles switches his wooden training sword from his right hand to his left, holding it up to Patroclus's neck. In response, Patroclus exclaims, "You told me never to change sword hands!" "Yes," replies Achilles. "When you know how to use it, you won't be taking *my* orders."
>
> Likewise, once we know when and why to extend built-in classes, we won't be taking orders from anti-monkeypatchers[7] (Figure 7.9).[8]

Figure 7.9: Patroclus and Achilles respectfully decline the advice of the anti-monkeypatchers.

[7] Avdi himself acknowledges that his phrasing is designed mainly to spark discussion:

> The title of this post is intended to be deliberately provocative… It's provocative because I want to get people talking about this issue. I don't actually think that monkey patching is "destroying" Ruby, but I do think the proliferation of the technique has real and troubling implications for Ruby's future.

So Avdi advocates using monkeypatching judiciously, not avoiding it entirely.

[8] Image courtesy of Historic Images/Alamy Stock Photo.

The ability to modify native objects is a powerful one, to be sure—a "sharp knife", as it were. But instead of passively accepting others' advice, we'll adhere to the philosophy espoused by David Heinemeier Hansson, creator of the Ruby on Rails web framework. As DHH puts it (Figure 7.10): "Don't let anyone tell you that a powerful technique is too scary or dangerous for you. Let it pique your curiosity instead."

Figure 7.10: DHH agrees that sharp knives are OK (when used with care).

With those caveats in mind, let's see how to add `palindrome?` to `String`. The trick is to remove `Phrase` and put `palindrome?` and `processed_content` into the `String` class itself.

Let's review where we are with the `Phrase` class (Listing 7.18). Note that we've removed the `TranslatedPhrase` class since we don't need it anymore, and we've also removed the comment before `Phrase`.

Listing 7.18: The current state of `Phrase`.
palindrome.rb

Click here to view code image

```
class Phrase < String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
```

```
  def palindrome?
    processed_content == processed_content.reverse
  end
end
```

Amazingly, Ruby lets us just open and modify the `String` class, even though it's defined by Ruby itself. That means we can accomplish our task by simply changing the first line of Listing 7.18, as seen in Listing 7.19.

Listing 7.19: Defining `palindrome?` on `String` itself.
palindrome.rb

Click here to view code image

```
class String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end
```

As required, our code still finds palindromes correctly (Figure 7.11), now with `String` in place of `Phrase`:[9]

Figure 7.11: Napoleon Bonaparte was able before being exiled to Elba.

[9]Image courtesy of Everett Collection/Shutterstock.

Click here to view code image

```
>> load "./palindrome.rb"
>> napoleonsLament = String.new("Able was I ere I saw Elba")
>> napoleonsLament.palindrome?
=> true
```

Even cooler, we can call `palindrome?` directly on string literals:

Click here to view code image

```
>> "foobar".palindrome?
=> false
>> "Racecar".palindrome?
=> true
>> "Able was I ere I saw Elba".palindrome?
=> true
```

This is a remarkable achievement, but it's worth asking whether adding `palindrome?` to `String` itself is really a good idea. The answer depends in part on the culture of the language. As indicated by Avdi's post, Ruby is relatively tolerant of adding methods to native objects, as long as the privilege isn't abused. In my view, adding `palindrome?` is marginal but defensible; I would be tempted to make the change in a shell script or small project, but might decline to monkeypatch `String` in a larger project with more opportunities for dangerous interactions with other code. In any case, if you want to add `palindrome?` to `String` yourself, neither I nor Achilles is going to stop you.

7.4.1 Exercises

1. Add a `blank?` method to the `String` class that returns true if the string is empty *or* consists solely of whitespace. *Hint*: Use a regular expression (Section 4.3). You will need the regex syntax for the start and end of a line (Figure 7.12). (The Ruby on Rails framework adds a more advanced version of `blank?` that handles multiple different kinds of whitespace.)

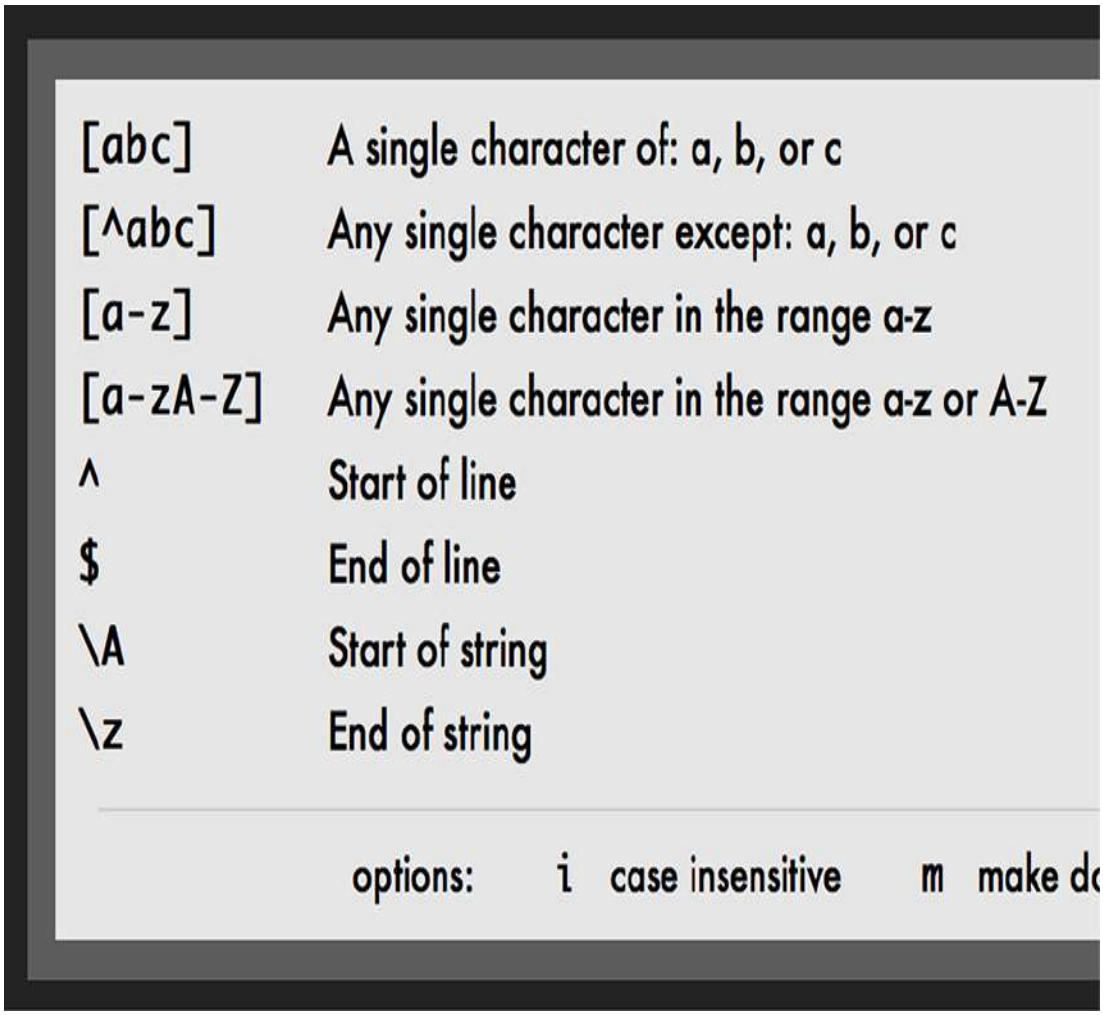


Figure 7.12: Start to end, a blank string is all whitespace.

2. As currently written, Listing 7.19 exposes not only `palindrome?` as a string method, but also `processed_content`. The latter method is really just for the internal use of `palindrome?`, though, so it's a better practice to hide it from outsiders by using the `private` keyword. Show that the code in Listing 7.20 still defines `palindrome?`, while hiding the `processed_content` method. What happens if you call `"racecar".processed_content`?[10]

[10] The extra level of indentation on the `processed_content` method is designed to make it visually apparent which methods are defined after `private`. Experience shows that this is a wise practice; in classes with a large number of methods, it is easy to define a private method accidentally, which leads to considerable confusion when it isn't available to call on the corresponding object.

Listing 7.20: Factoring `processed_content` into a `private` method.
palindrome.rb

Click here to view code image

```
class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

7.5 Modules

The final detail in our study of Ruby classes is *modules*, also called *mixins*. Ruby modules give us a way to factor out common functionality and then mix it in to multiple classes. That's a bit abstract, though, so let's look at a concrete example.

At this point in the tutorial, we've added a `palindrome?` method to the `String` class, using code that looks like Listing 7.21, where we've incorporated the results of the second exercise in Section 7.4.1 to ensure that the `processed_content` method is `private` and hence not exposed to users.

Listing 7.21: Factoring `processed_content` into a `private` method.
palindrome.rb

Click here to view code image

```
class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

As it turns out, strings aren't the only things that can be palindromes— integers can be palindromes, too. For example, the number 12321 is a palindrome because

its digits are the same forward and backward. This suggests defining a `palindrome?` method on the `Integer` class, as shown in Listing 7.22.

Listing 7.22: Defining a `palindrome?` method for integers.

Click here to view code image

```
class Integer

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.to_s
    end
end
```

Note that we've used the `to_s` method to convert the integer to a string, and then applied the same `palindrome?` method as before. In fact, the palindrome code in Listing 7.21 and Listing 7.22 is identical apart from

```
self.downcase
```

and

```
self.to_s
```

Since calling `to_s` on a string just returns the string, and calling `downcase` on a string of digits just returns the digits, we can combine these two methods like this:

```
self.to_s.downcase
```

This will now work on both strings and integers.

We could just include the full contents of Listing 7.21 and Listing 7.22 into the `palindrome.rb` file, but that would be a violation of the DRY principle, and keeping

the changes in sync could quickly get cumbersome. The solution is to factor the common palindrome code into a *module*, as shown in Listing 7.23.

Listing 7.23: Factoring the palindrome code into a module.
palindrome.rb

Click here to view code image

```
module Palindrome

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.to_s.downcase
    end
end
```

We can then `include` the `Palindrome` module into both the `String` and `Integer` classes, and they'll automatically get all the same methods (in this case, just `palindrome?` and the private `processed_content` method), as shown in Listing 7.24.

Listing 7.24: Including the `Palindrome` module.
palindrome.rb

Click here to view code image

```
module Palindrome

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
  private

    # Returns content for palindrome testing.
    def processed_content
      self.to_s.downcase
    end
end

class String
```

```
  include Palindrome
end

class Integer
  include Palindrome
end
```

At this point, both strings and integers have a `palindrome?` method! We can verify this using irb:

Click here to view code image

```
>> load "./palindrome.rb"
>> "Racecar".palindrome?
=> true
>> 12321.palindrome?
=> true
```

Ruby modules are a common technique for eliminating duplication, and many important Ruby methods are correspondingly part of modules rather than classes. For example, all the methods discussed in Section 6.3.4—including `map`/`collect`, `find_all`/`select`, and `reduce`/`inject`—are defined in the `Enumerable` module (see, e.g., my talk "Enumerable for Fun and Profit", mentioned in Section 6.3.4). As a result, any class that mixes `Enumerable` in— such as `Array`, `Range`, and `Hash`—automatically gets a huge amount of functionality for free.

7.5.1 Exercises

1. Reusable modules are usually defined as separate files and then are included into the corresponding class files using `require`. Implement this practice by factoring Listing 7.24 into three separate files: `palindrome.rb` (the module), `string_palindrome.rb` (to `include` it into `String`), and `integer_palindrome.rb` (to `include` it into `Integer`). Load all three into the REPL and confirm that it worked.

2. Using methods from the documentation for the `Enumerable` module, determine the maximum, minimum, and sum of the range `1..100`.

Chapter 8

Testing and Test-Driven Development

Although rarely covered in introductory programming tutorials, *automated testing* is one of the most important subjects in modern software development. Accordingly, this chapter includes an introduction to testing in Ruby, including a first look at *test-driven development*, or TDD.

> Test-driven development came up briefly in Section 6.3.3, which promised that we would use testing techniques to add an important capability to finding palindromes, namely, being able to detect complicated palindromes such as “A man, a plan, a canal—Panama!” (Figure 6.7) or “Madam, I’m Adam.” (Figure 8.1).[1] This chapter fulfills that promise.
>
> [1] “The Temptation of Adam” by Tintoretto. Image courtesy of Album/Alamy Stock Photo.

Figure 8.1: The Garden of Eden had it all—even palindromes.

As it turns out, learning how to write Ruby tests will also give us a chance to learn how to create (and publish!) a Ruby gem, another exceptionally useful Ruby skill rarely covered in introductory tutorials.

Here's our strategy for testing the current palindrome code and extending it to more complicated phrases:

1. Set up our system for automated testing (Section 8.1).
2. Write automated tests for the existing `palindrome?` functionality (Section 8.2).
3. Write a *failing* test for the enhanced palindrome detector (RED) (Section 8.3).
4. Write (possibly ugly) code to get the test *passing* (GREEN) (Section 8.4).
5. *Refactor* the code to make it prettier, while ensuring that the test suite stays GREEN (Section 8.5).

8.1 Testing and Ruby Gem Setup

We saw as early as Section 1.5 that the Ruby ecosystem includes a large number of self-contained packages called *gems*. In this section, we'll create a gem based on the palindrome detector developed in Chapter 7; along the way, we'll see that gem creation leads naturally to the generation of a sample *test suite* to test our code.

In Section 1.5.1, we briefly encountered *Bundler* when deploying our hello app to production. Bundler's principal use is in bundling together all the gem dependencies for a given application, but it also includes the ability to generate the skeleton of a brand-new Ruby gem.

Getting started with a new gem is easy using the `bundle` command, which can create a gem skeleton using `bundle gem`. Since we'll be publishing the gem publicly, you'll need to pick a unique name for your palindrome gem; I suggest using `<username>_palindrome`, where `<username>` is your unique username. (I'll use `mhartl_palindrome`.) The result appears in Listing 8.1.

Listing 8.1: Creating a new gem with Bundler.

Click here to view code image

```
$ cd ~/repos    # Use ~/environment/repos on Cloud9
$ bundle gem <username>_palindrome
Do you want to generate tests with your gem?
Type 'rspec' or 'minitest' to generate those test files now and in the future.
rspec/minitest/(none): minitest
Do you want to license your code permissively under the MIT license? y/(n): n
Do you want to include a code of conduct in gems you generate? y/(n): n
```

Listing 8.1 shows the interactive prompt that shows up the first time you generate a gem, which asks you to choose a test framework and decide whether or not to include a license and a code of conduct. We'll be using a simple yet powerful test framework called minitest, so enter `minitest` at the first prompt; for the other two, I recommend using the default answer of "no", but once you're ready for others to start using your gem you should consider adding an open-source license (in a file called `LICENSE.txt`),[2] and when you're ready for collaborators you should consider adding a code of conduct (in a file called `CODE_OF_CONDUCT.md`).[3] Bundler will remember these defaults for use in generating future gems, but you can use the `bundle config` command to change the defaults at any time.[4]

[2] The most common choice of license in the Ruby community is probably the MIT License, although Ruby itself is released under the Ruby License.

[3] Possible choices include the Contributor Covenant Code of Conduct and the Ruby Community Conduct Guideline.

[4] For example, running `bundle config set gem.test rspec` would switch to using the RSpec testing framework by default, and `bundle config set gem.mit true` would automatically add the MIT License for future invocations of `bundle gem`. Note that these settings affect only *future* gems, though; you'll still have to make changes to any existing gems by hand (e.g., by downloading a copy of the MIT License and copying it to the gem's local directory).

After creating the gem skeleton, `cd` into the directory and put the project under version control with Git:

Click here to view code image

```
$ cd <username>_palindrome
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

At this point, I recommend creating a public repository for the gem at GitHub by following the same basic steps from Section 1.5.1 (replacing "GitLab" with "GitHub" as necessary). This will also give you a GitHub repo URL for use in the next step.

Next, add the required summary and description (along with a project homepage) to the *gemspec* file generated by Bundler using your editor of choice (Listing 8.2). Note that the generated `gemspec` uses the notation `%q{...}` as an alternate way to write strings, which avoids having to worry about escaping out single or double quotes.[5]

[5] Another alternative for making strings, and my personal favorite, is simply to use percent-parentheses, like this: `%(...)`.

Listing 8.2: Adding the necessary gem information.
<username>_palindrome.gemspec

Click here to view code image

```
lib = File.expand_path("../lib", __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require "mhartl_palindrome/version"

Gem::Specification.new do |spec|
  spec.name          = "mhartl_palindrome"
  spec.version       = MhartlPalindrome::VERSION
  spec.authors       = ["Michael Hartl"]
  spec.email         = ["michael@michaelhartl.com"]

  spec.summary       = %q{Palindrome detector}
  spec.description   = %q{Learn Enough Ruby palindrome detector}
  spec.homepage      = "https://github.com/mhartl/mhartl_palindrome"
  spec.license       = "MIT"

  # Prevent pushing this gem to RubyGems.org. To allow pushes either set the
  # 'allowed_push_host'
  # to allow pushing to a single host or delete this section to allow pushing to
  # any host.
  if spec.respond_to? (:metadata)
    spec.metadata["allowed_push_host"] = "https://rubygems.org/"
  else
```

```
    raise "RubyGems 2.0 or newer is required to protect against " \
      "public gem pushes."
  end

  # Specify which files should be added to the gem when it is released.
  # The `git ls-files -z` loads the files in the RubyGem
  # that have been added into git.
  spec.files         = Dir.chdir(File.expand_path('..', __FILE__)) do
   `git ls-files -z`.split("\x0").reject do
     |f| f.match(%r{^(test|spec|features)/})
   end
  end
  spec.bindir        = "exe"
  spec.executables   = spec.files.grep(%r{^exe/}) { |f| File.basename(f) }
  spec.require_paths = ["lib"]

  spec.add_development_dependency "bundler", "~> 1.16"
  spec.add_development_dependency "rake", "~> 10.0"
  spec.add_development_dependency "minitest", "~> 5.0"
end
```

Finally, as a bit of preparation for our testing sequence, add the `mini-test-reporters` gem in the `Gemfile`, as shown in Listing 8.3. This gem adds color to the testing output in line with the "Red, Green, Refactor" pattern we'll be developing in the rest of the chapter.

Listing 8.3: Adding a gem for nice test output.
Gemfile

Click here to view code image

```
source "https://rubygems.org"

git_source(:github)  {|repo_name| "https://github.com/#{repo_name}" }

# Specify your gem's dependencies in <username>_palindrome.gemspec
gemspec

gem 'minitest-reporters', '1.2.0'
```

You'll also need to update the "test helper" file to include the reporters (whose effect we'll see later in this section), as shown in Listing 8.4.

Listing 8.4: Configuring the test helper.
test/test_helper.rb

Click here to view code image

```
$LOAD_PATH.unshift File.expand_path("../../lib", __FILE__)
require "mhartl_palindrome"
require "minitest/autorun"
require "minitest/reporters"
Minitest::Reporters.use!
```

Now install all necessary gems using Bundler and we'll be good to go:

```
$ bundle _2.3.10_ install
```

Before moving on to testing and extending our gem, let's take a moment to run our *default test suite*, which `bundle gem` generated automatically as part of Listing 8.1. We can run the test suite using the `rake` utility program (short for "Ruby make"), preceded by `bundle exec` to use the exact gem environment specified by our `Gemfile` (Listing 8.5).[6]

[6] Many Ruby developers alias the rather verbose `bundle exec` command to the much shorter `be` by putting `alias be='bundle exec'` in their `.bash_profile` file.

Listing 8.5: Running our test suite for the first time. RED

Click here to view code image

```
$ bundle exec rake test
Started with run options --seed 57409

 FAIL["test_it_does_something_useful", "MhartlPalindromeTest",
 0.0007609999738633633]
 test_it_does_something_useful#MhartlPalindromeTest (0.00s)
        Expected false to be truthy.
        /Users/mhartl/repos/mhartl_palindrome/test/mhartl_palindrome_test.rb:9:in
        `test_it_does_something_useful'

  2/2: [==================================] 100% Time: 00:00:00, Time: 00:00:00

Finished in 0.00119s
2 tests, 2 assertions, 1 failures, 0 errors, 0 skips
```

Don't worry about the details of the results of this command (which will vary by system). The key takeaway is that the suite is *failing* by default, as seen in the last line; as seen in Figure 8.2, it appears as RED in a terminal window (which is why we added `minitest-reporters` in Listing 8.3).

Figure 8.2: The RED state of the default test suite.

8.1.1 Exercises

1. It's a good practice to include a "README" file with information about the module. Create a file with the name `README.md` and fill it with information about the module. You can use my README as a reference if you like. The result will automatically be nicely formatted at GitHub (Figure 8.3).

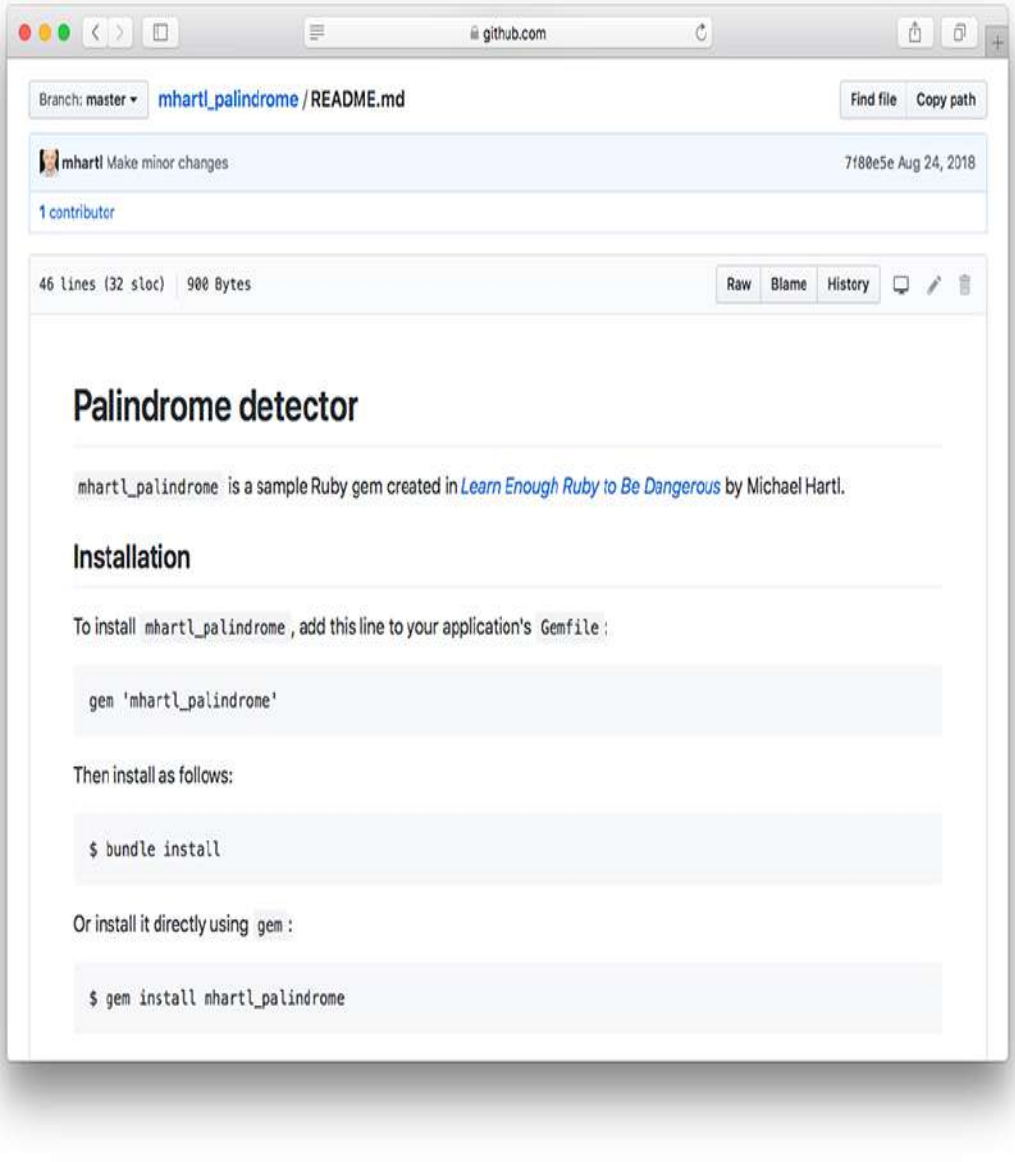


Figure 8.3: The README file at GitHub.

8.2 Initial Test Coverage

With the preparation from Section 8.1 done, we're now ready to get started with our automated tests. We ended Section 8.1 with a RED test suite; in this section, we'll get it to GREEN, and en route we'll write tests for the current palindrome-detection functionality.

Listing 8.6 shows the default test code itself.[7] Listing 8.6 is code generated by Bundler, and in general you don't need to understand all the details of generated code, but you might be able to guess the gist of it: First, it confirms that the gem has a *version number* (using a variant of `refute`, covered in Section 8.2.2), and

then it *asserts* something using the `assert` function. This kind of construction—called, appropriately enough, an *assertion*—is one of the most common ways to test code. (It will look especially familiar if you completed *Learn Enough JavaScript to Be Dangerous*.)

[7] This is exactly the kind of detail that often changes from version to version. It's possible that future versions of minitest will have a different default test suite. If your results don't match exactly, use your technical sophistication (Box 1.1) to resolve any discrepancies.

Note: For simplicity and concreteness, code samples throughout the rest of this tutorial will use my username (mhartl), but you should substitute your own.

Listing 8.6: The default test suite. RED
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test
  def test_that_it_has_a_version_number
    refute_nil ::MhartlPalindrome::VERSION
  end

  def test_it_does_something_useful
    assert false
  end
end
```

A plain assertion (using `assert` alone) passes if the argument is `true`. Since the assertion in Listing 8.6 literally asserts `false`, it fails by design:

Listing 8.7: RED

Click here to view code image

```
$ bundle exec rake test
2 tests, 2 assertions, 1 failures, 0 errors, 0 skips
```

(This is the same result that we got in Listing 8.5; I've omitted the other output for brevity.)

Our first step is to get the test suite passing (GREEN), which we can do (just as an example) by changing `false` to `true`, as seen in Listing 8.8.

Listing 8.8: Getting a passing test suite. GREEN
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class Mhartl::PalindromeTest < Minitest::Test
  def test_that_it_has_a_version_number
    refute_nil ::Mhartl::Palindrome::VERSION
  end

  def test_it_does_something_useful
    assert true
  end
end
```

We can confirm at the command line that it worked:

Listing 8.9: GREEN

Click here to view code image

```
$ bundle exec rake test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

Now let's get started with our gem's main application code, which is located in the `lib/` ("library") directory. The generated gem includes a Ruby module (Section 7.5), as shown in Listing 8.10.

Listing 8.10: The generated gem module.
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

module MhartlPalindrome
  # Your code goes here...
end
```

In place of the default module, we'll instead start with a `class`, as we used in most of Chapter 7. (Guidance on restoring the module is provided in the exercises for Section 8.5 (Section 8.5.2).) In particular, we'll copy the code from Listing 7.20 into `lib/<username>_palindrome.rb`, as shown in Listing 8.11.

Listing 8.11: Putting the palindrome detector in the gem.
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

Now we're ready to start making some tests to check that the code in Listing 8.11 is actually working. We'll start with a negative case:

Click here to view code image

```
def test_non_palindrome
  assert !"apple".palindrome?
end
```

Here we've used `assert` to assert that `"apple"` should *not* be a palindrome (Figure 8.4),[8] where "not" is indicated with an exclamation point ("bang") `!` as usual (Section 2.4.1). Note that the function call has higher *precedence*, so it happens before the negation; as a result, we don't need parentheses after the `!`.

[8] Image courtesy of Glayan/Shutterstock.

Figure 8.4: The word “apple”: not a palindrome.

In similar fashion, we can test a plain palindrome (one that’s literally the same forward and backward) with another `assert`:

Click here to view code image

```
def test_literal_palindrome
  assert "racecar".palindrome?
end
```

Combining the code from the above discussion (and eliminating the generated tests from Listing 8.6 and Listing 8.8) gives us our initial non-generated test file, as shown in Listing 8.12.

Listing 8.12: Our initial (non-generated) test suite.
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    assert !"apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end
end
```

Now for the real test (so to speak):

Listing 8.13: GREEN

Click here to view code image

```
$ bundle exec rake test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

The tests are now GREEN, indicating that they are in a passing state. That means our code is working!

8.2.1 Pending Tests

Before moving on, we'll add a couple of *pending* tests, which are placeholders/reminders for tests we want to write. The way to write a pending test is simply to use `skip`, as shown in Listing 8.14.

Listing 8.14: Adding two pending tests. YELLOW
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    assert !"apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    skip
```

```
  end

  def test_palindrome_with_punctuation
    skip
  end
end
```

We can see the result of Listing 8.14 by rerunning the test suite:

Listing 8.15: YELLOW

Click here to view code image

```
$ bundle exec rake test
4 tests, 2 assertions, 0 failures, 0 errors, 2 skips
```

Now the test runner displays indications that there are two “skips” representing pending tests. Sometimes people speak of a test suite with pending tests as being YELLOW, in analogy with the red-yellow-green color scheme of traffic lights (Figure 8.5), although it’s also common to refer to any non-RED test suite as GREEN.

Figure 8.5: A YELLOW (pending) test suite.

Filling in the test for a mixed-case palindrome is left as an exercise (with a solution shown in the next section), while filling in the second pending test is the subject of Section 8.3 and Section 8.4.

8.2.2 Exercises

1. By filling in the code in Listing 8.16, add a test for a mixed-case palindrome like “RaceCar”. Is the test suite still GREEN (or YELLOW)?
2. In order to make 100% sure that the tests are testing what we *think* they’re testing, it’s a good practice to get to a failing state (RED) by intentionally *breaking* the tests. Change the application code to break each of the existing tests in turn, and then confirm that they are GREEN again once the original code has been restored. An example of code that breaks the test in the previous exercise (but not the other tests) appears in Listing 8.17. (One advantage of writing the tests *first* is that this RED–GREEN cycle happens automatically.)
3. The most common method for making minitest assertions is, appropriately enough, `assert`, but there’s a second method called `refute` for making *negative* assertions—i.e., asserting that something is *not* true. In your test suite, replace `assert !"apple".palindrome?` with `refute "apple".palindrome?` and show that the test suite is still GREEN/YELLOW.

Listing 8.16: Adding a test for a mixed-case palindrome.
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    assert !"apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    # FILL IN
  end

  def test_palindrome_with_punctuation
    skip
  end
end
```

Listing 8.17: Intentionally breaking a test. RED
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"
```

```
class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self
    end
end
```

8.3 Red

In this section, we'll take the important first step toward being able to detect more complex palindromes like "Madam, I'm Adam." and "A man, a plan, a canal—Panama!". Unlike the previous strings we've encountered, these phrases—which contain both spaces and punctuation—aren't strictly palindromes in a literal sense, even if we ignore capitalization. Instead of testing the strings as they are, we have to figure out a way to select only the letters, and then see if the resulting letters are the same forward and backward.

The code to do this is fairly tricky, but the tests for it are simple. This is one of the situations where test-driven development particularly shines (Box 8.1). We can write our simple tests, thereby getting to RED, and then write the application code any way we want to get to GREEN (Section 8.4). At that point, with the tests protecting us against undiscovered errors, we can change the application code with confidence (Section 8.5).

Box 8.1. When to Test

When deciding when and how to test, it's helpful to understand *why* to test. In my view, writing automated tests has three main benefits:

1. Tests protect against *regressions*, where a functioning feature stops working for some reason.
2. Tests allow code to be *refactored* (i.e., changing its form without changing its function) with greater confidence.
3. Tests act as a *client* for the application code, thereby helping determine its design and its interface with other parts of the system.

Although none of the above benefits *require* that tests be written first, there are many circumstances where test-driven development (TDD) is a valuable tool to have in your kit. Deciding when and how to test depends in part on how

comfortable you are writing tests; many developers find that, as they get better at writing tests, they are more inclined to write them first. It also depends on how difficult the test is relative to the application code, how precisely the desired features are known, and how likely the feature is to break in the future.

In this context, it's helpful to have a set of guidelines on when we should test first (or test at all). Here are some suggestions based on my own experience:

- When a test is especially short or simple compared to the application code it tests, lean toward writing the test first.
- When the desired behavior isn't yet crystal clear, lean toward writing the application code first, then write a test to codify the result.
- Whenever a bug is found, write a test to reproduce it and protect against regressions, then write the application code to fix it.
- Write tests before refactoring code, focusing on testing error-prone code that's especially likely to break.

We'll start by writing a test for a palindrome with punctuation, which just parallels the tests from Listing 8.12:

Click here to view code image

```
def test_palindrome_with_punctuation
  assert "Madam, I'm Adam.".palindrome?
end
```

The updated test suite appears in Listing 8.18, which also includes the solution to a couple of exercises in Listing 8.16 (Figure 8.6).[9]

[9] Image courtesy of msyaraafiq/Shutterstock.

Figure 8.6: “RaceCar” is still a palindrome (ignoring case).

Listing 8.18: Adding a test for a punctuated palindrome. RED
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end
end
```

As required, the test suite is now RED:

Listing 8.19: RED

Click here to view code image

```
$ bundle exec rake test
4 tests, 4 assertions, 1 failures, 0 errors, 0 skips
```

At this point, we can start thinking about how to write the application code and get to GREEN. Our strategy will be to write a `letters` method that returns only the letters in the content string. In other words, the code

```
"Madam, I'm Adam.".letters
```

should evaluate to this:

```
MadamImAdam
```

Getting to that state will allow us to use our current palindrome detector to determine whether the original phrase is a palindrome or not.

Having made this specification, we can now write a simple test for `letters`. We could follow the pattern from previous tests and assert (strict) equality directly (Listing 8.20).

Listing 8.20: Asserting strict equality directly.

Click here to view code image

```
assert "Madam, I'm Adam.".letters == "MadamImAdam"
```

It turns out, though, that the `assert` module has native support for this kind of comparison (as seen in the official documentation for minitest assertions), leading to assertions of the form shown in Listing 8.21.[10]

[10] If you've studied JavaScript, it's worth noting that the order of `<actual>` and `<expected>` is reversed in Node's `assert` library, which follows the convention `assert.equal(<actual>, <expected>)` (or `assert.strictEqual(<actual>, <expected>)`, as discussed in Section 8.3 of *Learn Enough JavaScript to Be Dangerous*).

Listing 8.21: Using a native assertion.

Click here to view code image

```
assert_equal <expected>, <actual>
```

As we'll see in a moment, it's generally preferable to use native assertions when possible, since doing so leads to more helpful messages for failed tests. For the sake of such failing test messages, it's also important to include the arguments in the "expected, actual" order shown above.

In the present case, the "actual" result is `"Madam, I'm Adam.".letters`, and the "expected" value is `"MadamImAdam"`, so we can fill in the assertion as follows:

Click here to view code image

```
assert_equal "MadamImAdam", "Madam, I'm Adam.".letters
```

The new test appears with the others in Listing 8.22.

Listing 8.22: Adding a test for the `letters` method. RED
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end

  def test_letters
    assert_equal "MadamImAdam", "Madam, I'm Adam.".letters
  end
end
```

Because the `letters` method doesn't exist at all, the current failing message isn't all that helpful (indeed, it's an "error" rather than "failing"):

Listing 8.23: RED

Click here to view code image

```
$ bundle exec rake test
NoMethodError: undefined method `letters' for "Madam, I'm Adam.":String
5 tests, 4 assertions, 0 failures, 1 errors, 0 skips
```

We can get to a more useful RED state by adding a *stub* for `letters`: a method that doesn't work, but at least exists. For simplicity, we'll simply return nothing, as shown in Listing 8.24.[11]

[11] Arguably, `letters` should be a `private` method like `processed_content`. If we made this design choice, we wouldn't be able to test it directly by calling it on a string, as that would give a `NoMethodError`. We can route around this restriction using the `send` method, which takes in a symbol (Section 4.4.1) and calls the corresponding method on the object, as in `"Madam, I'm Adam.".send(:letters)`.

Listing 8.24: A stub for the `letters` method. RED
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

As promised, the error message is now quite helpful, as seen in Listing 8.25.

Listing 8.25: RED

Click here to view code image

```
$ bundle exec rake test
 FAIL["test_letters", "MhartlPalindromeTest", 0.0007690000347793102]
 test_letters#MhartlPalindromeTest (0.00s)
        Expected: "MadamImAdam"
          Actual: nil
5 tests, 5 assertions, 2 failures, 0 errors, 0 skips
```

With our two RED tests capturing the desired behavior, we're now ready to move on to the application code and try getting it to GREEN.

8.3.1 Exercises

1. What is the error message when using the direct `==` assertion shown in Listing 8.20? Why is this less useful than the message in Listing 8.25?
2. What happens if you reverse the actual and expected values (Listing 8.21) in Listing 8.25? Why is the resulting error message confusing?

8.4 Green

Now that we have RED tests to capture the enhanced behavior of our palindrome detector, it's time to make them GREEN. Part of the philosophy of TDD is to get them passing without worrying too much at first about the quality of the implementation. Once the test suite is GREEN, we can polish it up without introducing regressions (Box 8.1).

The main challenge is implementing `letters`, which returns a string of the letters (but not any other characters) making up the `content` of a `String`. In other words, we need to select the characters that match a certain pattern. This sounds like a job for regular expressions (Section 4.3).

At times like these, using an online regex matcher with a regex reference like the one shown in Figure 4.5 is an excellent idea. Indeed, sometimes they make things a little *too* easy, such as when the reference has the exact regex you need (Figure 8.7).

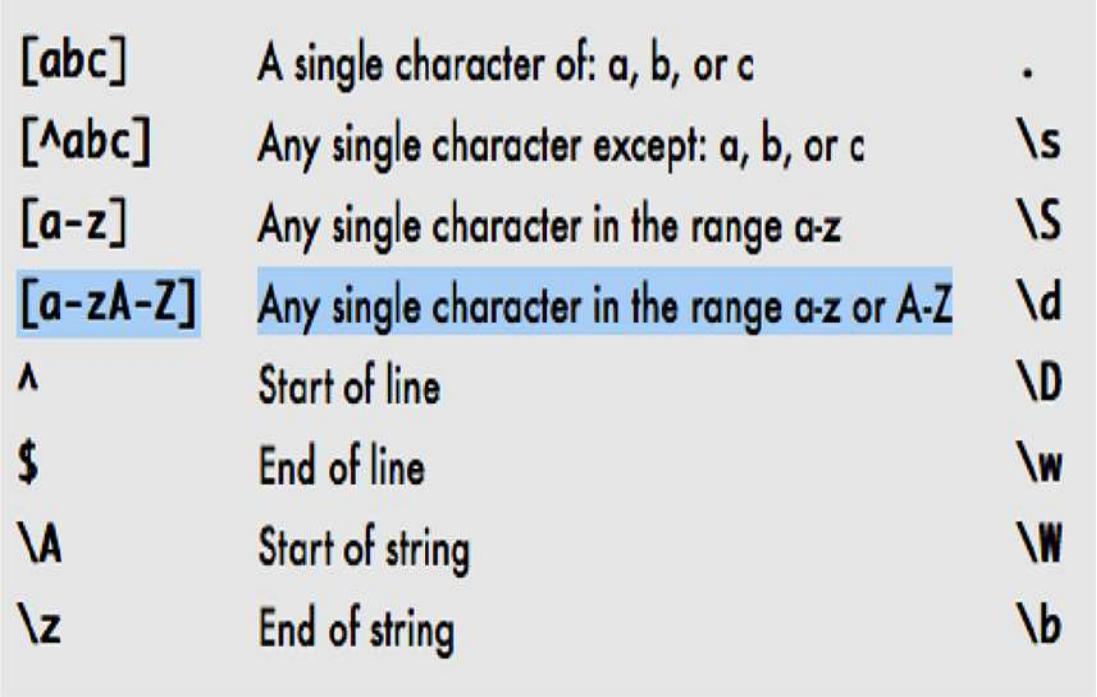


Figure 8.7: The exact regex we need.

Let's test it in the console to make sure it satisfies our criteria (using the `match?` method, which is like the `match` method introduced in Section 4.3 but returns a boolean value):[12]

[12] Note that this won't work for non-ASCII characters. If you need to match words containing such characters, the Google search ruby unicode letter regular expression might be helpful. Thanks to reader Paul Gemperle for pointing out this issue.

Click here to view code image

```
$ irb
>> "M".match?(/[a-zA-Z]/)
=> true
>> "d".match?(/[a-zA-Z]/)
=> true
>> ",".match?(/[a-zA-Z]/)
=> false
```

Lookin' good!

We're now in a position to build up an array of characters that matches upper- or lowercase letters. The most straightforward way to do this is with the `for` loop method we first saw in Section 2.6. We'll start with an array for the letters, and then iterate through the content string, pushing each character onto the array (Section 3.4.2) if it matches the letter regex:

Click here to view code image

```
the_letters = []
for i in 0..self.length - 1
  if (self[i].match(/[a-zA-Z]/))
    the_letters << self[i]
  end
end
```

Note that we've used `match` here in place of `match?`; although the latter is arguably more precise, the two are equivalent inside an `if` clause, and using `match` in this context is common in idiomatic Ruby code.

At this point, `the_letters` is an array of letters, which can be `join`ed to form a string of the letters in the original string:

```
the_letters.join
```

Putting everything together gives the `String#letter` method in Listing 8.26 (with a highlight added to indicate the beginning of the new method).

Listing 8.26: A working `letters` method (but full suite still RED).
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    the_letters = []
    for i in 0..self.length - 1
      if self[i].match(/[a-zA-Z]/)
        the_letters << self[i]
      end
    end
    the_letters.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

Although the full test suite is still RED, our `letters` test should now be GREEN:

Listing 8.27: RED

Click here to view code image

```
$ bundle exec rake test
5 tests, 5 assertions, 1 failures, 0 errors, 0 skips
```

We can get the final RED test to pass by replacing `self` with `self.letters` in the `processed_content` method. The result appears in Listing 8.28.

Listing 8.28: A working `palindrome?` method. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
```

```
  def letters
    the_letters = []
    for i in 0..self.length - 1
      if self[i].match(/[a-zA-Z]/)
        the_letters << self[i]
      end
    end
    the_letters.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.letters.downcase
    end
end
```

The result of Listing 8.28 is a GREEN test suite (Figure 8.8):[13]

[13] Image courtesy of Album/Alamy Stock Photo.

Figure 8.8: Our detector finally understands Adam's palindromic nature.

Listing 8.29: GREEN

Click here to view code image

```
$ bundle exec rake test
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

It may not be the prettiest code in the world, but this GREEN test suite means our code is working!

8.4.1 Exercises

1. By `require`-ing the `palindrome` gem in irb, verify by hand that the `String#palindrome?` code can successfully detect palindromes of the form "Madam, I'm Adam." (You may have to quit and restart the REPL to refresh all relevant object definitions. As discussed further in Section 8.5.1, you will also have to install the gem locally using `bundle exec rake install`.)

8.5 Refactor

Although the code in Listing 8.28 is now working, as evidenced by our GREEN test suite, it relies on a rather cumbersome (and very un-Rubyish) `for` loop, and there's some duplication as well. In this section, we'll *refactor* our code, which is the process of changing the form of code without changing its function.

By running our test suite after any significant changes, we'll catch any regressions quickly, thereby giving us confidence that the final form of the refactored code is still correct. Throughout this section, I suggest making changes incrementally and running the test suite after each change to confirm that the suite is still GREEN.

We start by observing that there's some duplication in Listing 8.28: The expression

```
self[i]
```

appears twice. This suggests eliminating the duplication by binding it to a variable, which we'll call `character`:

Click here to view code image

```
def letters
  the_letters = []
  for i in 0..self.length - 1
    character = self[i]
    if character.match(/[a-zA-Z]/)
      the_letters << character
    end
  end
```

```
  the_letters.join
end
```

As another bit of polish, we can simplify the regex by using `i` after `/.../` to make a case-insensitive match, while also binding it to a name to make its purpose clearer:

Click here to view code image

```
letter_regex = /[a-z]/i
for i in 0..self.length - 1
  character = self[i]
  if character.match(letter_regex)
    the_letters << character
  end
end
```

Per Section 3.5, it's usually better to use an `each` loop when we can. We can do this by combining the `String#chars` method mentioned briefly in Section 5.3 and an `each` loop as follows:

Click here to view code image

```
letter_regex = /[a-z]/i
self.chars.each do |character|
   if character.match(letter_regex)
     the_letters << character
   end
end
```

Notice that this has the side effect of eliminating the assignment we just made, replacing it with a loop variable.

We've got some more refactoring to do, but for reference the full state of the application code appears in Listing 8.30.

Listing 8.30: A refactored `letters` method. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    the_letters = []
    letter_regex = /[a-z]/i
    self.chars.each do |character|
      if character.match(letter_regex)
```

```
          the_letters << character
        end
      end
      the_letters.join
    end

    private

      # Returns content for palindrome testing.
      def processed_content
        self.letters.downcase
      end
end
```

The result of running the test suite is gratifying:

Listing 8.31: GREEN

Click here to view code image

```
$ bundle exec rake test
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

It's still GREEN! The changes above involved lots of tricky and error-prone manipulations, so a GREEN test suite gives us confidence that we didn't introduce any regressions.

To motivate a penultimate refactoring, we can note that the form of the code in Listing 8.30 is similar to that in Listing 6.4 from Section 6.2: we initialize an empty array and then `push` to it (using the shovel operator `<<`) in an `each` loop. In Listing 6.5, we used functional programming via the `select` method to convert that loop to a single line, and we can do exactly the same thing here.

As a quick refresher, let's drop into the REPL:

Click here to view code image

```
>> "Madam, I'm Adam.".chars
=> ["M", "a", "d", "a", "m", ",", " ", "I", "'", "m", " ",
    "A", "d", "a", "m", "."]
>> "Madam, I'm Adam".chars.select { |c| c.match(/[a-z]/i) }
=> ["M", "a", "d", "a", "m", "I", "m", "A", "d", "a", "m"]
>> "Madam, I'm Adam".chars.select { |c| c.match(/[a-z]/i) }.join
=> "MadamImAdam"
```

We see here how combining method chaining (Section 5.3) with functional programming makes it easy to select and join the letter characters in a string.

Applying `select` to the code in Listing 8.30, we can condense the `letters` method into a single line, as shown in Listing 8.32. (It could arguably be improved by retaining the `lettersRegEx` constant from Listing 8.30, but I find the austerity of a one-line function to be nearly impossible to resist.)

Listing 8.32: Refactoring `letters` down to a single line. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    self.chars.select { |c| c.match(/[a-z]/i) }.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.letters.downcase
    end
end
```

As noted in Chapter 6, functional programs are harder to build up incrementally, which is one reason why it's so nice to have a test suite to check that it had its intended effect:[14]

[14] IRL, I would probably write the `Phrase#letters` method by first writing the tests we saw in Section 8.3, and then try for a functional solution right away. If I failed at that, I would backtrack, do it an easier (loopier) way, and then make another run at a functional solution after getting the test suite GREEN. (I find this sort of backtracking to be especially necessary with the `reduce`/`inject` method we met in Section 6.3.)

Listing 8.33: GREEN

Click here to view code image

```
$ bundle exec rake test
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

Huzzah! Our test suite still passes, so our one-line `letters` method works.

This is a major improvement, but in fact there's one more refactoring that represents a great example of the power of Ruby. We'll start by *removing* the test for `letters`, as shown in Listing 8.34.

Listing 8.34: Removing the `letters` test. GREEN
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end
end
```

Now for some yummy Ruby goodness. Recall from Section 4.3 that strings support a `scan` method that lets us select regex-matching characters right from a string:

Click here to view code image

```
>> "Madam, I'm Adam.".scan(/[a-z]/i)
=> ["M", "a", "d", "a", "m", "I", "m", "A", "d", "a", "m"]
>> "Madam, I'm Adam.".scan(/[a-z]/i).join
=> "MadamImAdam"
```

By scanning on the same regex we've been using throughout this section and then joining on the empty string, we've replicated the functionality of the `letters` method! This means we can simplify the application code even further by eliminating `letters` entirely, as shown in Listing 8.35.

Listing 8.35: Replacing `letters` with a `scan`. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.scan(/[a-z]/i).join.downcase
```

```
    end
end
```

Per the exercise in Section 7.2.1, we can even eliminate the `self.`, since inside the `String` class Ruby is smart enough to apply the `scan` to the string itself. This yields the final version of the code, as shown in Listing 8.36.

Listing 8.36: Omitting the `self.` inside the `String` class. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      scan(/[a-z]/i).join.downcase
    end
end
```

One more run of the test suite confirms that everything is still copacetic (Figure 8.9):

Figure 8.9: Still a palindrome after all our work.

Listing 8.37: GREEN

Click here to view code image

```
$ bundle exec rake test
4 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

8.5.1 Publishing the Ruby Gem

Having finished a refactored version of our `palindrome` gem, we're now ready for the final step, which is to publish the gem publicly so that it can be included into other projects (such as the live web app in Chapter 10). Luckily, RubyGems makes this amazingly easy.

First, we should make a Git commit and push up the remote repository:

Click here to view code image

```
$ git add -A
$ git commit -m "Finish working and refactored palindrome method"
$ git push
```

Next, it's a good idea to install the gem using `rake install` and then test it locally using irb:

Click here to view code image

```
$ bundle exec rake install
$ irb
>> require '<username>_palindrome'
true
>> "Madam, I'm Adam.".palindrome?
true
```

With a working gem confirmed, it's time to publish it to the world. You'll need to register for an account at RubyGems.org (unless you're already a member, in which case you should run `gem signin`), at which point you can release your gem using `rake release`:

Click here to view code image

```
$ bundle exec rake release
mhartl_palindrome 0.1.0 built to pkg/mhartl_palindrome-0.1.0.gem.
Tagged v0.1.0.
Pushed git commits and tags.
Pushed mhartl_palindrome 0.1.0 to rubygems.org
```

After waiting a few minutes for RubyGems to update its system, you should be able to uninstall the locally installed version and then reinstall it from the Web:

Click here to view code image

```
$ gem uninstall <username>_palindrome
Successfully uninstalled <username>_palindrome-0.1.0
$ gem install <username>_palindrome -v 0.1.0
Successfully installed <username>_palindrome-0.1.0
1 gem installed
$ irb
>> require '<username>_palindrome'
true
>> "RaceCar".palindrome?
true
```

That's it! Your Ruby gem is now publicly available to be incorporated into anyone's project.

For a general Ruby gem project, you can continue adding features and making new releases. All you need to do is increment the version number in `lib/<username>_palindrome/version.rb` to reflect the changes you've made. For more guidance on how to increment the versions, I suggest learning a bit about the rules of so-called *semantic versioning*, or *semver* (Box 8.2).

Box 8.2. Semver

You might have noticed in this section that we've used the version number 0.1.0 for our new gem. The leading zero indicates that our package is at an early stage, often called "beta" (or even "alpha" for very early-stage projects).

We can indicate updates by incrementing the middle number in the version, e.g., from 0.1.0 to 0.2.0, 0.3.0, etc. Bugfixes are represented by incrementing the rightmost number, as in 0.2.1, 0.2.2, etc., and a mature version (suitable for use by others, and which may not be backward-compatible with prior versions) is indicated by version 1.0.0.

After reaching version 1.0.0, further changes follow this same general pattern: 1.0.1 would represent minor changes (a "patch release"), 1.1.0 would represent new (but backward-compatible) features (a "minor release"), and 2.0.0 would represent major or backward-incompatible changes (a "major release").

These numbering conventions are known as *semantic versioning*, or *semver* for short. For more information, see semver.org.

8.5.2 Exercises

1. Restore the module in the palindrome gem, as shown in Listing 8.38. Confirm that the test suite is still GREEN.
2. Let's generalize our palindrome detector by adding the capability to detect integer palindromes like 12321. By filling in `FILL_IN` in Listing 8.39, write tests for integer non-palindromes and palindromes. (Note the call to `to_s` to convert integers to strings so that we can apply `scan`.) Get both tests to GREEN by updating the regex to match digits and including the palindrome module in `Integer` (Listing 8.40).
3. Bump the version number, commit your changes, and release a new version of your gem.

Listing 8.38: Restoring the palindrome module. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

module MhartlPalindrome
```

```
  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      scan(/[a-z]/i).join.downcase
    end
end

class String
  include MhartlPalindrome
end
```

Listing 8.39: Testing integer palindromes. RED
test/<username>_palindrome_test.rb

Click here to view code image

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end
    def test_integer_non_palindrome
    FILL_IN 12345.palindrome?
  end

  def test_integer_palindrome
    FILL_IN 12321.palindrome?
  end
end
```

Listing 8.40: Adding detection of integer palindromes. GREEN
lib/<username>_palindrome.rb

Click here to view code image

```
require "mhartl_palindrome/version"

module MhartlPalindrome
```

```ruby
  # Returns true for a palindrome, false otherwise.
  def palindrome?
    if processed_content.empty?
      false
    else
      processed_content == processed_content.reverse
    end
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      to_s.scan(/[a-zFILL_IN]/i).join.downcase
    end
end

class String
  include MhartlPalindrome
end

class Integer
  include FILL_IN
end
```

Chapter 9

Shell Scripts

In this chapter, we'll build on the foundation laid in Section 1.4 and write three *shell scripts* of increasing sophistication. Although web development is the most common application of Ruby programming nowadays, shell scripting is Ruby's native habitat, so it's a task at which Ruby understandably excels. Indeed, readers who have studied the analogous material in the context of JavaScript may be impressed at how much more elegant and polished the Ruby versions are.

In the first two programs (Section 9.1 and Section 9.2), we'll take the Ruby gem developed in Chapter 8 and put it to work detecting palindromes drawn from two different sources: a file, and the Web. In the process, we'll learn how to read and write from files with Ruby, and also how to read from a live Web URL. (This latter example has an especially personal meaning to me, as I distinctly remember the first time I wrote an automated program to read and process text from the Web, which at the time seemed truly miraculous.)

Finally, in Section 9.3, we'll write a real-life utility program adapted from one I once wrote for myself. It includes an introduction to manipulation of the Document Object Model (or *DOM*) in a context outside of a web browser.[1]

[1]The Document Object Model was introduced in *Learn Enough CSS & Layout to Be Dangerous* and is explored in more depth in*Learn Enough JavaScript to Be Dangerous*.

9.1 Reading from Files

Our first task is to read and process the contents of a file. The example is simple by design, but it demonstrates the necessary principles, and gives you the background needed to read more advanced documentation.

We'll start by using `curl` to download a file of simple phrases (note that this should be in the `ruby_tutorial` directory we used prior to Chapter 8, not the palindrome gem directory):

Click here to view code image

```
$ cd ~/repos/ruby_tutorial/
$ curl -OL https://cdn.learnenough.com/phrases.txt
```

As you can confirm by running `less phrases.txt` at the command line, this file contains a large number of phrases—some of which (surprise!) happen to be palindromes.

Our specific task is to write a palindrome detector that iterates through each line in this file and prints out any phrases that are palindromes (while ignoring others). To do this, we'll need to open the file and read its contents. We'll then use the gem developed in Chapter 8 to determine which phrases are palindromes.

Most file operations in Ruby are handled by the `File` class. (Section 9.1.1 discusses the closely related `FileUtils` module.) Let's get started with the basic `read` operation by looking at some examples in irb:

Click here to view code image

```
>> text = File.read('phrases.txt')
<lots of output>
```

This reads the contents of `phrases.txt` and puts it in the `text` variable. Because default behavior in some versions of irb is to display the return value, this might also dump the contents to the screen. This can be inconvenient for longer files, but there's a neat trick that exploits Ruby's ability to separate statements using a semicolon instead of a newline:

Click here to view code image

```
>> s = 'supercalifragilisticexpialidocious'; 1 + 2
=> 3
```

Here we see the return value of `1 + 2`, but not the long string in the prior assignment.

My personal convention with a statement whose contents I don't want to display is to use a `0` for the second statement, like this:

Click here to view code image

```
>> text = File.read('phrases.txt'); 0
=> 0
```

This arranges to do the same assignment as before, but then just prints out the value of the second statement (`0`). (Note that the default behavior for assignments has changed in more recent versions of irb, so this trick may be unnecessary on your system.)

We can confirm that the assignment worked as follows:

Click here to view code image

```
>> text.length
=> 1373
>> text.split("\n")[0]    # Split on newlines and extract the 1st phrase.
=> "A butt tuba"
```

The second command here splits the text on the newline character `\n` and selects the zeroth element, revealing the enigmatic first line of the file, "A butt tuba".

Let's take the ideas from irb and put them in a script to detect the palindromes in `phrases.txt`:

```
$ touch palindrome_file
$ chmod +x palindrome_file
```

We'll then put in the necessary shebang line (Section 1.4) and require the palindrome gem, as shown in Listing 9.1. You should use your gem if possible, but you can use `mhartl_palindrome` if you didn't publish your own.

Listing 9.1: Including the shebang line and gem.
palindrome_file

Click here to view code image

```
#!/usr/bin/env ruby
require '<username>_palindrome'

puts "hello, world!"
```

The final line in Listing 9.1 is a habit I have of always making sure a script is in a working state before writing any more code:

```
$ ./palindrome_file
hello, world!
```

The script itself is simple: We just open the file, split the contents on new-lines, and iterate through the resulting array, printing any line that's a palindrome. The result, which at this stage you should aspire to read fairly easily, appears in Listing 9.2.

Listing 9.2: Reading and processing the contents of a file.
palindrome_file

Click here to view code image

```
#!/usr/bin/env ruby
require '<username>_palindrome'

text = File.read("phrases.txt")
text.split("\n").each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end
```

Running the script at the command line confirms that there are quite a few palindromes in the file:

Click here to view code image

```
$ ./palindrome_file
.
.

.
palindrome detected: Dennis sinned.
palindrome detected: Dennis and Edna sinned.
palindrome detected: Dennis, Nell, Edna, Leon, Nedra, Anita, Rolf, Nora,
Alice, Carol, Leo, Jane, Reed, Dena, Dale, Basil, Rae, Penny, Lana, Dave,
Denny, Lena, Ida, Bernadette, Ben, Ray, Lila, Nina, Jo, Ira, Mara, Sara,
Mario, Jan, Ina, Lily, Arne, Bette, Dan, Reba, Diane, Lynn, Ed, Eva, Dana,
Lynne, Pearl, Isabel, Ada, Ned, Dee, Rena, Joel, Lora, Cecil, Aaron, Flora,
Tina, Arden, Noel, and Ellen sinned.
palindrome detected: Go hang a salami, I'm a lasagna hog.
palindrome detected: level
palindrome detected: Madam, I'm Adam.
palindrome detected: No "x" in "Nixon"
palindrome detected: No devil lived on
palindrome detected: Race fast, safe car
palindrome detected: racecar
palindrome detected: radar
palindrome detected: Was it a bar or a bat I saw?
palindrome detected: Was it a car or a cat I saw?
palindrome detected: Was it a cat I saw?
palindrome detected: Yo, banana boy!
```

Among others, we see a rather elaborate expansion on the simple palindrome “Dennis sinned” (Figure 9.1)![2]

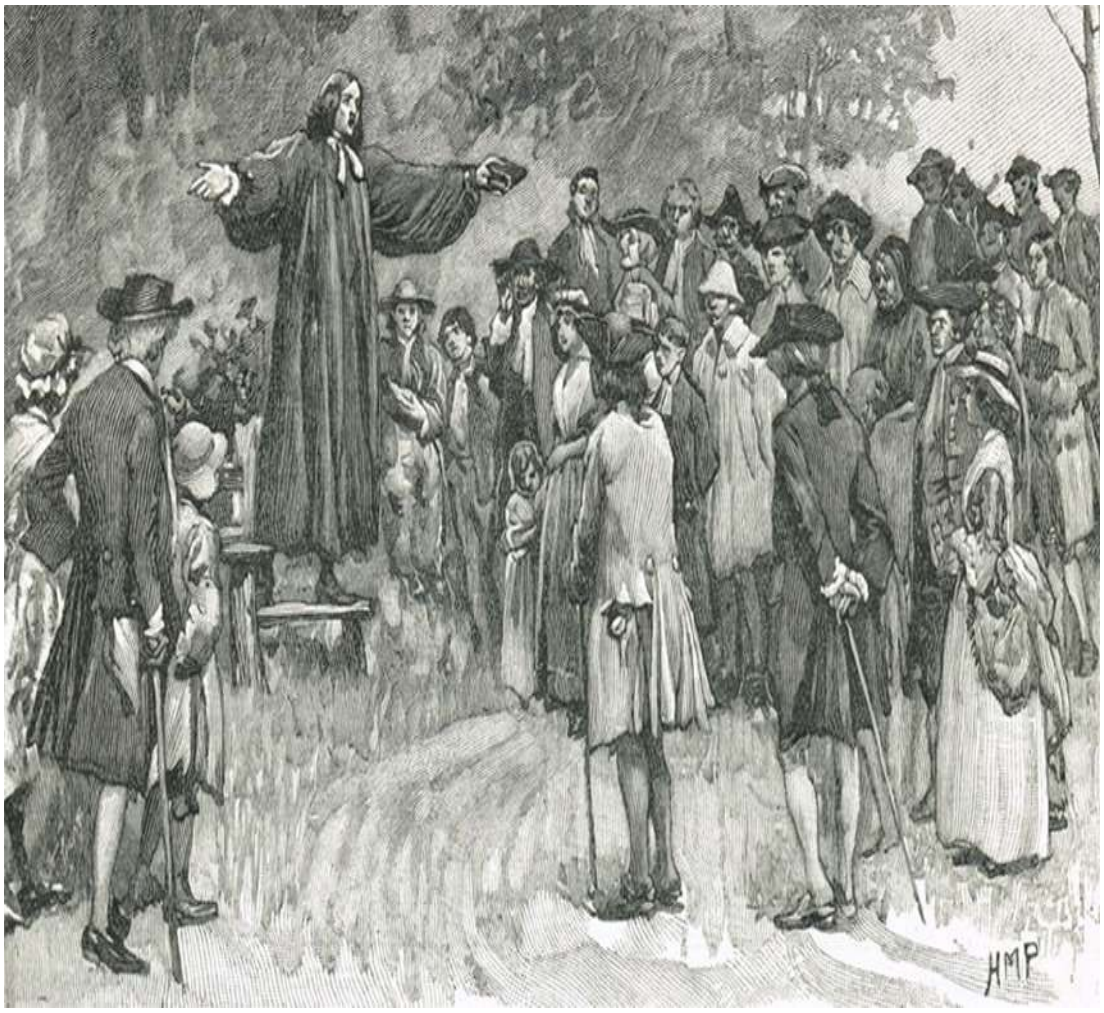


Figure 9.1: Dennis, Nell, Edna, Leon, Nedra, and many others sinned.

[2]Image courtesy of Historical Images Archive/Alamy Stock Photo.

This is a great start, but in fact `File` includes a `readlines` method that reads all the lines by default, without needing the additional `split`. Applying this to Listing 9.2 gives Listing 9.3.

Listing 9.3: Switching to `readlines`.
palindrome_file

Click here to view code image

```
#!/usr/bin/env ruby
require '<username>_palindrome'

lines = File.readlines("phrases.txt")
lines.each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
```

```
  end
end
```

You should confirm at the command line that the result is the same. *Note*: Each element in `File.readlines` actually *includes* a newline, so one might think the printed output would have an extra return between lines, but it turns out that `puts` takes this into account and ignores the first newline at the end of a string. This also means that we should `join` the array on the empty string rather than a newline, a detail we'll put to use in a moment.

Finally, let's look at how to *write* files in Ruby. It could hardly be simpler; the template looks like this:

Click here to view code image

```
File.write(filename, content_string)
```

Using the `lines` variable defined in Listing 9.3, we can use the `select` method from Section 6.2 to make an array of all the palindromes, like this:

Click here to view code image

```
palindromes = lines.select { |line| line.palindrome? }
```

Because the `palindrome?` method is called on each `line` itself, we can even use the "symbol-to-proc" notation also mentioned in Section 6.2, like this:

Click here to view code image

```
palindromes = lines.select(&:palindrome?)
```

Joining the array (Section 3.4.3) and writing the resulting string to a `palindromes_file.txt` file is then just two lines total, as seen in Listing 9.4.

Listing 9.4: Writing out the palindromes.
palindrome_file

Click here to view code image

```
#!/usr/bin/env ruby
require '<username>_palindrome'

lines = File.readlines("phrases.txt")
lines.each do |line|
```

```
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end

palindromes = lines.select(&:palindrome?)
File.write('palindromes_file.txt', palindromes.join)
```

Confirming that this works is left as an exercise (Section 9.1.1).

9.1.1 Exercises

1. You may have noticed some duplication in Listing 9.4: We first detect all palindromes, writing them out one at a time, and then find a list of all palindromes again (using `select`). Show that we can eliminate this duplication by replacing the whole file with the much more compact code shown in Listing 9.5.
2. Ruby has many utilities for replicating standard Unix-style filesystem operations like `mv`, `cp`, and `rm`, mostly concentrated in the `FileUtils` module. Using `File.exist?` and `FileUtils.rm`, write a program to remove the file `palindromes_file.txt` if it exists (you can use a file or irb). What happens if you use `FileUtils.rm` (*without* `File.exist?`) after removing the file?

Listing 9.5: Writing out palindromes the unduplicated way.
palindrome_file

Click here to view code image

```
#!/usr/bin/env ruby
require '<username>_palindrome'

palindromes = File.readlines('phrases.txt').select(&:palindrome?)
palindromes.each { |palindrome| puts "palindrome detected: #{palindrome}" }
File.write('palindromes_file.txt', palindromes.join)
```

9.2 Reading from URLs

In this section, we'll write a script whose effect is identical to the one in Section 9.1, except that it reads the `phrases.txt` file directly from its public URL. By itself, the program doesn't do anything fancy, but realize what a miracle this is: The ideas aren't specific to the URL we're hitting, which means that after this section you'll have the power to write programs to access and process practically any site on the Web. (This practice, sometimes called "web scraping", should be done with due consideration and caution.)

The main trick is to use the `OpenURI` module, which we can include like this:

```
require 'open-uri'
```

As noted in the documentation, this module includes an `open` method that can just, well, open a URI (also called a URL; the difference rarely matters). Indeed, it even has a `readlines` method to parallel the one on the `File` class in Listing 9.3, which means we can just copy that code while replacing only one line!

We can create our script as in Section 9.1:

```
$ touch palindrome_url
$ chmod +x palindrome_url
```

The only diffs between the new script and the one in Listing 9.3 are the extra `require` statement and changing `File.readlines(filename)` to `open-(url).readlines`. The resulting `palindrome_url` script appears in Listing 9.6.

Listing 9.6: Reading a URL.
palindrome_url

Click here to view code image

```
#!/usr/bin/env ruby
require '<username>_palindrome'
require 'open-uri'

lines = URI.open('https://cdn.learnenough.com/phrases.txt').readlines
lines.each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end
```

At this point, we're ready to try the script out at the command line:

Click here to view code image

```
$ ./palindrome_url
.
.
.

palindrome detected: Dennis sinned.
palindrome detected: Dennis and Edna sinned.
palindrome detected: Dennis, Nell, Edna, Leon, Nedra, Anita, Rolf, Nora,
Alice, Carol, Leo, Jane, Reed, Dena, Dale, Basil, Rae, Penny, Lana, Dave,
Denny, Lena, Ida, Bernadette, Ben, Ray, Lila, Nina, Jo, Ira, Mara, Sara,
Mario, Jan, Ina, Lily, Arne, Bette, Dan, Reba, Diane, Lynn, Ed, Eva, Dana,
Lynne, Pearl, Isabel, Ada, Ned, Dee, Rena, Joel, Lora, Cecil, Aaron, Flora,
Tina, Arden, Noel, and Ellen sinned.
palindrome detected: Go hang a salami, I'm a lasagna hog.
palindrome detected: level
palindrome detected: Madam, I'm Adam.
```

```
palindrome detected: No "x" in "Nixon"
palindrome detected: No devil lived on
palindrome detected: Race fast, safe car
palindrome detected: racecar
palindrome detected: radar
palindrome detected: Was it a bar or a bat I saw?
palindrome detected: Was it a car or a cat I saw?
palindrome detected: Was it a cat I saw?
palindrome detected: Yo, banana boy!
```

Amazing! The result is exactly as we saw in Section 9.1, but this time, we got the data right off the live Web.

By the way, if you actually visit the URL cdn.learnenough.com/phrases.txt, you'll find that in fact it *forwards* (using a 301 redirect) to a page on Amazon's Simple Storage Service (S3), as seen in Figure 9.2.

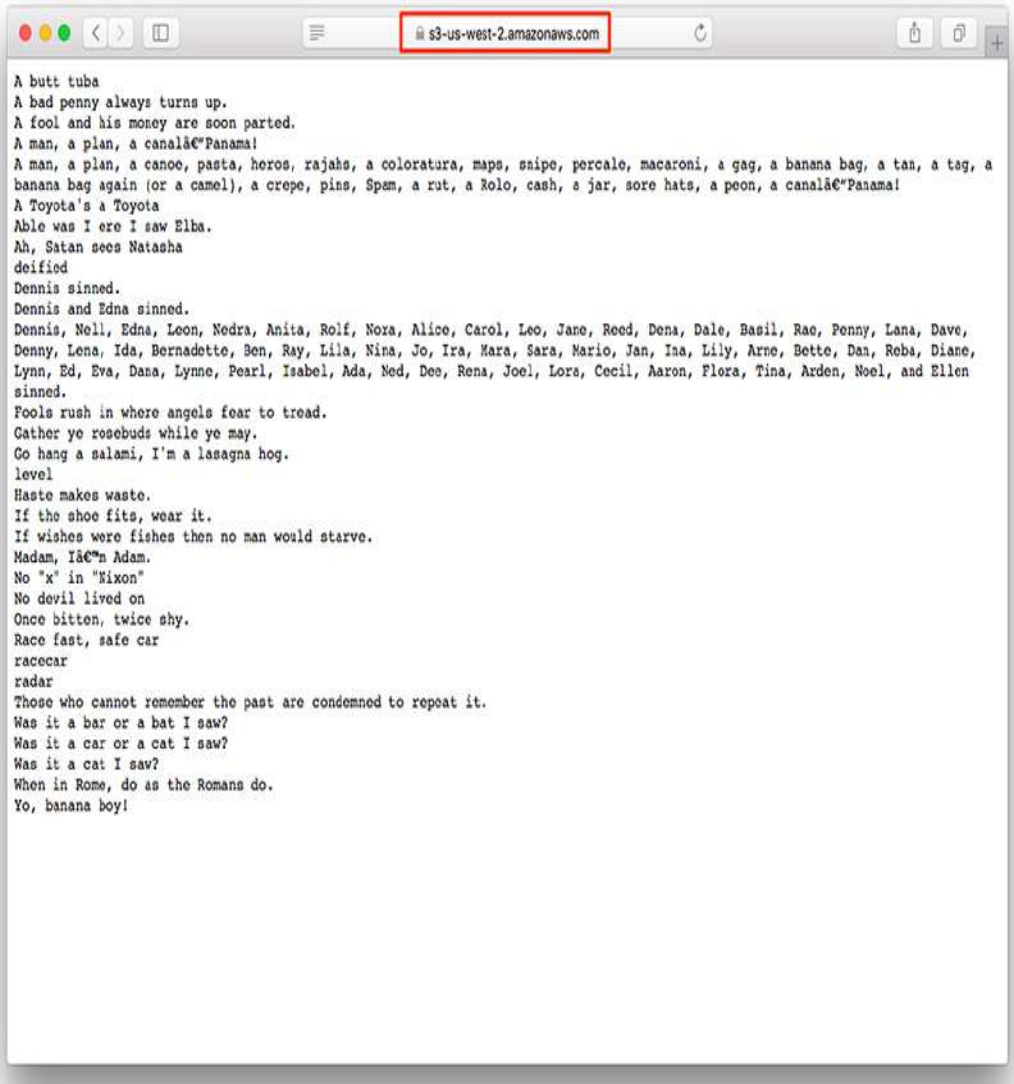


Figure 9.2: Visiting the phrase URL.

Luckily, the `open` function we used in Listing 9.6 automatically follows such redirects, so the script worked as written, but this behavior is not universal among URL libraries. Depending on the exact library you use, you might have to manually configure the web requester to follow redirects.

9.2.1 Exercises

1. In analogy with Listing 9.4, add code to Listing 9.6 that writes out a file called `palindromes_url.txt`. Confirm using the `diff` utility that the output is identical to the `palindromes_file.txt` file from Section 9.1.
2. Modify Listing 9.6 to use the more compact programming style seen in Listing 9.5 (including the step to write out the file).

9.3 DOM Manipulation at the Command Line

In this final section, we're going to put the URL-reading tricks we learned in Section 9.2 to good use by writing a version of an actual utility script I once wrote for myself. To begin, I'll explain the context in which the script arose, and the problem it solves.

In recent years, there has been an explosion in the resources available for learning foreign languages, including things like Duolingo, Google Translate, and native OS support for multilingual text-to-speech (TTS). A few years ago, I decided to take advantage of this opportunity to brush up on my high-school/college Spanish.

One of the resources I found myself turning to was Wikipedia, with its huge number of articles in languages other than English. In particular, I discovered how useful it was to copy text from Spanish-language Wikipedia (Figure 9.3) and drop it into Google Translate (Figure 9.4). At that point, I could use the text-to-speech from either Google Translate (the red square in Figure 9.4) or macOS to hear the words spoken in Spanish, while following along with either the native language or the translation. Es muy útil.

Figure 9.3: Un artículo sobre Ruby.

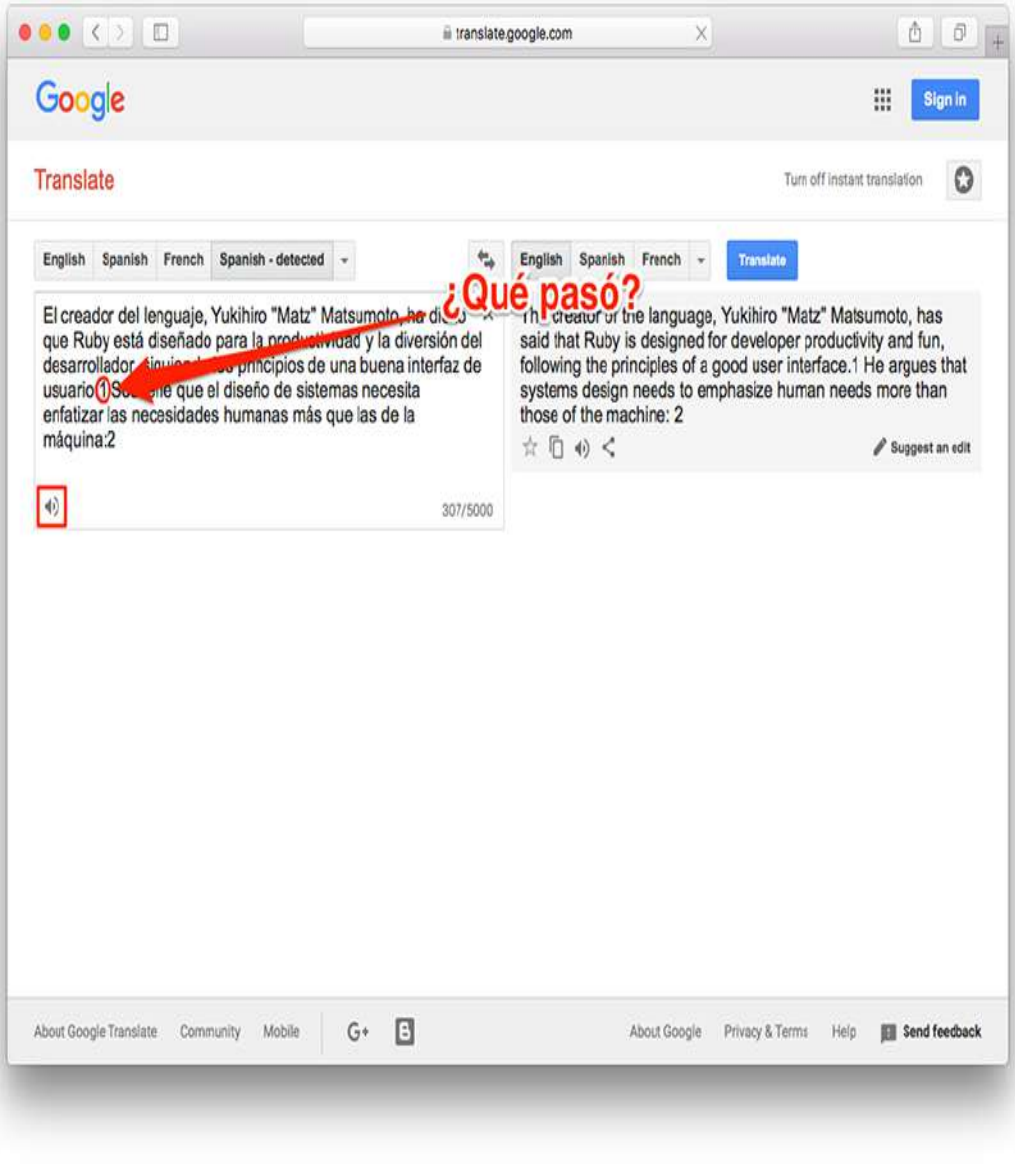


Figure 9.4: An article about Ruby dropped into Google Translate.

After a while, I noticed two consistent sources of friction:

- Copying a large number of paragraphs by hand was cumbersome.
- Hand-copying text often selected things that I didn't want, particularly *reference numbers*, which the TTS system duly pronounced, resulting in random numbers in the middle of sentences (e.g., "siguiendo los principios de una buena interfaz de usuario.1" = "Following the principles of a good user interface.1[uno]" ¿Qué pasó?).

Friction like this has inspired many a utility script, and thus was born `wikp` ("Wikipedia paragraphs"), a program to download a Wikipedia article's HTML source, extract its paragraphs, and eliminate its reference numbers, dumping all the results to the screen.

The original `wikp` program was written in Ruby; what appears here is a slightly simplified version. Let's think about how it will work.

We already know from Listing 9.6 how to download the source of a URL. The remaining tasks are then to:

- Take an arbitrary URL argument at the command line
- Manipulate the downloaded HTML using the DOM (Figure 9.5)
- Remove the references
- Collect and print the paragraphs

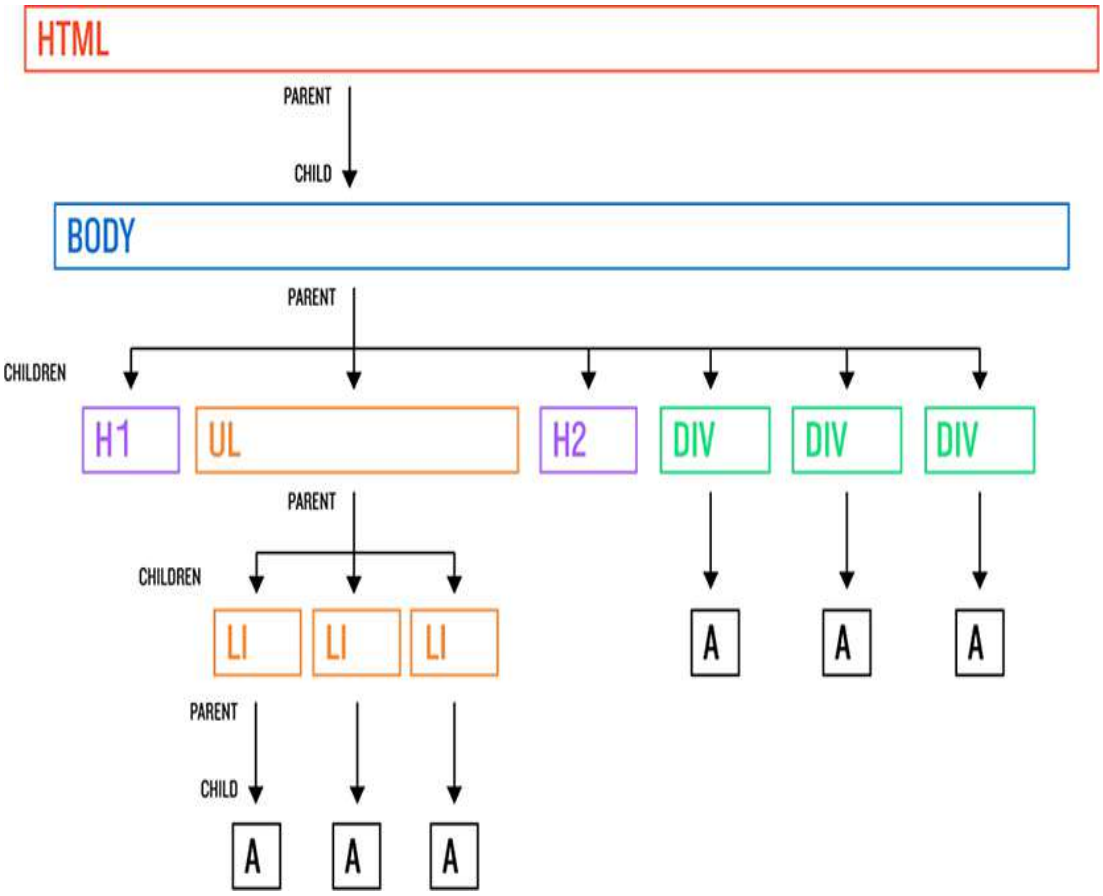


Figure 9.5: The Document Object Model (DOM).

Let's get started by creating the initial script:

```
$ touch wikp
$ chmod +x wikp
```

Now we're ready to get going on the main program. For each task above, I'll include the kind of Google search you might use to figure out how to do it.

The first step is to install a Ruby gem called `nokogiri` (named for a kind of Japanese saw) that can manipulate the DOM (ruby dom manipulation):

Click here to view code image

```
$ gem install nokogiri -v 1.8.5
Building native extensions. This could take a while...
```

Our principal task is sometimes known as "HTML parsing", and Nokogiri comes equipped with a powerful HTML parser. The official Nokogiri website has a bunch of useful tutorials; for our purposes, the most important method looks like Listing 9.7.

Listing 9.7: Parsing some HTML.

Click here to view code image

```
>> require 'nokogiri'
>> html = '<p>lorem<sup class="reference">1</sup></p><p>ipsum</p>'
>> doc = Nokogiri::HTML(html)
=> #<Nokogiri::HTML::Document:0x3fd87e023b18...
```

The resulting `doc` variable is a Nokogiri document, in this case with two paragraphs, one of which contains a `sup` (superscript) tag with CSS class `reference`. Nokogiri documents can be manipulated in any number of ways.

My favorite Nokogiri method for selecting elements is `css`, which lets us pull out HTML tags (nokogiri select html tag) and CSS ids/classes (nokogiri select css id class) using an intuitive syntax. For example:

Click here to view code image

```
>> doc.css('p')
=> [#<Nokogiri::XML::Element:0x3fd87e022664 name="p"...
>> doc.css('p').length
=> 2
>> doc.css('p')[0].content
=> "lorem1"
```

We see from the final line that we can get the content of a particular result using the `content` method, which in this case includes the reference number `1`. Meanwhile, we can grab the elements (in this case, only one) with a `reference` class using the same dot notation as in CSS:[3]

[3] By printing out the result of the `css` method, you can confirm that it's not actually an `Array`, but rather a custom Nokogiri class. And yet, we can call `each` on it just as we would an ordinary array. Ruby's ability to manipulate objects in this way, based on how they act rather than on their formal class type, is called *duck typing*, based on the aphorism that "If it looks like a duck, and it quacks like a duck, it's probably a duck."

Click here to view code image

```
>> doc.css('.reference')
=> [#<Nokogiri::XML::Element:0x3fd87e04d60c name="sup"...
>> doc.css('.reference').length
=> 1
```

Perhaps you can see where we're going with this. We're now in a position to parse an HTML document and select all the paragraphs and all the references (assuming, of course, they have class `reference`). All we need now is a way to *remove* the references. As it happens, this is not hard at all (nokogiri remove element), as seen in Listing 9.8.

Listing 9.8: Removing DOM elements.

Click here to view code image

```
>> doc.css('.reference').each { |reference| reference.remove }
```

Then, we can collect all the paragraph content using `map` from Section 6.1, as shown in Listing 9.9.

Listing 9.9: Mapping paragraph content.

Click here to view code image

```
>> doc.css('p').map { |paragraph| paragraph.content }
=> ["lorem", "ipsum"]
```

In the full script, we'll join this on newlines to get a nicely formatted output of the paragraph content.

First, we'll take in the URL as a command-line argument (ruby script command line argument), as seen in Listing 9.10. Note that we've included a `puts`

line as a temporary way to track our progress.

Listing 9.10: Accept a command-line argument.
wikp

Click here to view code image

```
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]

puts url
```

We can confirm that Listing 9.10 works as advertised:

Click here to view code image

```
$ ./wikp https://es.wikipedia.org/wiki/Ruby
https://es.wikipedia.org/wiki/Ruby
```

Next, we need to open the URL and read its contents. We saw both `File.-read` and `File.readlines` in Section 9.1, and `open(url).readlines` in Section 9.2, so perhaps it won't surprise you that you can read the full contents of a URL with `URL.open(url).read` (ruby_open_url_read). Combining the result with the Nokogiri parsing in Listing 9.7 gives Listing 9.11.

Listing 9.11: Parsing the live URL with Nokogiri.
wikp

Click here to view code image

```
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
```

Now all we need to do is apply the reference removal and paragraph collection code from Listing 9.8 and Listing 9.9. As hinted above, Wikipedia identifies its

references with the `.reference` class, which we can confirm using a web inspector (Figure 9.6). This suggests the reference removal code shown in Listing 9.12.

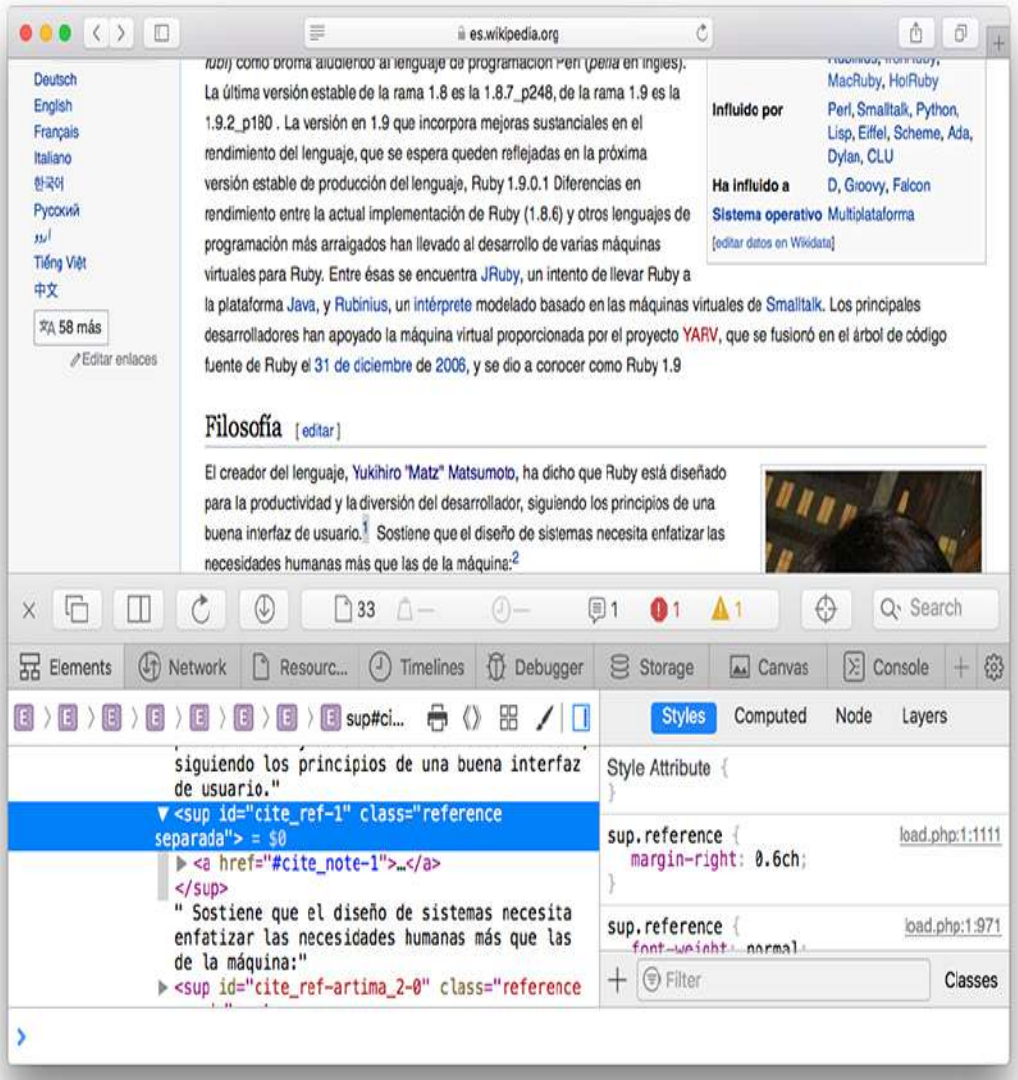


Figure 9.6: Viewing a reference in the web inspector.

Listing 9.12: Removing the references.
wikp

Click here to view code image

```
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'
```

```
# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
doc.css('.reference').each { |reference| reference.remove }
```

Now all that's left is to extract the paragraph content and print it out (Listing 9.13).

Listing 9.13: Printing the content.
wikp

Click here to view code image

```
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
doc.css('.reference').each { |reference| reference.remove }
content_array = doc.css('p').map { |paragraph| paragraph.content }
puts content_array.join("\n")
```

Let's see how things went:

Click here to view code image

```
$ ./wikp https://es.wikipedia.org/wiki/Ruby
Ruby es un lenguaje de programación interpretado, reflexivo y orientado a
objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien
comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en 1995.
.
.
.
A partir de la versión 1.9.3 se opta por una licencia dual bajo las licencias
BSD de dos cláusulas y Licencia pública Ruby.
```

Success! By scrolling up in our terminal, we can now select all the text and drop it into Google Translate or a text editor of our choice. On macOS, we can do even better by piping the results to `pbcopy`, which automatically copies the results to the macOS **p**aste**b**oard (also called the "clipboard"):

Click here to view code image

```
$ ./wikp https://es.wikipedia.org/wiki/Ruby | pbcopy
```

At this point, pasting into Google Translate (or anywhere else) will paste the full text.

As a final bit of polish, I can't resist converting both the `each` and `map` lines to use the "symbol-to-proc" notation (which also makes it convenient to eliminate the `content_array` variable), yielding the final `wikp` script shown in Listing 9.14.

Listing 9.14: The final Wikipedia paragraph script.
wikp

Click here to view code image

```
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
doc.css('.reference').each(&:remove)
puts doc.css('p').map(&:content).join("\n")
```

Consider how remarkable an accomplishment this is. The script in Listing 9.14 is a little tricky—and to get such a thing working completely on your own might involve more than a few `puts` statements as you go along—but it's really only four lines: not exactly rocket science. And yet, it's genuinely useful, something that (if you're active in foreign-language learning) you might well use all the time. Moreover, the basic skills involved—including not just the programming, but also the technical sophistication (<cough>Googling</cough>) —unlock a huge number of potential applications.

9.3.1 Exercises

1. By moving the file or changing your system's configuration, add the `wikp` script to your environment's PATH. (You may find the steps in *Learn Enough Text Editor to Be Dangerous* helpful.) Confirm that you can run `wikp` without prepending `./` to the command name. *Note*: If you have a conflicting `wikp` program from following *Learn Enough JavaScript to Be Dangerous*, I suggest replacing it—thus demonstrating the principle that the file's name is the user interface, and the implementation can change language without affecting users.
2. What happens if you run `wikp` with no argument? Add code to your script to detect the absence of a command-line argument and output an appropriate usage statement. *Hint*: After printing out the usage statement, you will have

to *exit*, which you can learn how to do with the search ruby how to exit script. *Extra credit*: Switch to using `Array#shift` to extract the command-line argument. This is a common pattern in Ruby scripts, allowing multiple arguments to be processed in turn.

3. The "pipe to `pbcopy`" trick mentioned in the text works only on macOS, but any Unix-compatible system can redirect the output to a file. What's the command to redirect the output of `wikp` to a file called `article.-txt`? (You could then open this file, select all, and copy the contents, which has the same basic result as piping to `pbcopy`.)

Chapter 10

A Live Web Application

In this final chapter, we reach the culmination of this Ruby tutorial: a dynamic web application. Our app will put the custom Ruby gem developed in Chapter 8 to good use through the development of a web-based *palindrome detector*. Along the way, we'll learn how to create dynamic content using *embedded Ruby* (ERB).

Detecting palindromes from the Web requires using a back-end web application to handle *form submission*, and our tool of choice is *Sinatra*, the micro-framework we met in Section 1.5 and applied further in Section 5.2. Although simple, Sinatra is not a toy—it's a production-ready web framework used by companies like Stripe, Apple, and Disney.[1]

[1] As of a few years ago, when I talked with a friend working at Walt Disney Imagineering, disney.com itself was a Sinatra app, while most of the Disney sub-properties were built using Ruby on Rails.

Our palindrome app will also feature two other pages—Home and About—which will give us an opportunity to learn how to use a Ruby-based site layout. As part of this, we'll apply and extend the work in Chapter 8 to write automated tests for our app.

Finally, as in Section 1.5, our final step will be to deploy our palindrome app to the live Web. We'll end with pointers to further resources for Ruby, Sinatra, and other topics like JavaScript and Ruby on Rails.

10.1 Setup

Our first step is to set up our app as a proof-of-concept and deploy it to production. We'll start by making a directory for it:

Click here to view code image

```
$ cd ~/repos                # cd ~/environments/repos on the cloud IDE
$ mkdir palindrome_app
$ cd palindrome_app/
```

The app itself will live in the file `app.rb`, and we'll also need a `Gemfile`:

```
$ touch app.rb Gemfile
```

In addition to the `sinatra` and `puma` gems, we'll also include `rerun`, which will let us see changes to the app without quitting and restarting the local server, as shown in Listing 10.1.

Listing 10.1: Defining the gems for our app.
Gemfile

Click here to view code image

```
source 'https://rubygems.org'

ruby '3.1.1'   # Change this line if you're using a different Ruby v
ersion.

gem  'sinatra', '2.2.0'
gem  'puma',    '5.6.4'
gem  'rerun',   '0.13.1'
```

We can then install the gems using `bundle install`:

```
$ bundle _2.3.10_ install
```

To get started with the app itself, let's write "hello, world!", as shown in Listing 10.2.

Listing 10.2: Getting to "hello"
app.rb

```
require 'sinatra'

get  '/' do
  'hello, world!'
end
```

To run the app, I recommend opening a new terminal tab and then using the `rerun` command, as shown in Listing 10.3.

Listing 10.3: Using `rerun` to run a Sinatra app.

Click here to view code image

```
$ bundle exec rerun app.rb

12:10:10 [rerun] Palindrome_app launched
12:10:10 [rerun] Rerun (79556) running Palindrome_app (79575)
== Sinatra has taken the stage on 4567 for development
Listening on localhost:4567, CTRL+C to stop
```

As with running a Sinatra app using `ruby` (e.g., Listing 1.9), Listing 10.3 runs the app on a local port, where the exact number can be read in the log output. The exact results may be system-dependent, but on my system the log says something like “Sinatra has taken the stage on 4567” (the highlighted line in Listing 10.3), meaning that the site can be accessed at localhost:4567. The result appears in Figure 10.1.

Figure 10.1: Our initial app running with `rerun`.

Now let's see what happens if we make a change to our app, as shown in Listing 10.4.

Listing 10.4: Time to say "goodbye".
app.rb

```
require 'sinatra'

get  '/' do
  'goodbye, world!'
end
```

In Section 5.2, we had to quit and restart the server in order to see changes to the app, but thanks to the `rerun` in Listing 10.3, the app is updated automatically (though we still do have to refresh the browser by hand). The result should look something like Figure 10.2.

Figure 10.2: An auto-updated Sinatra app.

Finally, following our practice to deploy early and often, we'll put our project under version control with Git in preparation for deploying to Heroku:

Click here to view code image

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

As in Section 1.5.1, we need to add a configuration file for Heroku's sake:

```
$ touch config.ru
```

The contents are shown in Listing 10.5 (which apart from the app name is identical to Listing 1.12).

Listing 10.5: The configuration file for production deployment.
palindrome_app/config.ru

```
require './app'
run Sinatra::Application
```

At this point, we're ready to deploy (the first line may or may not be necessary, but it does no harm to include it):

Click here to view code image

```
$ bundle _2.3.10_ lock --add-platform x86_64-linux
$ git add -A
$ git commit -m  "Add a config file"
$ heroku create
$ git push heroku main
```

The result is a working app in production, as seen in Figure 10.3.

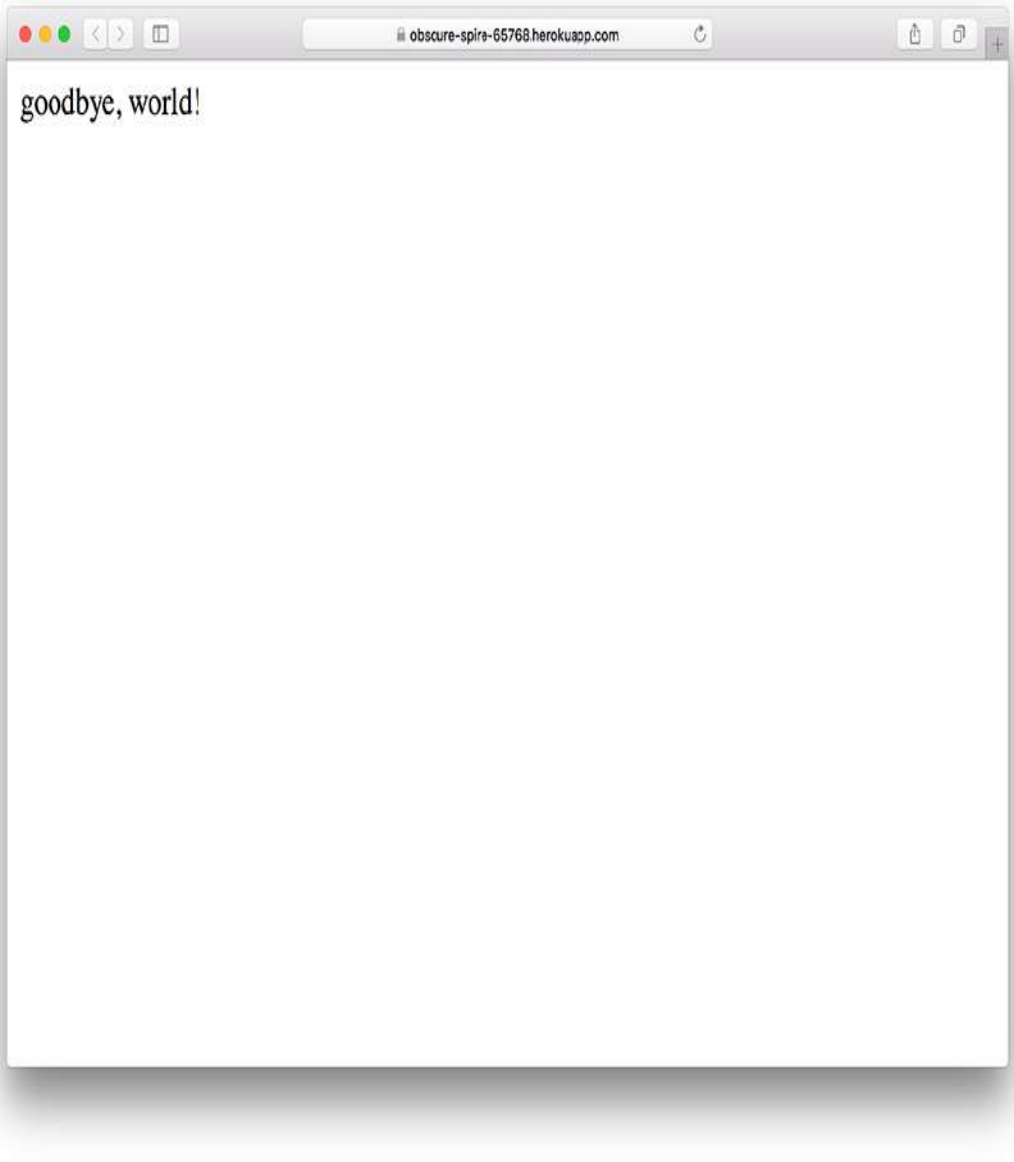


Figure 10.3: Our initial app in production.

10.1.1 Exercises

1. Change the app back to read “hello, world!”, and deploy the update to production. *Note*: The second time you deploy, you can leave off `main`, and just type `git push heroku`.

10.2 Site Pages

We'll start by making three pages for our site: Home, About, and Palindrome Detector. In contrast to our previous Sinatra apps, which have operated by simply returning strings in response to `GET` requests, for our full app we'll use a more powerful technique known as *views*, which consist of code that gets converted to HTML and sent to the browser.

Initially, these views will actually just be static HTML, but we'll see starting in Section 10.4 how to use them to generate HTML dynamically. The key to Sinatra views is the `erb` function, which stands for "embedded Ruby" (itself often abbreviated as ERB). The argument to *erb* is a Ruby symbol (Section 4.4.1) with the name of the view. For example to render the index page on the root url /, we can write

```
get '/' do
  erb :index
end
```

This code causes Sinatra to look for a file called `index.erb` in the `views` directory. (The `.erb` extension indicates that the file is to be processed with the `erb` function.)

Because the code for all three views is basically the same, we'll add them all at the same time, as shown in Listing 10.6. The main difference is, in place of the root URL /, the other two pages define named URLs: /about and /palindrome, respectively.

Listing 10.6: Rendering three views.
app.rb

```
require 'sinatra'

get '/' do
  erb :index
end

get '/about' do
  erb :about
end

get '/palindrome' do
  erb :palindrome
end
```

The file in Listing 10.6 is in effect a *controller*, which coordinates between different parts of the application, defines the URLs (or *routes*) supported by the app, responds to

requests, etc. Together, the views and controllers are two-thirds of the *Model-View-Controller* architecture for developing web applications, also known as *MVC*. (We won't get to the Model part of MVC in this tutorial, but the *Ruby on Rails Tutorial* discusses all three parts of MVC in depth.)

To get the three view renderings in Listing 10.6 to work, we have to create the three view files themselves:

Click here to view code image

```
$ mkdir views
$ cd views
$ touch index.erb about.erb palindrome.erb
$ cd ..
```

We then fill the three ERB files with HTML; this is straightforward but tedious, so I suggest you copy and paste from Listing 10.7, Listing 10.8, and Listing 10.9. (The indentation of the material inside the `body` tags is at the wrong depth, but we'll see in Section 10.3 why this is.)

Listing 10.7: The initial Home (index) view.
views/index.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">

<h1>Sample Sinatra App</h1>

<p>
```

```
  This is the sample Sinatra app for
          <a      href="https://www.learnenough.com/ruby-tutorial">
<em>Learn Enough Ruby
                          to            Be            Dangerous</em>
</a>. Learn more on the <a href="/about">About</a> page.
</p>

<p>
  Click the <a href="https://en.wikipedia.org/wiki/Sator_Square">Sat
or
  Square</a> below to run the custom <a href="/palindrome">Palindrom
e
  Detector</a>.
</p>

<a class="sator-square" href="/palindrome">
  <img src="/images/sator_square.jpg" alt="Sator Square">
</a>
      </div>
    </div>
  </body>
</html>
```

Listing 10.8: The initial About view.
views/about.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">

<h1>About</h1>

<p>
  This site is the final application in
          <a      href="https://www.learnenough.com/ruby-tutorial">
```

```
<em>Learn Enough Ruby
  to Be Dangerous</em></a>
  by <a href="https://www.michaelhartl.com/">Michael Hartl</a>,
  a tutorial introduction to the
                              <a              href="https://www.ruby-
lang.org/en/">Ruby programming language</a> that
  is part of
  <a href="https://www.learnenough.com/">LearnEnough.com</a>.
</p>

<p>
  <em>Learn Enough Ruby to Be Dangerous</em> is a natural prerequisi
te to
  the <a href="https://www.railstutorial.org/"><em>Ruby on Rails
                                                        Tutorial</em>
</a>, a book and video series that is one of the leading
  introductions to web development. <em>Learn Enough Ruby</em> is al
so an
  excellent choice <em>after</em> the <em>Rails Tutorial</em> for th
ose who
  prefer to start with the latter first.
</p>

      </div>
    </div>
  </body>
</html>
```

Listing 10.9: The initial Palindrome Detector view.
views/palindrome.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">
```

```
<h1>Palindrome Detector</h1>

<p>This will be the palindrome detector.</p>

      </div>
    </div>
  </body>
</html>
```

Visiting localhost:4567 or the equivalent causes Sinatra to serve up the default (index) page, as shown in Figure 10.4. To get to the About page, we can type localhost:4567/about into the browser address bar, as seen in Figure 10.5.

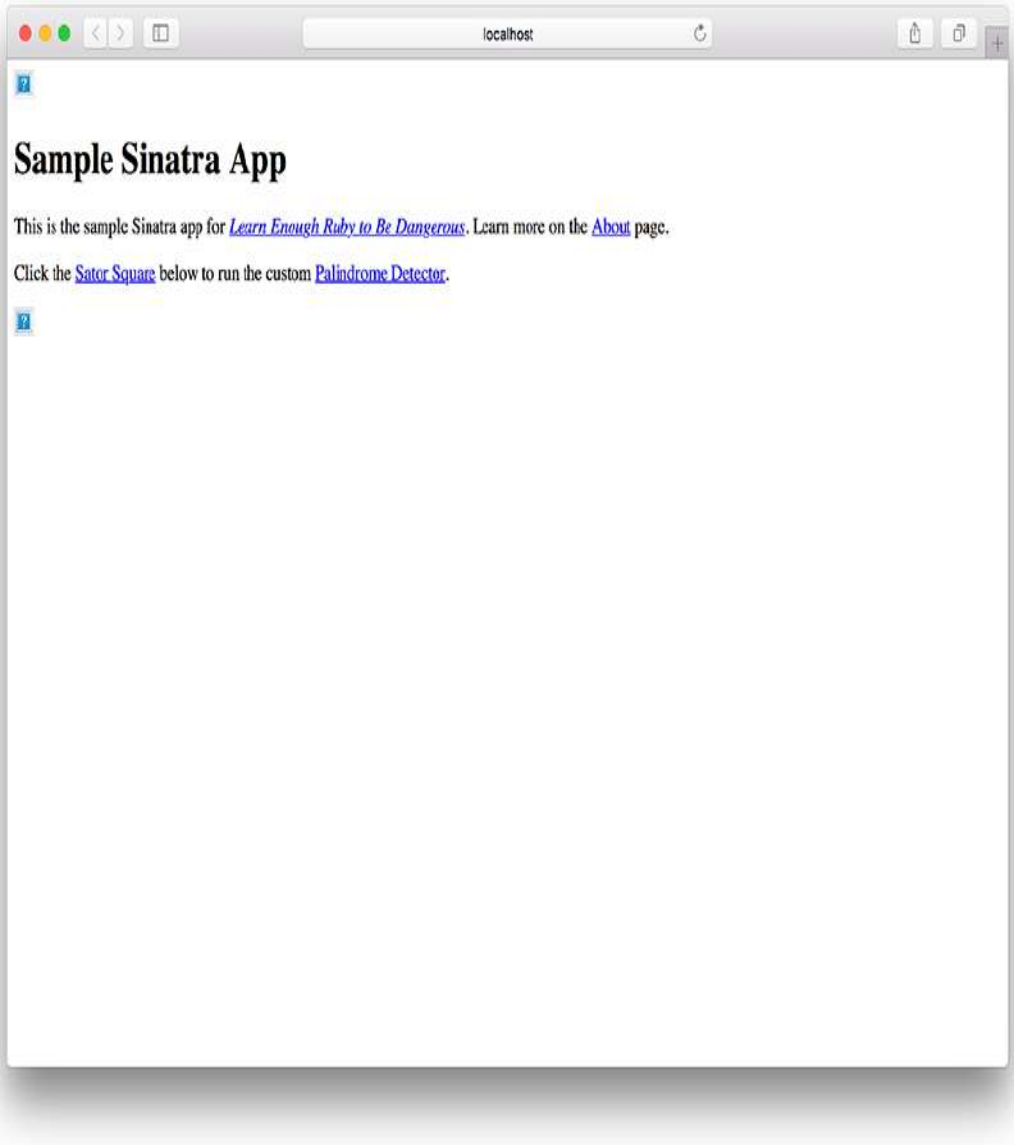


Figure 10.4: The initial Home page.

Figure 10.5: The initial About page.

Figure 10.4 and Figure 10.5 show that the pages are basically working, but Listing 10.7 and subsequent listings include both images and a CSS file, which aren't currently present on the local system. We can change this situation by downloading the needed files from the Learn Enough CDN and putting them in the `public` directory, which is where Sinatra looks for them by default.

The way to do this is to use `curl` to fetch a *tarball*, which is similar to a ZIP file and is common on Unix-compatible systems:

Click here to view code image

```
$ curl -OL https://cdn.learnenough.com/le_ruby_palindrome_public.tar.gz
```

This kind of file is created by `tar`, or "tape archive", whose name is an old-school throwback to the time when external tapes were routinely used for large backups. Meanwhile, the `gz` extension refers to the important *gzip* method for compressing files.

The way to unzip the file is to use `tar zxvf`, which stands for "**t**ape **ar**chive g**z**ip e**x**tract **v**erbose **f**ile":[2]

[2] I created this tarball using the command `tar zcf <filename>.tar.gz`, where `c` stands for `c` reate.

Click here to view code image

```
$ tar zxvf le_ruby_palindrome_public.tar.gz
x public/
x public/images/
x public/stylesheets/
x public/stylesheets/main.css
x public/images/sator_square.jpg
x public/images/logo_b.png
$ rm -f le_ruby_palindrome_public.tar.gz
```

With experience, you may prefer to omit the `v` flag, but initially I suggest using verbose output so that you can see what's going on during the extraction process. By the way, note that `tar` flags are just letters by themselves, with no preceding hyphens as in most other Unix commands. On many systems, you can in fact use hyphens, as in `tar -z -x -v -f <filename>`, but for reasons unknown to me the usual convention with `tar` is to omit them.

As seen from the verbose output above, unzipping the file has created a `public` directory:

```
$ ls public/
images      stylesheets
```

Refreshing the About page confirms that the logo image and CSS are now working (Figure 10.6). The improvement on the Home page is even more dramatic, as seen in Figure 10.7.

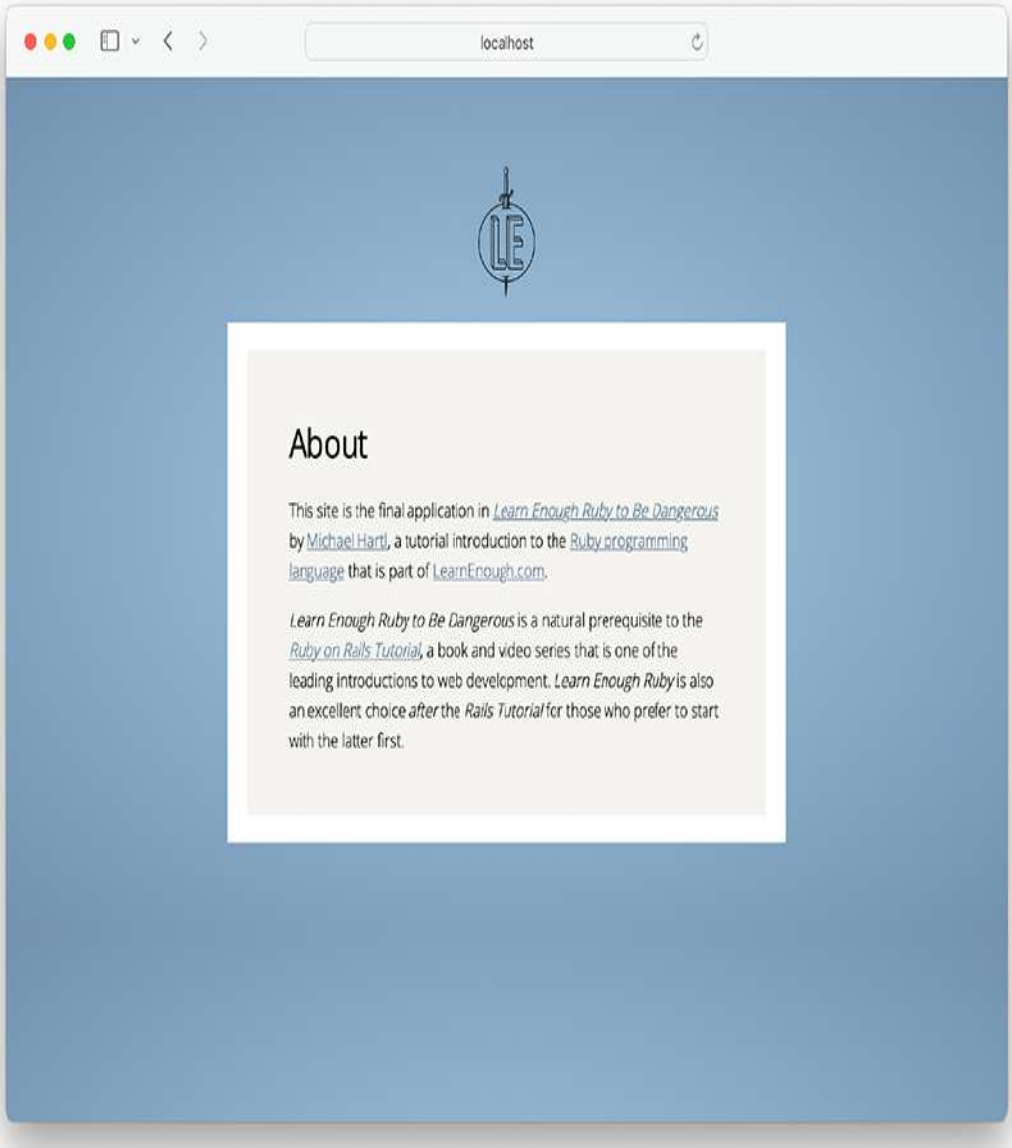


Figure 10.6: A nicer-looking About page.

Figure 10.7: A much-improved Home page.

10.2.1 Exercises

1. Visit the /palindrome URL and confirm that the CSS and images are working.
2. Make a commit and deploy the changes.

10.3 Layouts

At this point, our app is looking pretty good, but there are two significant blemishes: The HTML code for the three pages is highly repetitive, and navigating by hand from page to page is rather cumbersome. We'll fix the first blemish in this section, and the second in Section 10.4. (And of course our app doesn't yet detect palindromes, which is the subject of Section 10.5.)

> If you followed *Learn Enough CSS & Layout to Be Dangerous*, you'll know that the *Layout* in the title referred to page layout generally—using Cascading Style Sheets to move elements around on the page, align them properly, etc.— but we also saw that doing this properly requires defining *layout templates* that capture common patterns and eliminate duplication.
>
> In the present case, each of our site's pages has the same basic structure, as shown in Listing 10.10.

Listing 10.10: The HTML structure of our site's pages.

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">

        <!-- page-specific content -->
      </div>
    </div>
  </body>
</html>
```

Everything except the page-specific content (indicated by the highlighted HTML comment) is the same on each page. In *Learn Enough CSS & Layout to Be Dangerous*, we eliminated this duplication using Jekyll templates; in this tutorial, we'll use *Sinatra layouts* instead.

Now, our site is currently working, in the sense that each page has the proper content at this stage of development. We're about to make a change that involves moving around and deleting a bunch of HTML, and we'd like to do this without breaking the site. Does that sound like something we've seen before?

It does indeed. This is exactly the kind of problem we faced in Chapter 8 when we developed and then refactored the palindrome gem. In that case, we wrote automated tests to catch any regressions, and in this case we're going to do the same. (I started making websites long before automated testing of web applications was possible, much less the norm, and believe me, automated tests are a *huge* improvement over testing web apps by hand.)

To get started, we'll add some more gems to our `Gemfile`, this time in a *group* called `:test` that indicates gems that are needed only for tests. The result appears in Listing 10.11.

Listing 10.11: Adding gems for testing.
Gemfile

Click here to view code image

```
source 'https://rubygems.org'

ruby '3.1.1'   # Change this line if you're using a different Ruby v
ersion.

gem 'sinatra', '2.2.0'
gem 'puma',    '5.6.4'
gem 'rerun',   '0.13.1'

group :test do
   gem 'minitest',              '5.15.0'
   gem 'minitest-reporters',    '1.5.0'
   gem 'rack-test',             '1.1.0'

   gem 'rake',                  '13.0.6'
   gem 'nokogiri',              '1.13.3'
end
```

We then install the gems as usual:

```
$  bundle _2.3.10_ install
```

If you encounter an error while installing `nokogiri` (which is unfortunately common), you'll have to apply your technical sophistication (Box 1.1) to resolve the issue; I

especially recommend the “Google the error message” algorithm for this case.

We’ll factor out common elements needed in all tests into a test helper file, while also creating a file for our initial site pages tests:

Click here to view code image

```
$ mkdir test
$ touch test/test_helper.rb test/site_pages_test.rb
```

The test helper requires all the necessary gems, and also includes the app itself using `require_relative`, which requires files relative to the current directory. The result appears in Listing 10.12.

Listing 10.12: The initial test helper.
test/test_helper.rb

```
ENV['RACK_ENV'] = 'test'

require_relative '../app'
require 'rack/test'
require 'nokogiri'
require 'minitest/autorun'
require 'minitest/reporters'
Minitest::Reporters.use!
```

We then need to create a `Rakefile` to tell the `rake` utility how to run the tests:

```
$ touch Rakefile
```

The contents are shown in Listing 10.13.

Listing 10.13: Configuring `rake` to run tests.
Rakefile

```
require 'rake/testtask'

Rake::TestTask.new do |t|
  t.pattern = 'test/*_test.rb'
  t.warning = false
```

```
end

task :default => [:test]
```

We'll put the result of Listing 10.13 to good use in a moment, but first we need to write some tests. We'll start with super-basic tests so that the app serves up *something*, as indicated by the HTTP response code `200 (OK)`, which we can do like this:

```
def test_index
  get '/'
  assert last_response.ok?
end
```

Here we use the `get` command in the test to issue a `GET` request to the root URL /, which automatically creates an object called `last_response` as a side effect. We can then use a minitest assertion (Section 8.2) using the boolean method `ok?`, which is also available automatically. The result is a test that makes sure the server responded properly to the request.

Applying the above discussion to the About and Palindrome Detector pages as well, and combining them with configuration code pulled right from the official Sinatra documentation on tests, we arrive at our initial test suite, shown in Listing 10.14.

Listing 10.14: Our initial test suite. GREEN
test/site_pages_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index
    get '/'
    assert last_response.ok?
  end

  def test_about
    get '/about'
```

```
    assert last_response.ok?
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
  end
end
```

With the setup in Listing 10.13, we can use `rake` to execute the test suite as follows:

Listing 10.15: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

Also note that Listing 10.13 has followed the common Ruby convention of making running the test suite the default (an indication of how important tests are in the Ruby community), so we can actually just run `rake` by itself:[3]

[3] If you set up the `bundle exec` alias suggested in Section 8.1, you can thus run the compact command `be rake` to execute the full test suite.

Listing 10.16: GREEN

Click here to view code image

```
$ bundle exec rake
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

We'll continue to use `rake test` for clarity, but it's good to know about this alternative convention.

The tests in Listing 10.14 are a fine start, but they really only check if the pages are there at all. It would be nice to have a slightly more stringent test of the HTML content, though not *too* stringent—we don't want our tests to make it hard to make changes in the future. As a middle ground, we'll check that each page in the site has an `h1` tag somewhere in the document.

In order to do this, we'll write a short function in `test_helper.rb` to return a `doc` object, akin to the `document` object in JavaScript. We actually already know pretty

much how to make it based on our work in Section 9.3, where we saw how to use Nokogiri to create a document from HTML, like this:

```
Nokogiri::HTML(html)
```

The only extra ingredient is the knowledge that Sinatra's HTTP response objects always have a `body` attribute representing the HTML body (the full page, not just the `body` tag). This means we can define `doc` as shown in Listing 10.17.

Listing 10.17: What's up, Doc?
test/test_helper.rb

Click here to view code image

```
ENV['RACK_ENV'] = 'test'

require_relative '../app'
require 'rack/test'
require 'nokogiri'
require 'minitest/autorun'
require 'minitest/reporters'
Minitest::Reporters.use!

# Returns the document.
def doc(response)

  Nokogiri::HTML(response.body)
end
```

Now we're ready to use `doc` to find the `h1` (if any) on the page. One possibility would be to use `doc.css` as in, e.g., Listing 9.12:

```
doc(last_response).css('h1')
```

In the present case, though, we only need to see if there's *any* `h1` tag, so we only need the first element:

Click here to view code image

```
doc(last_response).css('h1').first
```

This would work fine, but this is such a common case that Nokogiri has a special method for it, called `at_css`:

Click here to view code image

```
doc(last_response).at_css('h1')
```

Because this will be `nil` if there's no `h1` and non-`nil` otherwise, we can use a simple assertion, like this:

Click here to view code image

```
assert doc(last_response).at_css('h1')
```

Adding this to each page in our site yields the updated test suite shown in Listing 10.18.

Listing 10.18: Adding assertions for the presence of an `h1` tag. GREEN
test/site_pages_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end


  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
  end
```

```
  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
  end
end
```

By the way, some programmers adopt the convention of only ever having one assertion per test, whereas in Listing 10.18 we have two. In my experience, the overhead associated with setting up the right state (e.g., duplicating the calls to `get`) makes this convention inconvenient, and I've never run into any trouble from including multiple assertions in a test.

The tests in Listing 10.18 should now be GREEN as required:

Listing 10.19: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

At this point, we're ready to use a Sinatra layout to eliminate duplication. Our first step is to remove the extraneous material from our pages, leaving only the core content, as shown in Listing 10.20, Listing 10.21, and Listing 10.22. (This is why the body content wasn't fully indented before.)

Listing 10.20: The core Home (index) view.
views/index.erb

Click here to view code image

```
<h1>Sample Sinatra App</h1>


<p>
  This is the sample Sinatra app for
           <a      href="https://www.learnenough.com/ruby-tutorial">
<em>Learn Enough Ruby
                          to            Be            Dangerous</em>
</a>. Learn more on the <a href="/about">About</a> page.
</p>

<p>
  Click the <a href="https://en.wikipedia.org/wiki/Sator_Square">Sat
or
```

```
  Square</a> below to run the custom <a href="/palindrome">Palindrom
e
  Detector</a>.
</p>

<a class="sator-square" href="/palindrome">
  <img src="/images/sator_square.jpg" alt="Sator Square">
</a>
```

Listing 10.21: The core About view.
views/about.erb

Click here to view code image

```
<h1>About</h1>

<p>
  This site is the final application in
            <a      href="https://www.learnenough.com/ruby-tutorial">
<em>Learn Enough Ruby
  to Be Dangerous</em></a>
  by <a href="https://www.michaelhartl.com/">Michael Hartl</a>,
  a tutorial introduction to the
                                <a               href="https://www.ruby-
lang.org/en/">Ruby programming language</a> that
  is part of
  <a href="https://www.learnenough.com/">LearnEnough.com</a>.
</p>

<p>
  <em>Learn Enough Ruby to Be Dangerous</em> is a natural prerequisi
te to
  the <a href="https://www.railstutorial.org/"><em>Ruby on Rails
                                                        Tutorial</em>
</a>, a book and video series that is one of the leading
  introductions to web development. <em>Learn Enough Ruby</em> is al
so an
  excellent choice <em>after</em> the <em>Rails Tutorial</em> for th
ose who
  prefer to start with the latter first.
</p>
```

Listing 10.22: The core Palindrome Detector view.
views/palindrome.erb

Click here to view code image

```
<h1>Palindrome Detector</h1>

<p>This will be the palindrome detector.</p>
```

Having stripped all the layout material from our pages, we'll now create a file called `layout.erb` in the `views` directory that restores it:

```
$ touch views/layout.erb
```

Let's start by filling it with the same content as the schematic HTML structure we saw in Listing 10.10, as seen in Listing 10.23.

Listing 10.23: Using an initial (incorrect) schematic layout. RED
views/layout.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">
        <!-- page-specific content -->
      </div>
    </div>
  </body>
</html>
```

As indicated in the caption to Listing 10.23, this layout breaks our test suite:

Listing 10.24: RED

Click here to view code image

```
$ bundle exec rake test
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

The `ok?` assertions in Listing 10.18 are passing, but the `h1` assertions are failing. This is because the site is now a bare layout, as shown in Figure 10.8.

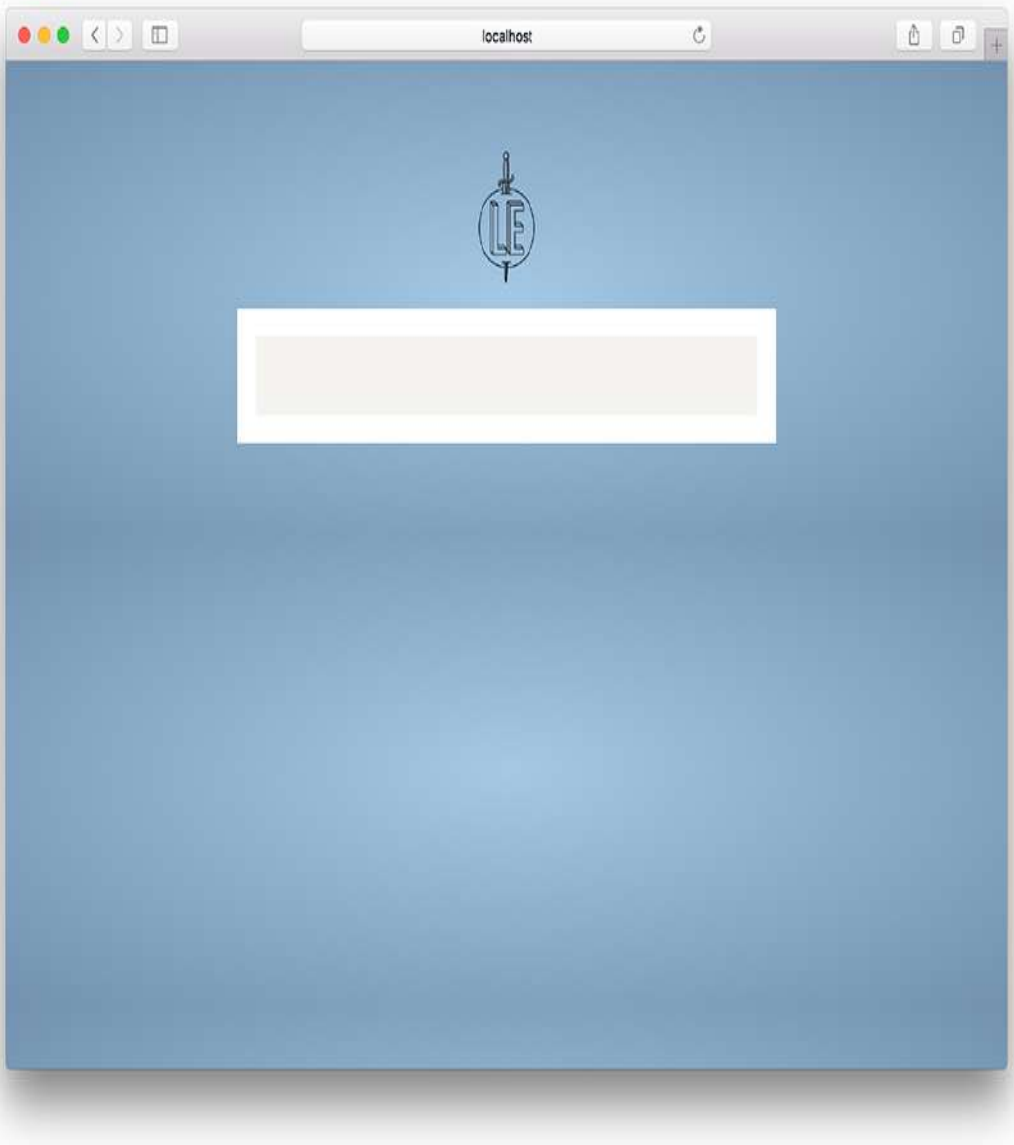


Figure 10.8: The bare layout, with no content.

To get the tests to pass and restore our site to full functionality, all we need to do is replace the HTML comment

```
<!-- page-specific content -->
```

with some special Ruby code:

```
<%= yield %>
```

This is our first example of *embedded Ruby*, which we'll learn more about in Section 10.4. The special `<%= ... %>` notation arranges to evaluate the code represented by `...` and insert it into the site.

In this case, that code is `yield`, the keyword used in Section 5.4.1 to yield content to a block, but the truth is that I don't know offhand exactly how this works or precisely why a block is involved. Indeed, I encourage you to think of `<%= yield %>` as meaning "special code that inserts the page content into a layout", and not worry about the details. (If you do much Ruby web development, you'll soon get used to it—the exact same code is used in Ruby on Rails application layouts.)

Making the replacement suggested above yields (heh) the layout shown in Listing 10.25.

Listing 10.25: Inserting content into the site layout. GREEN
views/layout.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">

      </a>
      <div class="container">
        <div class="content">
          <%= yield %>
```

```
        </div>
      </div>
    </body>
</html>
```

With that, our layout is working, the horrible repeated code has been eliminated, and our tests are GREEN:

Listing 10.26: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

A quick check in the browser confirms that things are working as expected (Figure 10.9).

Figure 10.9: Our Home page, now created using a layout.

By the way, it's worth noting that we could have used a layout file called, say, `views/page.erb`, in which case we would have had to update `app.rb` with an explicit hash option telling each page to use that file for the layout:

Click here to view code image

```
require 'sinatra'

get '/' do
  erb :index, :layout => :page
end
```

```
get '/about' do
  erb :about, :layout => :page
end

get '/palindrome' do
  erb :palindrome, :layout => :page
end
```

(Note that Ruby allows us to omit curly braces on hashes when they're the last argument to a function.) When the layout file has the special name `layout.-erb`, though, Sinatra knows to look for it by default, so we can omit it in the app file. This sort of practice foreshadows the "convention over configuration" philosophy adopted and popularized by Ruby on Rails.

10.3.1 Exercises

1. As you can confirm by running the source of any page through an HTML validator, the current pages are valid HTML, but there's a warning with a suggestion to add a `lang` (language) attribute to the `html` tag. Add the attribute `lang="en"` (for "English") to the `html` tag in Listing 10.25 and confirm using a web inspector that it appears correctly on all three pages.
2. Make a commit and deploy the changes.

10.4 Embedded Ruby

Now that we've defined a proper layout, in this section we'll use embedded Ruby (seen ever-so-briefly in Section 10.3) to add a couple of nice refinements to our site: *variable titles* and *navigation*. Variable titles are HTML `title` tag contents that vary from page to page, giving each page a nice polish of customization. Navigation, meanwhile, saves us the hassle of having to type each sub-page in by hand—certainly not the kind of user experience we're trying to create.

Our variable titles will combine a *base title*, which is the same on each page, with a piece that varies based on the page's name. In particular, for our Home, About, and Palindrome Detector pages, we want the titles to look something like this:

Click here to view code image

```
<title>Learn Enough Ruby Sample App | Home</title>
```

Click here to view code image

```
<title>Learn Enough Ruby Sample App | About</title>
```

Click here to view code image

```
<title>Learn Enough Ruby Sample App | Palindrome Detector</title>
```

Our strategy has three steps:

1. Write GREEN tests for the current page title.
2. Write RED tests for the variable titles.
3. Get to GREEN by adding the variable component of the title.

Note that Steps 2 & 3 constitute TDD—writing the tests for the variable title is easier than getting them to pass, which is one of the cases for TDD described in Box 8.1.

To get started with Step 1, we'll use the `doc` helper introduced in Listing 10.17 to extract the `text` component of the `title` tag. Recalling from Listing 10.18 that we can use `doc.at_css(<tagname>)` to select the first tag with a given tag name, we can find the `title` tag as follows:

Click here to view code image

```
doc(last_response).at_css('title')
```

We can then find the title content using the Nokogiri `content` method:

Click here to view code image

```
title_content = doc(last_response).at_css('title').content
```

This lets us add assertions for the title content to the tests in Listing 10.18, using the same `assert_equal` method we saw in Listing 8.21. The result appears in Listing 10.27.

Listing 10.27: Adding assertions for the base title content. GREEN
test/site_pages_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index

    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App", title_content
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App", title_content
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App", title_content
  end
end
```

As required, the tests are GREEN:

Listing 10.28: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 9 assertions, 0 failures, 0 errors, 0 skips
```

Now we're ready for Step 2—all we need to do is add the vertical bar | and the page-specific titles, as shown in Listing 10.29. Note that we've broken the Palindrome Detector assertion into two lines, in accordance with the 80-column rule (Box 2.2).

(Improving this by adding an instance variable to eliminate the duplication of the base title is left as an exercise (Section 10.4.1).)

Listing 10.29: Adding assertions for the variable title content. RED
test/site_pages_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | Home", title_conten
t
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | About", title_conte
nt
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | Palindrome Detector
",
                 title_content
  end
end
```

Because we haven't updated the application code, the tests are now RED:

Listing 10.30: RED

Click here to view code image

```
$ bundle exec rake test
3 tests, 9 assertions, 3 failures, 0 errors, 0 skips
```

Now for Step 3. The trick is to use an instance variable (Section 7.1) together with embedded Ruby to add the variable component of the title to the site layout, as shown in Listing 10.31.

Listing 10.31: Adding a variable component to the title. RED
views/layout.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    .
    .
    .
```

As noted briefly in Section 10.3, the `<%= ... %>` syntax evaluates the code represented by `...` and inserts it into the site at that point. In this case, that content is simply `@title`.

But what is `@title`, and where does it come from? It turns out that in Sinatra (and also in Rails), instance variables defined in controllers are automatically available in views. This means that we can define an `@title` variable for each of our pages, and it will automatically show up in the title thanks to Listing 10.31. The result appears in Listing 10.32.

Listing 10.32: Adding `@title` variables to each page. GREEN
app.rb

```
require 'sinatra'

get '/' do
  @title = 'Home'
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
```

```
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end
```

With that, our tests are passing!

Listing 10.33: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 9 assertions, 0 failures, 0 errors, 0 skips
```

Of course, it's probably a good idea to double-check in the browser, just to make sure (Figure 10.10).

Figure 10.10: Confirming the correct variable titles in the browser.

Now that we have a proper layout file, adding navigation to every page is easy. The nav code appears in Listing 10.34, with the result shown in Figure 10.11.

Figure 10.11: The site navigation.

Listing 10.34: Adding site navigation.
views/layout.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
```

```
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <header class="header">
            <nav>
              <ul class="header-nav">
                <li><a href="/">Home</a></li>
                    <li><a href="/palindrome">Is It a Palindrome?</a>
</li>
                <li><a href="/about">About</a></li>
              </ul>
            </nav>
      </header>
      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

As a final flourish, we'll factor the navigation from Listing 10.34 into a separate file, sometimes called a *partial*. This will lead to a nicely clean and tidy layout page.

Because this involves refactoring the site, we'll add a simple test (per Box 8.1) to catch any regressions. Because the navigation appears on the site layout, we could use any page to test for its presence, and for convenience we'll use the index page. As shown in Listing 10.35, all we need to do is assert the existence of a `nav` tag.

Listing 10.35: Testing the navigation. GREEN
test/site_pages_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
```

```
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | Home", title_conten
t
    assert doc(last_response).at_css('nav')
  end
  .
  .
  .
end
```

Because the nav was already added, the tests should be GREEN:

Listing 10.36: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 10 assertions, 0 failures, 0 errors, 0 skips
```

It's a good practice to watch the tests change to RED to make sure we're testing the right thing, so we'll start by cutting the navigation (Listing 10.37) and pasting it into a separate file, which we'll call `navigation.erb` (Listing 10.38):

```
$ touch views/navigation.erb
```

Listing 10.37: Cutting the navigation. RED
views/layout.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
```

```
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">

      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

Listing 10.38: Adding a navigation partial. RED
views/navigation.erb

Click here to view code image

```
<header class="header">
  <nav>
    <ul class="header-nav">
      <li><a href="/">Home</a></li>
      <li><a href="/palindrome">Is It a Palindrome?</a></li>
      <li><a href="/about">About</a></li>
    </ul>
  </nav>
</header>
```

You should confirm that the tests are now RED:

Listing 10.39: RED

Click here to view code image

```
$ bundle exec rake test
3 tests, 10 assertions, 1 failures, 0 errors, 0 skips
```

To restore the navigation, we can use embedded Ruby to evaluate the same `erb` function we've been using in `app.rb`:

```
<%= erb :navigation %>
```

This code automatically looks for a file in `views/navigation.erb`, evaluates the result, and inserts the return value where it was called.

Putting this code into the layout gives Listing 10.40.

Listing 10.40: Evaluating the nav partial in the layout. GREEN
views/layout.erb

Click here to view code image

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.c
ss">
                    <link     href="https://fonts.googleapis.com/css?
family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <%= erb :navigation %>
      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

With the code in Listing 10.40, our test suite is once again GREEN:

Listing 10.41: GREEN

Click here to view code image

```
$ bundle exec rake test
3 tests, 10 assertions, 0 failures, 0 errors, 0 skips
```

A quick click over to the About page confirms that the navigation is working (Figure 10.12). Sweet!

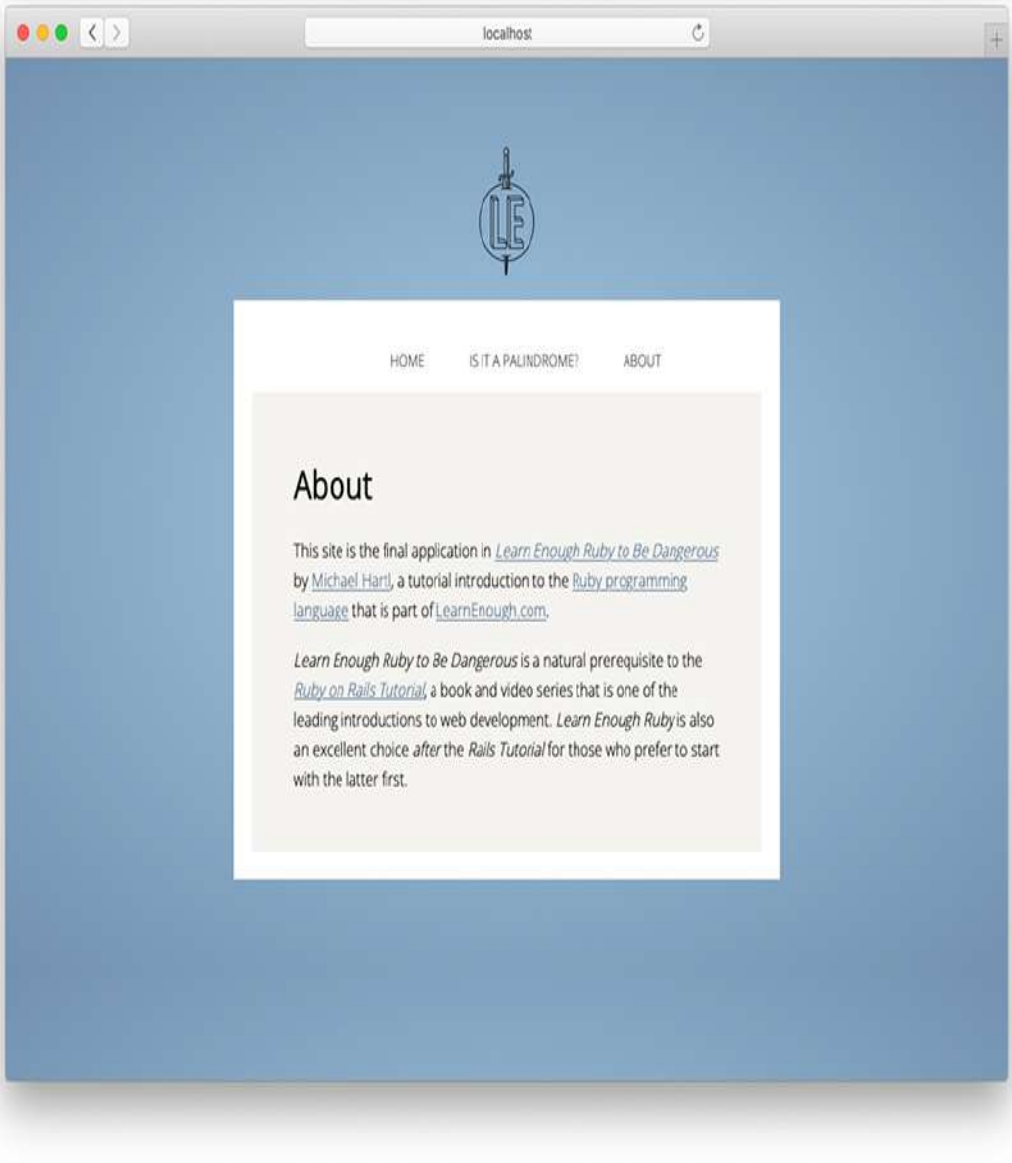


Figure 10.12: The navigation menu on the About page.

10.4.1 Exercises

1. We can eliminate some duplication in Listing 10.29 by creating a `setup` method (which is automatically run before each test) that defines an instance variable for the base title, as shown in Listing 10.42. Confirm that this code still gives a GREEN test suite.
2. Use embedded Ruby to include a call to `Time.now` (Section 4.2) somewhere on the index page. What happens when you refresh the browser?

3. Make a commit and deploy the changes. (If you did the previous exercise, undo those changes first.)

Listing 10.42: Adding a `setup` method to eliminate some duplication. GREEN
test/site_pages_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def setup
    @base_title = "Learn Enough Ruby Sample App"
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "#{@base_title} | Home", title_content
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "#{@base_title} | About", title_content
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
                                          assert_equal          "#
{@base_title} | Palindrome Detector", title_content
  end
end
```

10.5 Palindrome Detector

In this section, we'll complete the sample Sinatra app by adding a working palindrome detector. This will involve putting the Ruby gem developed in Chapter 8 to good use. We'll also see the first truly working HTML *form* in the Learn Enough introductory sequence.

Our first step is to add a palindrome gem. I recommend using your own, but if for any reason you didn't publish one you can use mine, which is shown in Listing 10.43.

Listing 10.43: Adding a palindrome gem.
Gemfile

Click here to view code image

```
source 'https://rubygems.org'

ruby '3.1.1'   # Change this line if you're using a different Ruby v
ersion.

gem 'sinatra',           '2.2.0'
gem 'puma',              '5.6.4'
gem 'rerun',             '0.13.1'
gem 'mhartl_palindrome', '0.1.0'

group :test do

  gem 'minitest',           '5.15.0'
  gem 'minitest-reporters', '1.5.0'
  gem 'rack-test',          '1.1.0'
  gem 'rake',               '13.0.6'
  gem 'nokogiri',           '1.13.3'
end
```

Then we install as usual:

```
$ bundle _2.3.10_ install
```

We also need to include the palindrome gem in our app (Listing 10.44).

Listing 10.44: Adding the palindrome gem to the app.
app.rb

Click here to view code image

```
require 'sinatra'
require 'mhartl_palindrome'

get '/' do
  @title = 'Home'
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end
```

With that prep work done, we're now ready to add a form to our Palindrome Detector page, which is currently just a placeholder (Figure 10.13). The form consists of three principal parts: a `form` tag to define the form, a `textarea` for entering a phrase, and a button for submitting the phrase to the server.

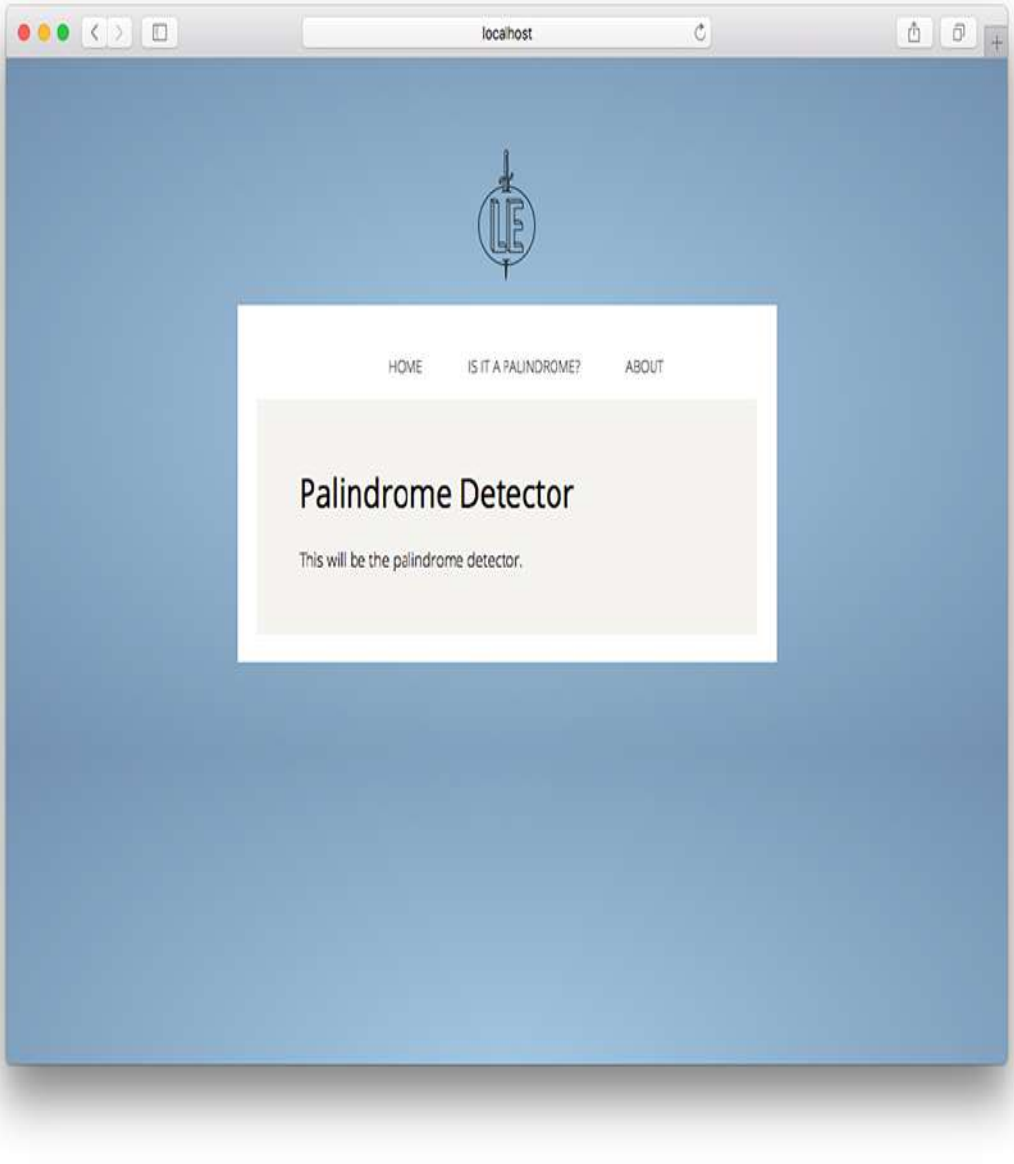


Figure 10.13: The current state of the palindrome page.

Let's work inside out. The `button` has two attributes: a CSS class for styling and a `type` indicating that it's designed to submit information:

Click here to view code image

```
<button class="form-submit" type="submit">Is it a palindrome?
</button>
```

The `textarea` has three attributes: a `name` attribute, which as we'll see in a moment passes important information back to the server, along with `rows` and `cols` to define the size of

the textarea box:

Click here to view code image

```
<textarea name="phrase" rows="10" cols="60"></textarea>
```

The `textarea` tag's content is the default text displayed in the browser, which in this case is just blank.

Finally, the `form` tag itself has three attributes: a CSS `id`, which isn't used here but is conventional to include; an `action`, which specifies the action to take when submitting the form; and a `method` indicating the HTTP request method to use:

Click here to view code image

```
<form id="palindrome_tester" action="/check" method="post">
```

We saw as early as Listing 1.8 how Sinatra apps define responses to URLs:

```
get '/' do
  'hello, world!'
end
```

Here `get` is a Sinatra function specifying how to respond when someone hits the root URL / with a `GET` request, which is the kind of request a web browser sends for an ordinary click. In contrast, a `POST` request is the kind of request typically submitted by a form. (As you might guess, there's a corresponding Sinatra function called `post` for handling this kind of request, as we'll see in a moment.)

Putting the above discussion together (and adding a `br` tag to add a line break) yields the form shown in Listing 10.45. Our updated Palindrome Detector page appears in Figure 10.14.

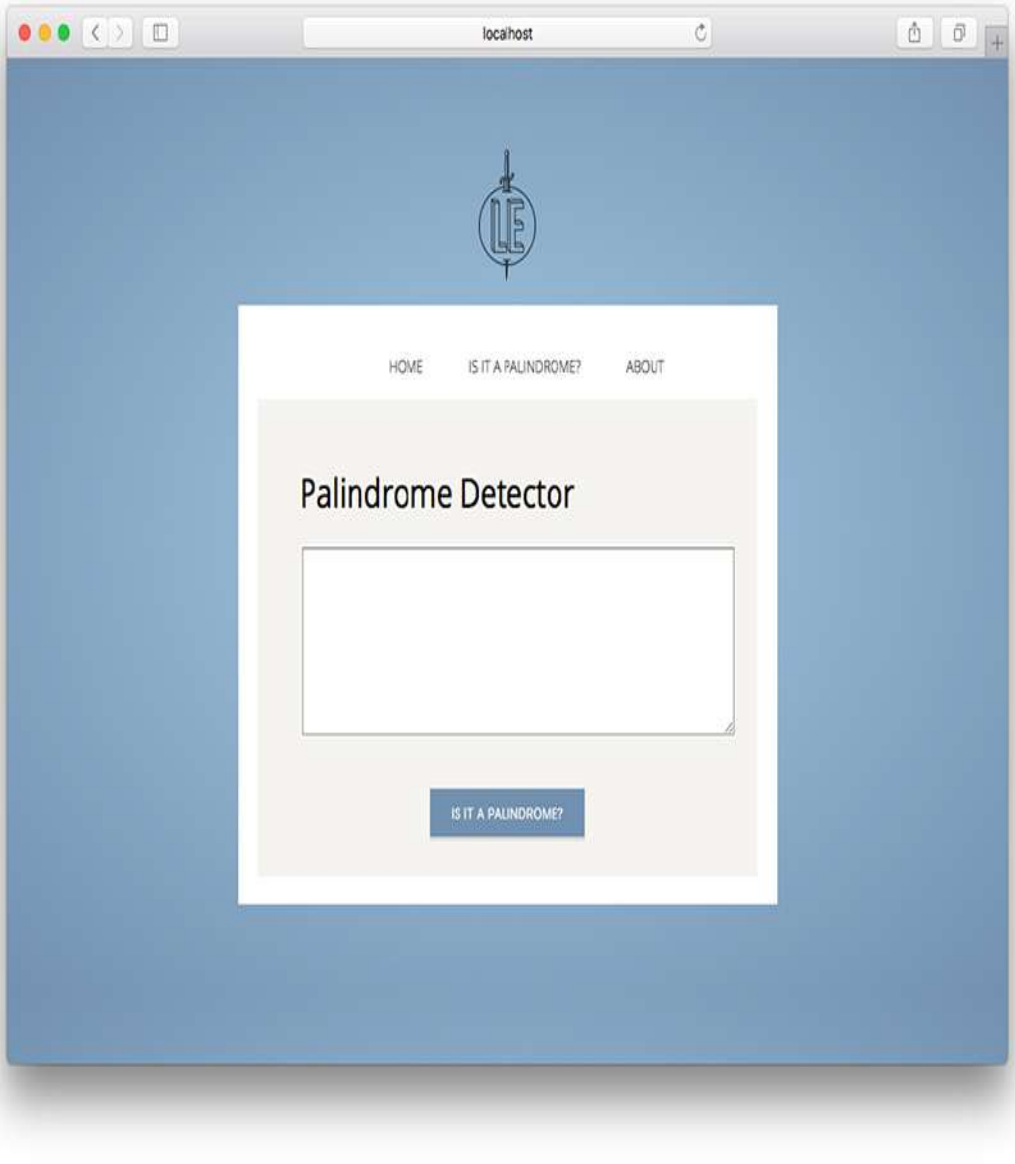


Figure 10.14: The new palindrome form.

Listing 10.45: Adding a form to the palindrome page.
views/palindrome.erb

Click here to view code image

```
<h1>Palindrome Detector</h1>

<form id="palindrome_tester" action="/check" method="post">
  <textarea name="phrase" rows="10" cols="60"></textarea>
  <br>
```

```
    <button class="form-submit" type="submit">Is it a palindrome?
</button>
</form>
```

The form in Listing 10.45 is, apart from cosmetic details, identical to the analogous form developed in *Learn Enough JavaScript to Be Dangerous*:

Click here to view code image

```
<form id="palindromeTester">
  <textarea name="phrase" rows="10" cols="30"></textarea>
  <br>
  <button type="submit">Is it a palindrome?</button>
</form>
```

In that case, though, we "cheated" by using a JavaScript event listener to intercept the submit request from the form, and no information ever got sent from the client (browser) to the server. (It's important to understand that, when developing web applications on a local computer, the client and server are the same physical machine, but in general they are different.)

This time, we won't cheat: The request will really go all the way to the server, which means we'll have to handle the `POST` request on the back-end. As hinted above, the way to do this is with the `post` function:

Click here to view code image

```
post '/check' do
  # Do something to handle the submission
end
```

Here the name of the URL path, `/check`, matches the value of the `action` parameter in the form (Listing 10.45).

To get our first hint of what form submission does, we'll use one of my favorite heavy-handed debugging tricks (Box 5.1), which is to `raise` an exception right in `app.rb`. In this case, we'll `raise` the contents of a special object called `params`, called using `inspect` to ensure that the result is a proper string. The code appears in Listing 10.46, while the result of submitting "Madam, I'm Adam." in the form appears in Figure 10.15.

Figure 10.15: The result of a `raise` after form submission.

Listing 10.46: Investigating the effects of a form submission.
app.rb

Click here to view code image

```
require 'sinatra'
require 'mhartl_palindrome'

get '/' do
  @title = 'Home'
```

```
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end

post '/check' do
  raise params.inspect
end
```

As seen in Figure 10.15, `params` is a hash (Section 4.4), with key `"phrase"` and value `"Madam, I'm Adam."`:

Click here to view code image

```
{ "phrase" => "Madam, I'm Adam." }
```

This `params` hash is created automatically by Sinatra according to the key–value pairs in the form (Listing 10.45). In this case, we have only one such pair, with key given by the `name` attribute of the `textarea` (`"phrase"`) and value given by the string entered by the user. By the way, inside the application code it's possible to use a symbol key instead, and indeed this is the more common convention, so that

```
params[:phrase]
```

extracts the value of the phrase.

Now that we know about the existence and contents of `params`, detecting a palindrome is easy: Just extract the phrase and call `palindrome?` on it. If we put the phrase into an instance variable called `@phrase`, our code would look something like this in plain Ruby:

Listing 10.47: What our palindrome results might look like in plain Ruby.

Click here to view code image

```
if @phrase.palindrome?
  puts "\"#{@phrase}\" is a palindrome!"
else
  puts "\"#{@phrase}\ "isn't a palindrome."
end
```

We can do the same basic thing using embedded Ruby (Section 10.4), only using `<%= ... %>` instead of interpolation, and surrounding any other code in `<% ... %>` tags. This is the same syntax we've seen before, only without an equals sign `=`, which tells ERB to evaluate the code but *not* to insert it into the page. Schematically, it looks something like Listing 10.48.

Listing 10.48: Schematic code for the palindrome result.

Click here to view code image

```
<% if @phrase.palindrome? %>
   "<%= @phrase %>" is a palindrome!
<% else %>
    "<%= @phrase %>" isn't a palindrome!
<% end %>
```

We'll create a file called `result.erb`:

```
$ touch views/result.erb
```

The code itself is an expanded version of Listing 10.48 with a few more HTML tags for a better appearance, as shown in Listing 10.49.

Listing 10.49: Displaying the palindrome result using ERB.
views/result.erb

Click here to view code image

```
<h1>Palindrome Result</h1>

<% if @phrase.palindrome? %>
  <div class="result result-success">
    <p>"<%= @phrase %>" is a palindrome!</p>
  </div>

<% else %>
  <div class="result result-fail">
```

```
    <p>"<%= @phrase %>" isn't a palindrome!</p>
  </div>
<% end %>
```

All that's left now is handling the submission, putting the value of `params[:phrase]` in `@phrase`, and rendering the result. Since the form issues an HTTP `POST` request, the trick is to use the `post` function in place of the `get` function we've used on every page so far. The URL itself is `'/check'`, as indicated by the value of the `action` attribute in Listing 10.45. The result appears in Listing 10.50.

Listing 10.50: Handling a palindrome form submission.
app.rb

Click here to view code image

```
require 'sinatra'
require 'mhartl_palindrome'

get '/' do
  @title = 'Home'
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end

post '/check' do
  @phrase = params[:phrase]
  erb :result
end
```

With that, our palindrome detector should be working! Let's see if it can correctly identify one of the most ancient palindromes, the so-called Sator Square first found in the ruins of Pompeii (Figure 10.16).[4] (Authorities differ on the exact meaning of the Latin words in the square, but the likeliest translation is "The sower [farmer] Arepo holds the wheels with effort.")

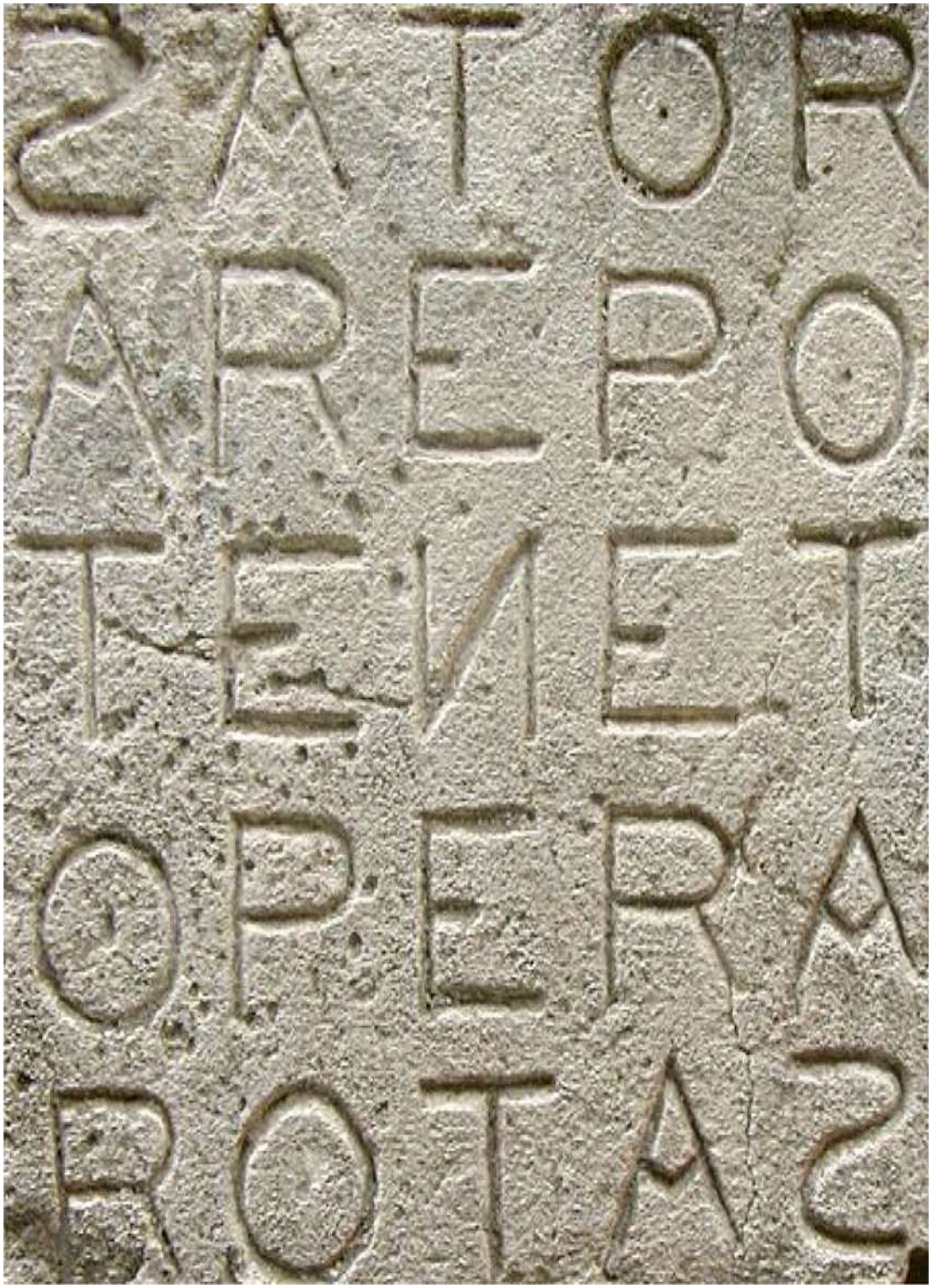


Figure 10.16: A Latin palindrome from the lost city of Pompeii.

[4]Image courtesy of CPA Media Pte Ltd/Alamy Stock Photo.

Entering the text “SATOR AREPO TENET OPERA ROTAS” (Figure 10.17) and submitting it leads to the result shown in Figure 10.18.

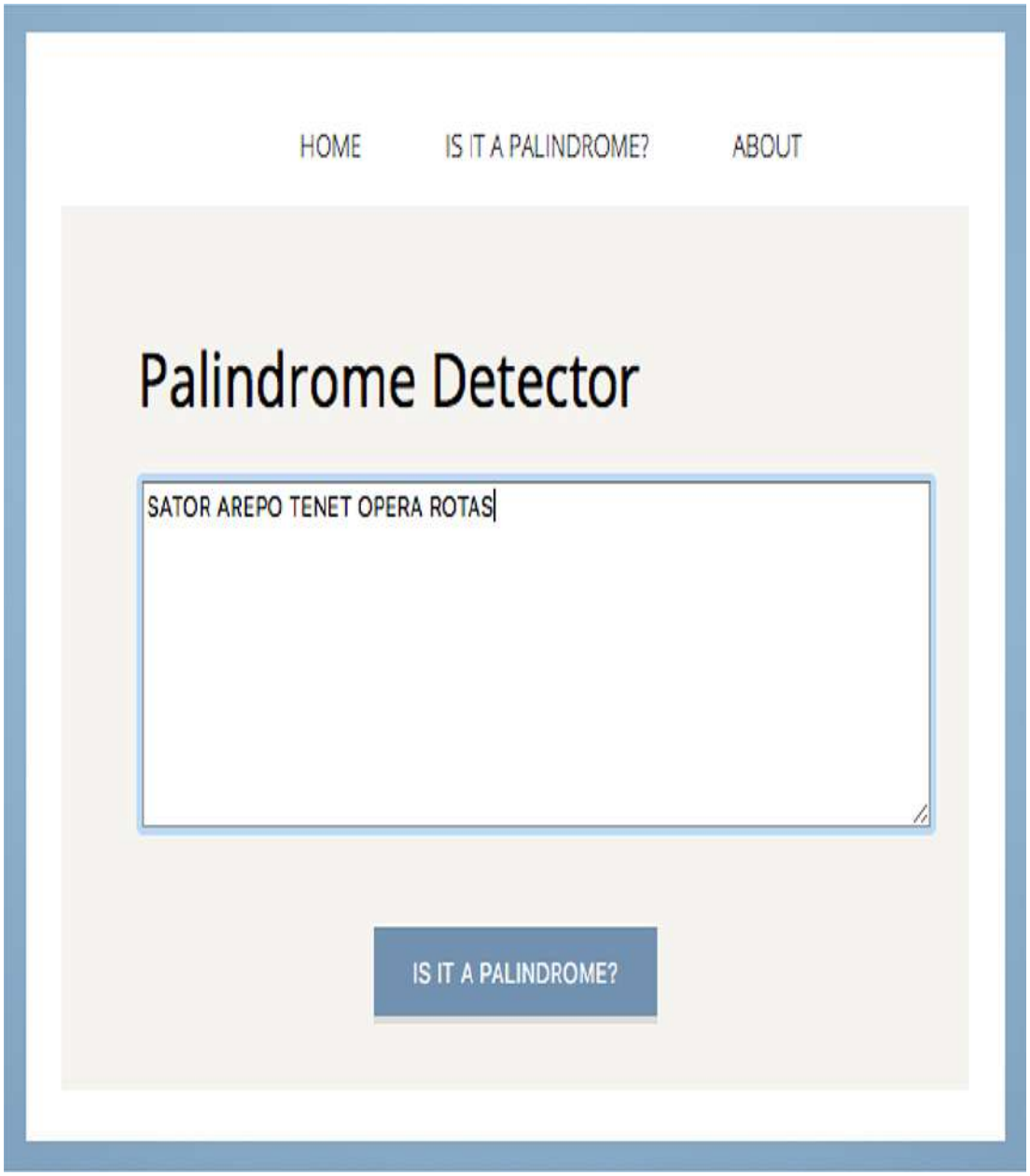


Figure 10.17: A Latin palindrome?

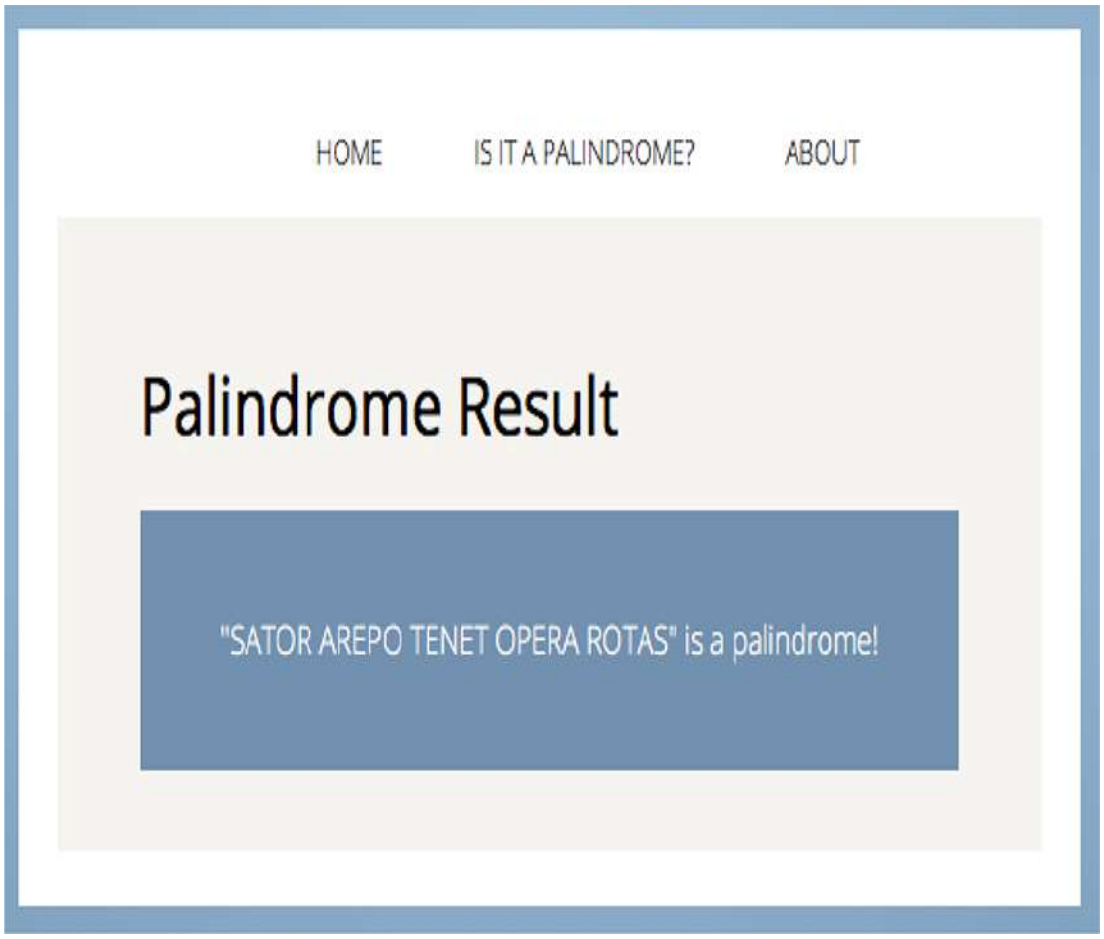


Figure 10.18: A Latin palindrome!

10.5.1 Form Tests

Our application is now working, but note that testing a *second* palindrome requires clicking on “IS IT A PALINDROME?” It would be more convenient if we included the same submission form on the result page as well.

To do this, we’ll first add a simple test for the presence of a `form` tag on the palindrome page. Because the tests we’ll be adding are specific to that page, we’ll create a new test file to contain them:

Click here to view code image

```
$ touch test/palindrome_test.rb
```

The test itself is closely analogous to the `h1` test in Listing 10.18, as shown in Listing 10.51.

Listing 10.51: Testing for the presence of a form tag. GREEN
palindrome_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_form_presence
    get '/palindrome'
    assert doc(last_response).at_css('form')
  end
end
```

Now we'll add tests for the existing form submission for both non-palindromes and palindromes. Just as `get` in tests issues a GET request, `post` in tests issues a POST request. The first argument of `post` is the URL, and the second is the `params` hash:

Click here to view code image

```
post '/check', phrase: "Not a palindrome"
```

(Note that this uses the more compact `key: value` hash notation mentioned in Section 4.4; per the note near the end of Section 10.3, we have also omitted the curly braces.)

To test the response, we'll verify that the text in the page's paragraph tag includes the right result. The most elegant way to do this is with `assert_-includes` from the minitest assertions, where

Click here to view code image

```
assert_includes result, substring
```

is effectively equivalent to

Click here to view code image

```
assert result.include?(substring)
```

using the `String#include?` method discussed in Section 2.5. (As with `assert_equal`, using the native assertion is generally more convenient because the failing messages are more descriptive.) For a non-palindrome, the assertion would look something like this:

Click here to view code image

```
assert_includes doc(last_response).at_css('p').content, "isn't a pal
indrome"
```

Taking the ideas above and applying them to both non-palindromes and palindromes gives the tests shown in Listing 10.52 (with only inner lines highlighted for brevity).

Listing 10.52: Adding tests for form submission. GREEN
palindrome_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_form_presence
    get '/palindrome'
    assert doc(last_response).at_css('form')
end

  def test_non_palindrome_submission
    post '/check', phrase: "Not a palindrome"
    assert_includes doc(last_response).at_css('p').content, "isn't a
 palindrome"
  end

  def test_palindrome_submission
    post '/check', phrase: "Able was I, ere I saw Elba."
    assert_includes doc(last_response).at_css('p').content, "is a pa
```

```
lindrome"
  end
end
```

Because we were testing existing functionality, our tests should already be GREEN:

Listing 10.53: GREEN

Click here to view code image

```
$ bundle exec rake test
6 tests, 15 assertions, 0 failures, 0 errors, 0 skips
```

As a capstone to our development, we'll now add a form on the result page using the RED, GREEN, refactor cycle that is a hallmark of TDD. Without loss of generality, we'll work in the non-palindrome test; all we need to do is add a `form` test identical to the one in Listing 10.51, as shown in Listing 10.54.

Listing 10.54: Adding a test for a form on the result page. RED
palindrome_test.rb

Click here to view code image

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_form_presence
    get '/palindrome'
    assert doc(last_response).at_css('form')
  end

  def test_non_palindrome_submission
    post '/check', phrase: "Not a palindrome"
    assert_includes doc(last_response).at_css('p').content, "isn't a
 palindrome"
    assert doc(last_response).at_css('form')
  end

  def test_palindrome_submission
```

```
    post '/check', phrase: "Able was I, ere I saw Elba."
    assert_includes doc(last_response).at_css('p').content, "is a pa
lindrome"
  end
end
```

As required, the test suite is now RED:

Listing 10.55: RED

Click here to view code image

```
$ bundle exec rake test
6 tests, 16 assertions, 1 failures, 0 errors, 0 skips
```

We can get the tests to GREEN again by copying the form from `palindrome.erb` and pasting it into `result.erb`, as shown in Listing 10.56.

Listing 10.56: Adding a form to the result page. GREEN
views/result.erb

Click here to view code image

```
<h1>Palindrome Result</h1>

<% if @phrase.palindrome? %>
  <div class="result result-success">
    <p>"<%= @phrase %>" is a palindrome!</p>
  </div>
<% else %>
  <div class="result result-fail">
    <p>"<%= @phrase %>" isn't a palindrome!</p>
  </div>
<% end %>

<h2>Try another one!</h2>

<form id="palindrome_tester" action="/check" method="post">
  <textarea name="phrase" rows="10" cols="60"></textarea>
  <br>
    <button class="form-submit" type="submit">Is it a palindrome?
</button>
</form>
```

This gets our tests to GREEN:

Listing 10.57: GREEN

Click here to view code image

```
$ bundle exec rake test
6 tests, 16 assertions, 0 failures, 0 errors, 0 skips
```

That cut-and-paste should have set your programmer Spidey-sense tingling though: It's repetition! Pasting in content is a clear violation of the Don't Repeat Yourself (DRY) principle. Happily, we saw how to eliminate such duplication in the case of the site navigation by refactoring the code to use a partial (Listing 10.40), which we can apply to this case as well. As with the nav, we'll first create a separate file for the form HTML:

Click here to view code image

```
$ touch views/palindrome_form.erb
```

Then we can fill it with the form (Listing 10.58), while replacing the form with an ERB rendering on the result page (Listing 10.59) and on the main palindrome page itself (Listing 10.60).

Listing 10.58: A partial for the palindrome form. GREEN
views/palindrome_form.erb

Click here to view code image

```
<form id="palindrome_tester" action="/check" method="post">
  <textarea name="phrase" rows="10" cols="60"></textarea>
  <br>
    <button class="form-submit" type="submit">Is it a palindrome?
</button>
</form>
```

Listing 10.59: Rendering the form partial on the result page. GREEN
views/result.erb

Click here to view code image

```
<h1>Palindrome Result</h1>
```

```
<% if @phrase.palindrome? %>
  <div class="result result-success">
    <p>"<%= @phrase %>" is a palindrome!</p>
  </div>
<% else %>
  <div class="result result-fail">
    <p>"<%= @phrase %>" isn't a palindrome!</p>
  </div>
<% end %>

<h2>Try another one!</h2>

<%= erb :palindrome_form %>
```

Listing 10.60: Rendering the form partial on the main palindrome page. GREEN
views/palindrome.erb

```
<h1>Palindrome Detector</h1>

<%= erb :palindrome_form %>
```

As required for a refactoring, the tests are still GREEN!

Listing 10.61: GREEN

Click here to view code image

```
$ bundle exec rake test
6 tests, 16 assertions, 0 failures, 0 errors, 0 skips
```

Submitting the Sator Square palindrome shows that the form on the result page is rendering properly, as shown in Figure 10.19.

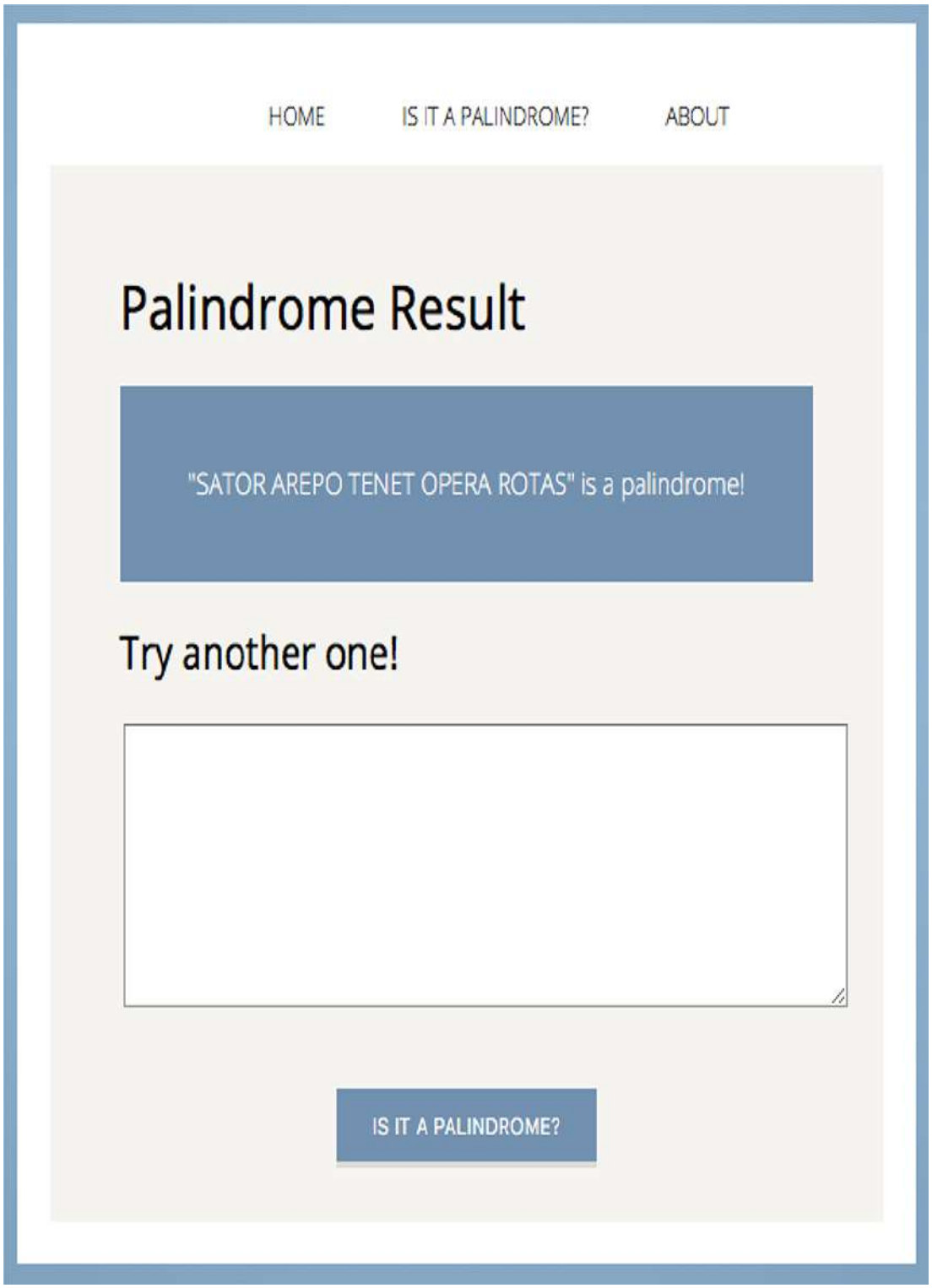


Figure 10.19: The form on the result page.

Filling the textarea with one of my favorite looooong palindromes (Figure 10.20) gives the result shown in Figure 10.21.[5]

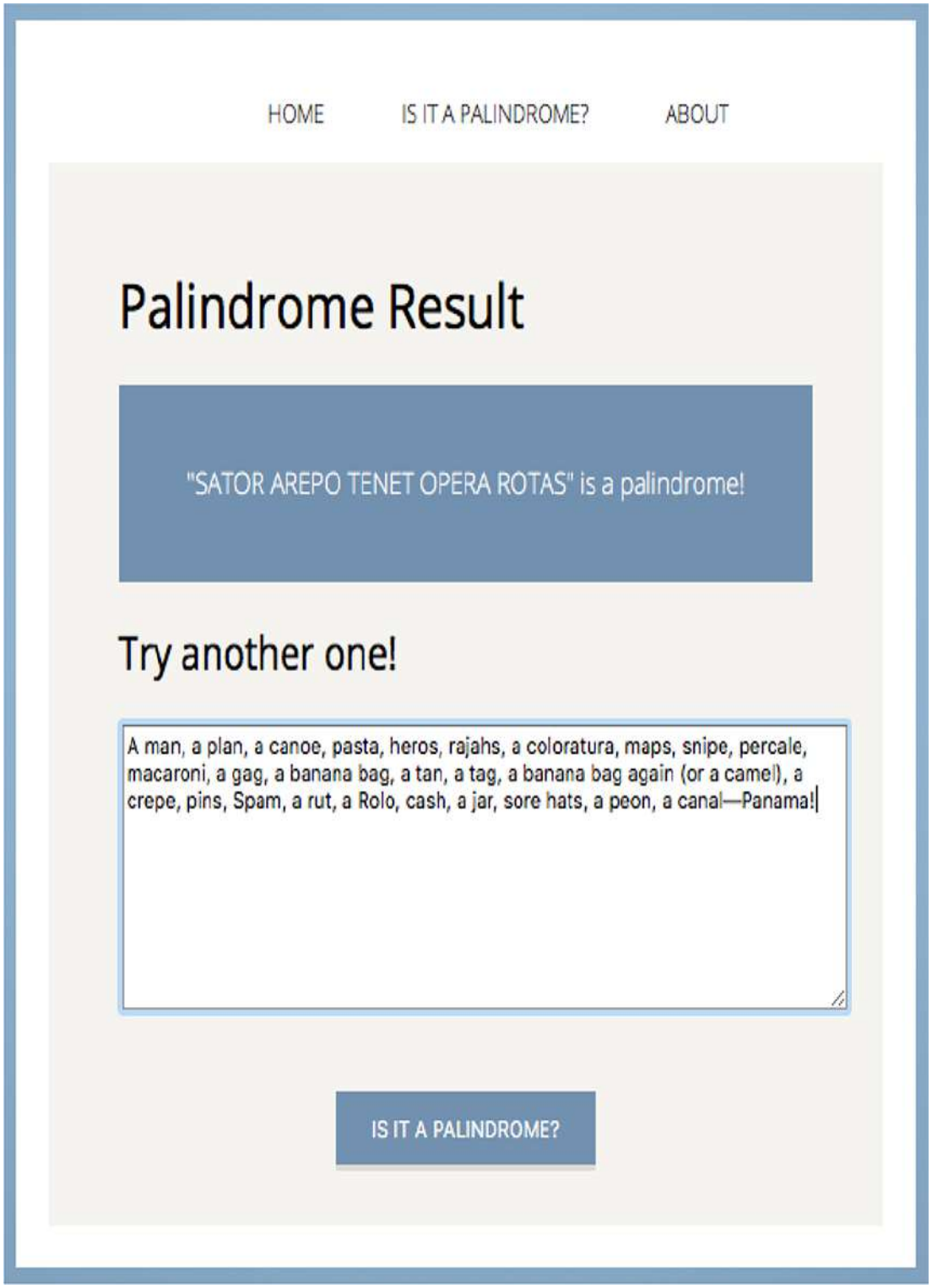


Figure 10.20: Entering a long string in the form’s textarea field.

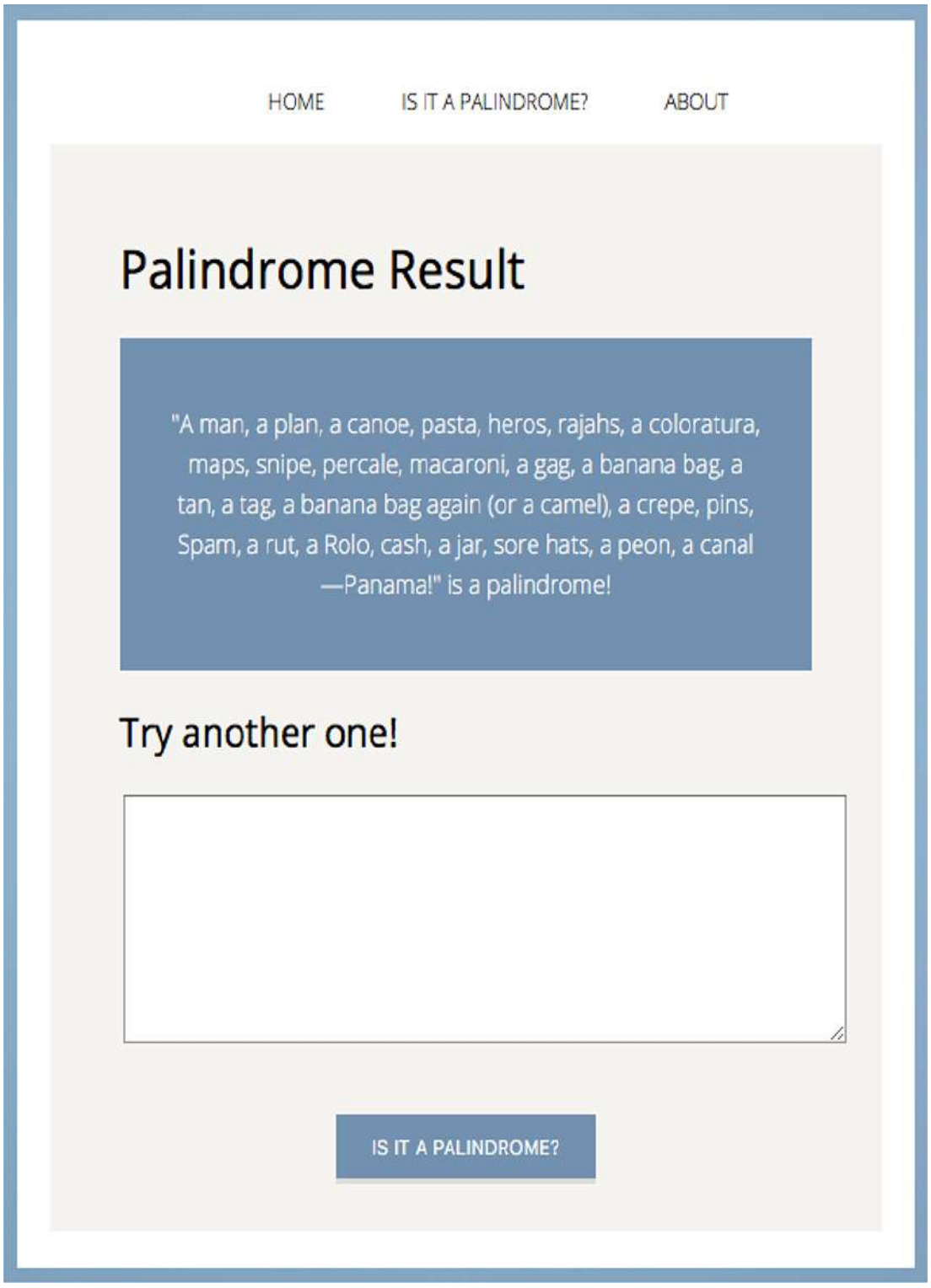


Figure 10.21: That long string is a palindrome!

[5] The amazingly long palindrome in Figure 10.20 was created in 1983 by pioneering computer scientist Guy Steele with the aid of a custom program.

And with that—“A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, macaroni, a gag, a banana bag, a tan, a tag, a banana bag again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats, a peon, a canal—Panama!”—we’re done with our palindrome detector web application. Whew!

The only thing left is to commit and deploy:

Click here to view code image

```
$ git add -A
$ git commit -am "Finish working palindrome detector"
$ git push heroku
```

Note that, after pushing once, we can omit the branch name (`main`) and just type `git push heroku`. The result is a palindrome application working in production (Figure 10.22)! (To learn how to host a Heroku site using a custom domain instead of a herokuapp.com subdomain, see the free tutorial *Learn Enough Custom Domains to Be Dangerous*.)

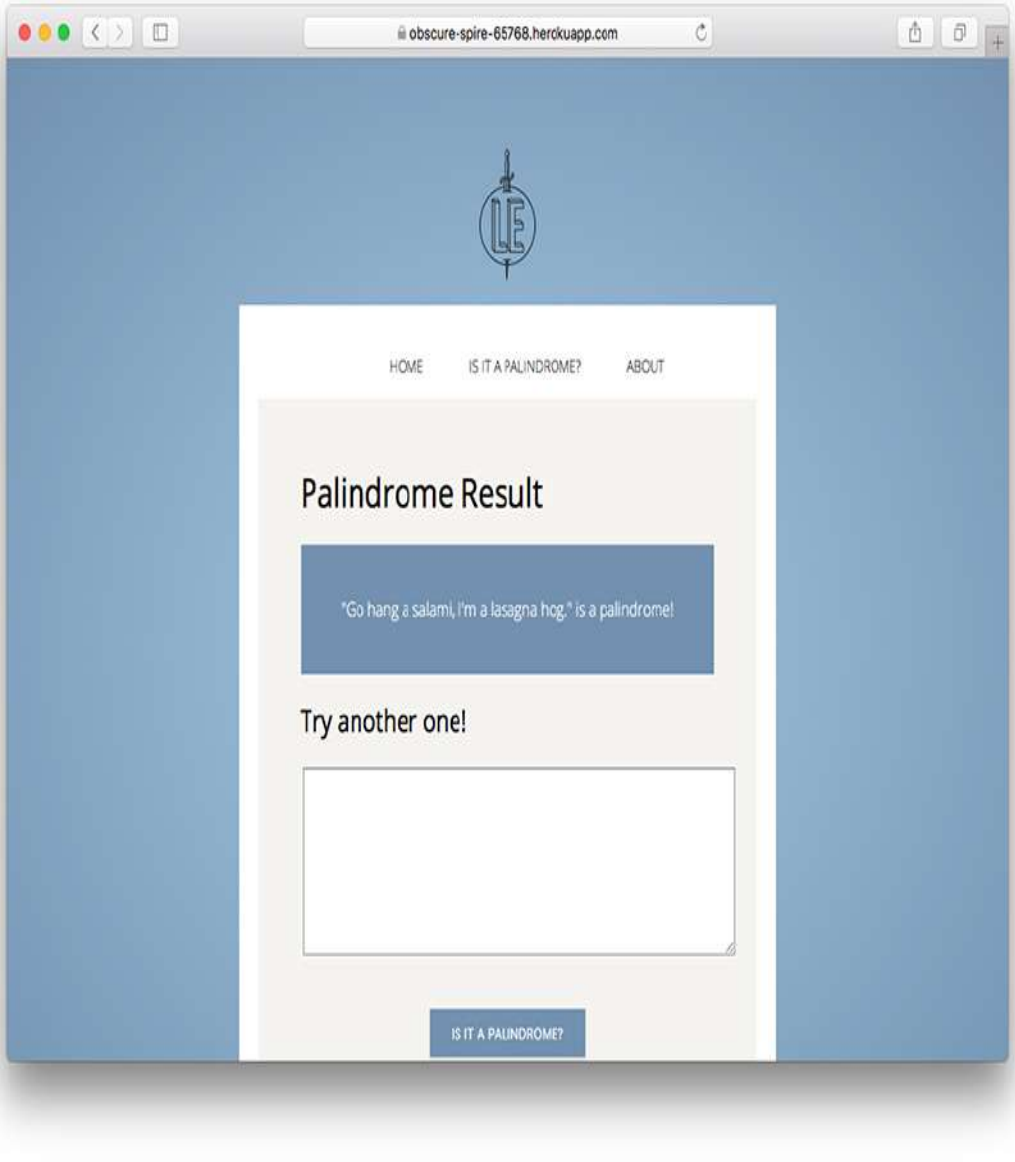


Figure 10.22: Our palindrome detector working on the live Web.

10.5.2 Exercises

1. Confirm by submitting an empty textarea that the palindrome detector currently returns `true` for the empty string, which is a flaw in the palindrome gem itself. What happens if you submit a bunch of spaces?
2. In the palindrome gem, write a test asserting that a string of spaces *isn't* a palindrome (RED). Then write the application code necessary to get that test

to GREEN. Bump the version number and publish your gem as in Section 8.5.1. (You can refer to my version if you'd like some help.)

3. After waiting a few minutes for RubyGems to update, bump the version number in the `Gemfile` (Listing 10.43), update the gems using `bundle update`, and confirm that an empty submission is no longer a palindrome, both locally and (after redeploying) in production.
4. Make a commit and deploy the changes.

10.6 Conclusion

Congratulations! You now know enough Ruby to be *dangerous*.

With the skills developed in this tutorial, you now have the preparation to go in multiple different directions. The most natural follow-on is the *Ruby on Rails Tutorial*, which teaches you how to make professional-grade web applications using the Ruby on Rails web framework:

- *Ruby on Rails Tutorial*

Learn Enough Ruby to Be Dangerous is perfect preparation for the *Ruby on Rails Tutorial*; in particular, you will find that the background in Sinatra will make learning Rails much easier than it would be otherwise.

If you haven't followed it already, I also recommend learning the basics of JavaScript:

- *Learn Enough JavaScript to Be Dangerous*

You can get surprisingly far with Ruby alone, but JavaScript's unique ability to be run in the browser means everyone who makes web apps should learn it eventually.

For more about Ruby (and programming generally), I recommend these fine titles:

- *Learn to Program* by Chris Pine (Pragmatic Bookshelf, 2021)
- *Programming Ruby* by Dave Thomas (Pragmatic Bookshelf, 2013)

Finally, for people who want the most solid foundation possible in technical sophistication, Learn Enough All Access is a subscription service that has special online versions of all the Learn Enough books and over 40 hours of streaming video tutorials. We hope you'll check it out!

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

https://github.com/learnenough/learn_enough_ruby_code_listings

```
$ ruby -v
ruby 3.1.1p18 (2022-02-18 revision 53f5fc4236) [x86_64-linux]
```

```
$ ruby -v
-bash: ruby: command not found
```

```
$ # on cloud IDE
$ rvm get stable
$ rvm install 3.1.1
$ rvm --default use 3.1.1
```

```
$ ruby -v
ruby 3.1.1p18 (2022-02-18 revision 53f5fc4236) [x86_64-linux]
```

```
IRB.conf[:PROMPT_MODE] = :SIMPLE
IRB.conf[:AUTO_INDENT_MODE] = false
```

```
>> puts "hello, world!"
hello, world!
=> nil
```

```
$ cd    # Change to the home directory; use cd ~/environment on the cloud IDE.
$ mkdir -p repos/ruby_tutorial
$ cd repos/ruby_tutorial
$ touch hello.rb
```

```
puts "hello, world!", "how's it going?"
```

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

puts "hello, world!"
```

```
$ echo "gem: --no-document" >> ~/.gemrc
```

```
require 'sinatra'

get '/' do
  'hello, world!'
end
```

```
$ ruby hello_app.rb
== Sinatra has taken the stage on 4567 for development with
Maximum connections set to 1024
Listening on localhost:4567, CTRL+C to stop
```

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

```
$ git remote add origin https://github.com/<username>/ruby_tutorial.git
$ git push -u origin main
```

```
$ heroku --version    # will work only if heroku is installed
heroku: command not found
```

```
$ source <(curl -sL https://cdn.learnenough.com/heroku_install)
```

```
$ heroku --version
heroku/7.59.2 linux-x64 node-v12.21.0
```

```
$ heroku login          # on a native system but not on the cloud IDE
$ # Spawns a browser window. Log in with your email and Heroku password.
```

```
$ heroku login --interactive     # on the cloud IDE
Email: <your email>
Password: <your API Key, NOT your Heroku password>
```

```
$ heroku create
Creating app... done, ⬢ damp-depths-3
https://damp-depths-3.herokuapp.com/ | https://git.heroku.com/damp-depths-3.git
```

```
require './hello_app'
run Sinatra::Application
```

```
source 'https://rubygems.org'

ruby '3.1.2'   # Change this line if you're using a different Ruby version.

gem 'sinatra', '2.2.2'
gem 'puma',    '5.6.5'
```

```
$ gem install bundler -v 2.3.10
$ bundle _2.3.10_ install
$ bundle _2.3.10_ lock --add-platform x86_64-linux
$ git add -A
$ git commit -m "Add deployment configuration"
```

```
$ irb
>> "foo" + "bar"             # String concatenation
=> "foobar"
>> "ant" + "bat" + "cat"     # Multiple strings can be concatenated at once.
=> "antbatcat"
```

```
>> first_name + " " + last_name
=> "Michael Hartl"
```

```
>> "#{first_name} is my first name."
=> "Michael is my first name."
```

```
>> first_name + " " + last_name     # Concatenation, with a space in between
=> "Michael Hartl"
>> "#{first_name} #{last_name}"     # The equivalent interpolation
=> "Michael Hartl"
```

```
>> 'foo'           # A single-quoted string
=> "foo"
>> 'foo' + 'bar'
=> "foobar"
```

```
>> '#{first_name} #{last_name}'      # No interpolation!
=> "\#{first_name} \#{last_name}"
```

```
>> '\n'        # A literal 'backslash n' combination
=> "\\n"
```

```
>> 'Newlines (\n) and tabs (\t) both use the backslash character: \.'
=> "Newlines (\\n) and tabs (\\t) both use the backslash character: \\."
```

```
$ irb
>> 'It's not easy being green'
```

```
$ irb
>> 'It's not easy being green'
^C
>> 'It\'s not easy being green'
=> "It's not easy being green"
```

```
>> puts "hello, world!"      # Print output
hello, world!
=> nil
```

```
$ irb
>> "badger".length    # Accessing the "length" property of a string
=> 6
>> "".length          # The empty string has zero length.
=> 0
```

```
>> password = "foobar"
>> if password.length < 6
>>   "Password is too short."
>> else
>>   "Password is long enough."
>> end
=> "Password is long enough."
```

```
>> password = "goldilocks"
>> if password.length < 6
>>   "Password is too short."
>> elsif password.length < 50
>>   "Password is just right!"
>> else
>>   "Password is too long."
>> end
=> "Password is just right!"
```

```
>> password = "foo"
>> "Password is too short." if password.length < 6
=> "Password is too short."
```

```
>> "Password is too short." unless password.length >= 6
=> "Password is too short."
```

```
>> x = "foo"
>> y = ""
>> if x.length == 0 && y.length == 0
>>   "Both strings are empty!"
>> else
>>   "At least one of the strings is nonempty."
>> end
=> "At least one of the strings is nonempty."
```

```
>> if x.length == 0 || y.length == 0
>>   "At least one of the strings is empty!"
>> else
>>   "Neither of the strings is empty."
>> end
=> "At least one of the strings is empty!"
```

```
>> if x.empty? && y.empty?
>>   "Both strings are empty!"
>> else
>>   "At least one of the strings is nonempty."
>> end
=> "At least one of the strings is nonempty."
```

```
>> first_name = "Michael"
>> username = first_name.downcase
>> "#{username}@example.com"     # Sample email address
=> "michael@example.com"
```

`include? other_str → true or false`

```
>> soliloquy = "To be, or not to be, that is the question:"
>> soliloquy.include?("To be")        # Does it include the substring "To be"?
=> true
>> soliloquy.include?("question")     # What about "question"?
=> true
>> soliloquy.include?("nonexistent")  # This string doesn't appear.
=> false
>> soliloquy.include?("TO BE")        # String inclusion is case-sensitive.
=> false
```

```
>> puts soliloquy   # Just a reminder of what the string is
To be, or not to be, that is the question:
>> soliloquy[0]
=> "T"
>> soliloquy[1]
=> "o"
>> soliloquy[2]
=> " "
```

```
>> for i in 0..(soliloquy.length - 1)
?>   puts soliloquy[i]
>> end
T
o

b
e
.
.
.
t
i
o
n
:
```

```
>> "ant bat cat".split(" ")     # Split a string into a three-element array.
=> ["ant", "bat", "cat"]
```

```
>> "ant,bat,cat".split(",")
=> ["ant", "bat", "cat"]
>> "ant, bat, cat".split(", ")
=> ["ant", "bat", "cat"]
>> "antheybatheycathey".split("hey")
=> ["ant", "bat", "cat"]
```

```
>> "badger".split("")
=> ['b', 'a', 'd', 'g', 'e', 'r']
```

```
>> "ant bat cat".split
=> ["ant", "bat", "cat"]
>> "ant     bat\t\tcat\n    duck".split
=> ["ant", "bat", "cat", "duck"]
```

```
>> a = "badger".split("")
=> ["b", "a", "d", "g", "e", "r"]
```

```
>> soliloquy = "To be, or not to be, that is the question:"
>> a = ["badger", 42, soliloquy.include?("To be")]
=> ["badger", 42, true]
>> a[2]
=> true
>> a[3]
=> nil
```

```
>> (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> ('a'..'z').to_a
=> ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
    "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
```

```
>> a = [42, 8, 17, 99]
=> [42, 8, 17, 99]
>> a.include?(42)        # Test for element inclusion.
=> true
>> a.include?("foo")
=> false
```

```
>> a.sort
=> [8, 17, 42, 99]
>> a                          # `a` hasn't changed as the result of `sort`.
=> [42, 8, 17, 99]
```

```
>> a.sort!
=> [8, 17, 42, 99]
>> a                          # `a` has changed as the result of `sort!`.
=> [8, 17, 42, 99]
```

```
>> a.reverse
=> [99, 42, 17, 8]
>> a                        # Like `sort`, `reverse` doesn't mutate the array.
=> [8, 17, 42, 99]
```

```
>> a.push(6)                    # Pushing onto an array
=> [8, 17, 42, 99, 6]
>> a.push("foo")
=> [8, 17, 42, 99, 6, "foo"]
>> a.pop                        # `pop` returns the value itself
=> "foo"
>> a.pop
=> 6
>> a.pop
=> 99
>> a
=> [8, 17, 42]
```

```
>> the_answer_to_life_the_universe_and_everything = a.pop
=> 42
```

```
>> a << "badger"
=> [8, 17, "badger"]
>> a << "ant" << "bat" << "cat"
=> [8, 17, "badger", "ant", "bat", "cat"]
```

```
>> a = ["ant", "bat", "cat", 42]
=> ["ant", "bat", "cat", 42]
>> a.join                          # Join on default (empty space).
=> "antbatcat42"
>> a.join(", ")                    # Join on comma-space.
=> "ant, bat, cat, 42"
>> a.join(" -- ")                  # Join on double dashes.
=> "ant -- bat -- cat -- 42"
```

```
>> for i in 0..(a.length - 1)
?>   puts a[i]
>> end
ant
bat
cat
42
```

```
>> a.each do |element|
?>   puts element
>> end
ant
bat
cat
42
```

```
>> s = String.new("A man, a plan, a canal—Panama!")
=> "A man, a plan, a canal—Panama!"
>> s.split(", ")
=> ["A man", "a plan", "a canal—Panama!"]
```

```
>> moon_landing = Time.new(1969, 7, 20, 20, 17, 40)
=> 1969-07-20 20:17:40 -0700
>> moon_landing.day
=> 20
```

```
>> moon_landing = Time.utc(1969, 7, 20, 20, 17, 40)
=> 1969-07-20 20:17:40 UTC
```

```
>> moon_landing.wday          # wday = weekday
=> 0
```

```
>> DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
?>             "Thursday", "Friday", "Saturday"]
>> DAYNAMES[moon_landing.wday]
=> "Sunday"
>> DAYNAMES[Time.now.wday]
=> "Tuesday"
```

```ruby
require 'sinatra'

get '/' do
  DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"]
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

```ruby
require 'sinatra'

DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"]

get '/' do
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

```ruby
require 'sinatra'

get '/' do
  dayname = Date::DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

```
>> "no match".match(zip_code)
=> nil
>> "Beverly Hills 90210".match(zip_code)
=> #<MatchData "90210">
```

```
>> s = "Beverly Hills 90210"
>> puts "It's got a ZIP code!" if s.match(zip_code)
It's got a ZIP code!
```

```
>> s =  "Beverly Hills 90210 was a '90s TV show set in Los Angeles."
>> s += " 91125 is another ZIP code in the Los Angeles area."
=> "Beverly Hills 90210 was a '90s TV show set in Los Angeles. 91125 is another
 ZIP code in the Los Angeles area."
```

```
>> "ant bat cat duck".split(" ")
=> ['ant', 'bat', 'cat', 'duck']
```

```
>> "ant bat cat duck".split(/\s+/)
=> ["ant", "bat", "cat", "duck"]
```

```
>> "ant    bat\tcat\nduck".split(/\s+/)
=> ["ant", "bat", "cat", "duck"]
```

```
>>  "ant    bat\tcat\nduck".split
=> ["ant", "bat", "cat", "duck"]
```

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
```

Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."

```
>> user = {}                              # {} is an empty hash.
=> {}
>> user["first_name"] = "Michael"         # Key "first_name", value "Michael"
=> "Michael"
>> user["last_name"] = "Hartl"            # Key "last_name", value "Hartl"
=> "Hartl"
```

```
>> user["first_name"]       # Element access is like arrays
=> "Michael"
>> user["last_name"]
=> "Hartl"
>> user["nonexistent"]
=> nil
```

```
>> user
=> {"first_name"=>"Michael", "last_name"=>"Hartl"}
>> moonman = { "first_name" => "Buzz", "last_name" => "Aldrin" }
=> {"first_name"=>"Buzz", "last_name"=>"Aldrin"}
```

```
>> "name".split('')
=> ["n", "a", "m", "e"]
>> :name.split('')
undefined method `split' for :name:Symbol (NoMethodError)
```

```
>> :foo-bar
undefined local variable or method `bar' for main:Object (NameError)
>> :2foo
syntax error, unexpected integer literal, expecting literal content or
terminator or tSTRING_DBEG or tSTRING_DVAR (SyntaxError)
>> :"foo-bar"
=> :"foo-bar"
>> :"2foo"
=> :"2foo"
```

```
>> user = { :name => "Michael Hartl", :email => "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
>> user[:name]               # Access the value corresponding to :name.
=> "Michael Hartl"
>> user[:password]           # Access the value of an undefined key.
=> nil
```

```
>> user = { name: "Michael Hartl", email: "michael@example.com" }
=> {:name=>"Michael Hartl", :email=>"michael@example.com"}
```

```
{ name: "Michael Hartl", email: "michael@example.com" }
```

```
>> params = {}        # Define a hash called 'params' (short for 'parameters').
=> {}
>> params[:user] = { name: "Michael Hartl", email: "mhartl@example.com" }
=> {:name=>"Michael Hartl", :email=>"mhartl@example.com"}
>> params
=> {:user=>{:name=>"Michael Hartl", :email=>"mhartl@example.com"}}
>> params[:user][:email]
=> "mhartl@example.com"
```

```
>> flash = { success: "It worked!", danger: "It failed." }
=> {:success=>"It worked!", :danger=>"It failed."}
>> flash.each do |key, value|
?>   puts "Key #{key.inspect} has value #{value.inspect}"
>> end
Key :success has value "It worked!"
Key :danger has value "It failed."
```

```
>> puts (1..5).to_a             # Put an array as a string.
1
2
3
4
5
>> puts (1..5).to_a.inspect     # Put a literal array.
[1, 2, 3, 4, 5]
>> puts :name, :name.inspect
name
:name
>> puts "It worked!", "It worked!".inspect
It worked!
"It worked!"
```

```
>> p :name              # Same output as 'puts :name.inspect'
:name
```

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."
```

```
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."

uniques = {}
words = sonnet.scan(/\w+/)
```

```ruby
sonnet = "Let me not to the marriage of true minds
Admit impediments. Love is not love
Which alters when it alteration finds,
Or bends with the remover to remove.
O no, it is an ever-fixed mark
That looks on tempests and is never shaken
It is the star to every wand'ring bark,
Whose worth's unknown, although his height be taken.
Love's not time's fool, though rosy lips and cheeks
Within his bending sickle's compass come:
Love alters not with his brief hours and weeks,
But bears it out even to the edge of doom.
  If this be error and upon me proved,
  I never writ, nor no man ever loved."

# Unique words
uniques = {}
# All words in the text
words = sonnet.scan(/\w+/)

# Iterate through `words` and build up a hash of unique words.
words.each do |word|
  if uniques[word]
    uniques[word] += 1
  else
    uniques[word]  = 1
  end
end

puts uniques
```

```
$ ruby count.rb
{"Let"=>1, "me"=>2, "not"=>4, "to"=>4, "the"=>4, "marriage"=>1,
"of"=>2, "true"=>1, "minds"=>1, "Admit"=>1, "impediments"=>1,
"Love"=>3, "is"=>4, "love"=>1, "Which"=>1, "alters"=>2, "when"=>1,
```

```
"it"=>3, "alteration"=>1, "finds"=>1, "Or"=>1, "bends"=>1, "with"=>2,
"remover"=>1, "remove"=>1, "O"=>1, "no"=>2, "an"=>1, "ever"=>2,
"fixed"=>1, "mark"=>1, "That"=>1, "looks"=>1, "on"=>1, "tempests"=>1,
"and"=>4, "never"=>2, "shaken"=>1, "It"=>1, "star"=>1, "every"=>1,
"wand"=>1, "ring"=>1, "bark"=>1, "Whose"=>1, "worth"=>1, "s"=>4,
"unknown"=>1, "although"=>1, "his"=>3, "height"=>1, "be"=>2,
"taken"=>1, "time"=>1, "fool"=>1, "though"=>1, "rosy"=>1,
"lips"=>1, "cheeks"=>1, "Within"=>1, "bending"=>1, "sickle"=>1,
"compass"=>1, "come"=>1, "brief"=>1, "hours"=>1, "weeks"=>1,
"But"=>1, "bears"=>1, "out"=>1, "even"=>1, "edge"=>1, "doom"=>1,
"If"=>1, "this"=>1, "error"=>1, "upon"=>1, "proved"=>1, "I"=>1,
"writ"=>1, "nor"=>1, "man"=>1, "loved"=>1}
```

```
>> puts "hello, world!"
hello, world!
```

```
>> puts("hello, world!")
hello, world!
```

```
>> now = Time.now
>> now.wday
=> 4
```

```
>> require 'date'
>> Date::DAYNAMES
=> ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```

```
>> Date::DAYNAMES[now.wday]
=> "Thursday"
```

```
>> def day_of_the_week(time)
?>   return Date::DAYNAMES[time.wday]
>> end
```

```
>> day_of_the_week(Time.now)
=> "Thursday"
```

```
>> def day_of_the_week(time)
?>   Date::DAYNAMES[time.wday]
>> end
```

```ruby
require 'sinatra'

get '/' do
  DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"]
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

```ruby
require 'sinatra'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end

get '/' do
  DAYNAMES = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"]
  dayname = DAYNAMES[Time.now.wday]
  "Hello, world! Happy #{dayname}."
end
```

```ruby
require 'sinatra'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end

get '/' do
  "Hello, world! Happy #{day_of_the_week(Time.now)}."
end
```

```ruby
require 'date'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end
```

```
require 'sinatra'

get '/' do
  "Hello, world! Happy #{day_of_the_week(Time.now)}—now from a file!"
end
```

```ruby
require 'sinatra'
require './day'

get '/' do
  "Hello, world! Happy #{day_of_the_week(Time.now)}—now from a file!"
end
```

```ruby
require 'date'

# Returns the day of the week for the given Time object.
def day_of_the_week(time)
  Date::DAYNAMES[time.wday]
end

# Returns a friendly greeting.
def greeting(time)
  # FILL IN
end
```

```
require 'sinatra'
require './day'

get '/' do
  greeting(Time.now)
end
```

```
>> "racecar".split("")
=> ["r", "a", "c", "e", "c", "a", "r"]
```

```
>> a = [ 17, 42, 8, 99 ]
>> a.reverse
=> [99, 8, 42, 17]
```

```
>> ["r", "a", "c", "e", "c", "a", "r"].join
=> "racecar"
```

```
>> "Racecar".split("").reverse.join
=> "racecaR"
```

```
>> "Racecar".chars.reverse.join
=> "racecaR"
```

```
>> "Racecar".reverse
=> "racecaR"
```

```
>> "Racecar".downcase.reverse
=> "racecar"
```

```ruby
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  string == string.reverse
end
```

```
>> palindrome?("racecar")
=> true
>> palindrome?("Racecar")
=> false
```

```ruby
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  string.downcase == string.downcase.reverse
end
```

```
>> load './palindrome.rb'
>> palindrome?("Racecar")
=> true
```

```ruby
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  processed_content = string.downcase
  processed_content == processed_content.reverse
end
```

```
>> load './palindrome.rb'
>> palindrome?("Racecar")
=> true
>> palindrome?("Able was I ere I saw Elba")
=> true
```

```
>> def email_parts(email)
?>   # FILL IN
>> end
```

```
>> (1..5).each { |i| puts 2**i }
2
4
8
16
32
```

```
>> (1..5).each do |i|
?>   puts 2**i
>> end
2
4
8
16
32
=> 1..5
```

```
get '/' do
  'hello, world!'
end
```

```ruby
a.each do |element|
  puts element
end
```

```
>> (1..5).each do |number|
?>   puts 2 ** number
>>   puts '--'
>> end
2
--
4
--
8
--
16
--
32
--
```

```
> ["ant", "bat", "cat", 42].forEach(function(element) {
  console.log(element);
});
ant
bat
cat
42
```

```
>> ["ant", "bat", "cat", 42].each do |element|
?>   puts element
>> end
ant
bat
cat
42
```

```
>> 3.times { puts "Betelgeuse!" }   # `times` takes a block with no variables.
"Betelgeuse!"
"Betelgeuse!"
"Betelgeuse!"
```

```ruby
def sandwich
  puts "top bread"
  yield
  puts "bottom bread"
end
```

```ruby
def sandwich
  puts "top bread"
  yield
  puts "bottom bread"
end

sandwich do
  puts "mutton, lettuce, and tomato"
end
```

```
$ ruby blocks.rb
top bread
mutton, lettuce, and tomato
bottom bread
```

```
.
.
.
def tag(tagname, text)
  html = "<#{tagname}>#{text}</#{tagname}>"
  yield html
end
```

```ruby
.
.
.
def tag(tagname, text)
  html = "<#{tagname}>#{text}</#{tagname}>"
  yield html
end

# Wrap some text in a paragraph tag.
```

```ruby
tag("p", "Lorem ipsum dolor sit amet") do |markup|
  puts markup
end
```

```
$ ruby blocks.rb
top bread
mutton, lettuce, and tomato
bottom bread
<p>Lorem ipsum dolor sit amet</p>
```

```
.
.
.
def bad_sandwich(contents)
  puts "top bread"
  contents
  puts "bottom bread"
end

bad_sandwich(puts "mutton, lettuce, and tomato")
```

"North Dakota" -> "north-dakota"

```ruby
states_map = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]

# urls: Imperative version
def imperative_urls(states)
  urls = []
  states.each do |state|
    urls << state.downcase.split.join("-")
  end
  urls
end
p imperative_urls(states)
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
```

```
>> [1, 2, 3, 4].map { |n| n*n }
=> [1, 4, 9, 16]
```

```
>> ["ALICE", "BOB", "CHARLIE"].map { |name| name.downcase }
=> ["alice", "bob", "charlie"]
```

```
>> ["ALICE", "BOB", "CHARLIE"].map(&:downcase)
=> ["alice", "bob", "charlie"]
```

```
>> states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
>> states.map { |state| state.downcase.split.join('-') }
=> ["kansas", "nebraska", "north-dakota", "south-dakota"]
```

```ruby
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]

# urls: Imperative version
def imperative_urls(states)
  urls = []
  states.each do |state|
    urls << state.downcase.split.join("-")
  end
```

```ruby
  urls
end
p imperative_urls(states)

# urls: Functional version
def functional_urls(states)
  states.map { |state| state.downcase.split.join('-') }
end
puts functional_urls(states).inspect
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
```

```ruby
# Returns a URL-friendly version of a string.
#   Example: "North Dakota" -> "north-dakota"
def urlify(string)
  string.downcase.split.join('-'))
end
```

```ruby
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]

# Returns a URL-friendly version of a string.
#   Example: "North Dakota" -> "north-dakota"
def urlify(string)
  string.downcase.split.join('-')
```

```ruby
end

# urls: Imperative version
def imperative_urls(states)
  urls = []
  states.each do |state|
    urls << urlify(state)
  end
  urls
end
puts imperative_urls(states).inspect

# urls: Functional version
def functional_urls(states)
  states.map { |state| urlify(state) }
end
puts functional_urls(states).inspect
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
```

```ruby
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
.
.
.
# singles: Imperative version
def imperative_singles(states)
  singles = []
  states.each do |state|
    if state.split.length == 1
      singles << state
    end
  end
  singles
end
puts imperative_singles(states).inspect
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
```

```
>> [1, 2, 3, 4, 5, 6, 7, 8].select { |n| n % 2 == 0}
=> [2, 4, 6, 8]
```

```
>> [1, 2, 3, 4, 5, 6, 7, 8].select { |n| n.even? }
=> [2, 4, 6, 8]
```

```
>> states.select { |state| state.split.length == 1 }
```

```ruby
states = ["Kansas", "Nebraska", "North Dakota", "South Dakota"]
.
.
.
# singles: Imperative version
def imperative_singles(states)
  singles = []
  states.each do |state|
    if (state.split.length == 1)
      singles << state
    end
  end
  singles
end
puts imperative_singles(states).inspect

# singles: Functional version
def functional_singles(states)
  states.select { |state| state.split.length == 1 }
end
puts functional_singles(states).inspect
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
["Kansas", "Nebraska"]
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
["Kansas", "Nebraska"]
55
```

```
>> numbers = 1..10
>> numbers.reduce(0) do |total, n|
?>  total += n
?>  total
?> end
55
```

```
>> numbers.reduce(0) { |total, n| total += n }
55
```

```
>> numbers.reduce { |total, n| total += n }
55
```

```
>> numbers.reduce { |total, n| total + n }
55
```

```ruby
.
.
.
numbers = 1..10

# sum: Imperative solution
def imperative_sum(numbers)
  total = 0
  numbers.each do |n|
    total += n
  end
  total
end
puts imperative_sum(numbers)

# sum: Functional solution
def functional_sum(numbers)
  numbers.reduce { |total, n| total + n }
end
puts functional_sum(numbers)
```

```
$ ruby functional.rb
["kansas", "nebraska", "north-dakota", "south-dakota"]
["kansas", "nebraska", "north-dakota", "south-dakota"]
["Kansas", "Nebraska"]
["Kansas", "Nebraska"]
55
55
```

```ruby
.
.
.
# lengths: Imperative version
def imperative_lengths(states)
  lengths = {}
  states.each do |state|
    lengths[state] = state.length
  end
  lengths
end
puts imperative_lengths(states)
```

```
$ ruby functional.rb
.
.
.
{"Kansas"=>6, "Nebraska"=>8, "North Dakota"=>12, "South Dakota"=>12}
```

```ruby
reduce({}) do |lengths, state|
  lengths[state] = state.length
  lengths
end
```

```ruby
.
.
.
# lengths: Imperative version
def imperative_lengths(states)
  lengths = {}
  states.each do |state|
    lengths[state] = state.length
  end
  lengths
end
puts imperative_lengths(states)

# lengths: Functional version
def functional_lengths(states)
  states.reduce({}) do |lengths, state|
    lengths[state] = state.length
    lengths
  end
end
puts functional_lengths(states)
```

```
states.reduce({}) { |lengths, state| lengths.merge({state => state.length}) }
```

```
$ ruby functional.rb
.
.
.
{"Kansas"=>6, "Nebraska"=>8, "North Dakota"=>12, "South Dakota"=>12}
{"Kansas"=>6, "Nebraska"=>8, "North Dakota"=>12, "South Dakota"=>12}
```

```ruby
.
.
.
# lengths: Functional version using `inject`
def functional_lengths(states)
  states.inject({}) do |lengths, state|
    lengths[state] = state.length
    lengths
  end
end
puts functional_lengths(states)
```

```
>> String.new("Madam, I'm Adam.")
=> "Madam, I'm Adam."
>> Time.new(1969, 7, 20, 20, 17, 40)
=> 1969-07-20 20:00:00 -0700
```

```ruby
# Defines a Phrase class.
class Phrase
end

phrase = Phrase.new
puts phrase
```

```
$ ruby palindrome.rb
#<Phrase:0x00007fa3d30a3e98>
```

```ruby
# Defines a Phrase class.
class Phrase

  def initialize(content)
    @content = content
  end
end

phrase = Phrase.new("Madam, I'm Adam.")
puts phrase.content
```

```
$ ruby palindrome.rb
Traceback (most recent call last):
palindrome.rb:15:in `<main>': undefined method `content' for
#<Phrase:0x00007fe8c70a3db0 @content="Madam, I'm Adam."> (NoMethodError)
```

```ruby
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
   @content = content
  end
end

phrase = Phrase.new("Madam, I'm Adam.")
puts phrase.content
```

```ruby
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
   @content = content
  end
end

phrase = Phrase.new("Madam, I'm Adam.")
puts phrase.content

phrase.content = "Able was I, ere I saw Elba."
puts phrase.content
```

```
$ ruby palindrome.rb
Madam, I'm Adam.
Able was I, ere I saw Elba.
```

```ruby
# Returns true for a palindrome, false otherwise.
def palindrome?(string)
  processed_content = string.downcase
  processed_content == processed_content.reverse
end

# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end
end
```

```
>> load "./palindrome.rb"
>> phrase = Phrase.new("Racecar")
>> phrase.content
=> "Racecar"
>> palindrome?(phrase.content)
=> true
```

```ruby
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.content.downcase
    processed_content == processed_content.reverse
  end
end
```

```
phrase = Phrase.new("Racecar")
```

```
>> load "./palindrome.rb"
>> phrase = Phrase.new("Racecar")
>> phrase.palindrome?
=> true
```

```ruby
# Defines a Phrase class.
class Phrase
  attr_accessor :content

  def initialize(content)
    @content = content
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.content.downcase
    processed_content == processed_content.reverse
  end

  # Makes the phrase LOUDER.
  def louder
    # FILL IN
  end
end
```

```
>> load "./palindrome.rb"
>> p = Phrase.new("yo adrian!")
>> p.louder
=> "YO ADRIAN!"
```

```
>> s = String.new("foobar")
=> "foobar"
>> s.class                        # Find the class of s.
=> String
>> s.class.superclass             # Find the superclass of String.
=> Object
>> s.class.superclass.superclass  # Ruby has a BasicObject base class as of 1.9
=> BasicObject
>> s.class.superclass.superclass.superclass
=> nil
```

```
>> "honey badger".class
=> String
```

```ruby
# Defines a Phrase class.
class Phrase < String
  attr_accessor :content

  def initialize(content)
    @content = content
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.downcase
    processed_content == processed_content.reverse
  end
end
```

```ruby
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = self.downcase
    processed_content == processed_content.reverse
  end
end
```

```
>> load "./palindrome.rb"
>> phrase = Phrase.new("Racecar")
>> phrase.palindrome?
=> true
```

```
>> phrase.class
=> Phrase
>> phrase.class.superclass
=> String
>> phrase.class.superclass.superclass
=> Object
>> phrase.class.superclass.superclass.superclass
=> BasicObject
```

```ruby
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content = downcase
    processed_content == processed_content.reverse
  end
end
```

```ruby
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end
```

```ruby
# Defines a translated Phrase.
class TranslatedPhrase < Phrase

end
```

```
TranslatedPhrase.new("recognize", "reconocer")
```

```ruby
# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    @translation = translation
  end
end
```

```ruby
# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end
end
```

```ruby
# Defines a Phrase class (inheriting from String).
class Phrase < String
  .
  .
  .
end

# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end
end
```

```
>> load "./palindrome.rb"
>> frase = TranslatedPhrase.new("recognize", "reconocer")
>> frase.palindrome?
=> false
```

```ruby
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end

# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end

  # Processes the translation for palindrome testing.
  def processed_content
    self.translation.downcase
  end
end
```

```
>> load "./palindrome.rb"
>> frase = TranslatedPhrase.new("recognize", "reconocer")
>> frase.palindrome?
=> true
```

```ruby
# Defines a Phrase class (inheriting from String).
class Phrase < String

  # Processes the string for palindrome testing.
  def processor(string)
    # FILL IN
  end
```

```ruby
  # Returns content for palindrome testing.
  def processed_content
    processor(self)
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end

# Defines a translated Phrase.
class TranslatedPhrase < Phrase
  attr_accessor :translation

  def initialize(content, translation)
    super(content)
    @translation = translation
  end

  # Processes the translation for palindrome testing.
  def processed_content
    processor(translation)
  end
end
```

```ruby
class Phrase < String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end
```

```ruby
class String

  # Returns content for palindrome testing.
  def processed_content
    self.downcase
  end

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
end
```

```
>> load "./palindrome.rb"
>> napoleonsLament = String.new("Able was I ere I saw Elba")
>> napoleonsLament.palindrome?
=> true
```

```
>> "foobar".palindrome?
=> false
>> "Racecar".palindrome?
=> true
>> "Able was I ere I saw Elba".palindrome?
=> true
```

```ruby
class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
```

```ruby
class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

```ruby
class Integer

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.to_s
    end
end
```

```ruby
module Palindrome

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.to_s.downcase
    end
end
```

```ruby
module Palindrome

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end
  private

    # Returns content for palindrome testing.
    def processed_content
      self.to_s.downcase
    end
end

class String
  include Palindrome
end

class Integer
  include Palindrome
end
```

```
>> load "./palindrome.rb"
>> "Racecar".palindrome?
=> true
>> 12321.palindrome?
=> true
```

```
$ cd ~/repos     # Use ~/environment/repos on Cloud9
$ bundle gem <username>_palindrome
Do you want to generate tests with your gem?
Type 'rspec' or 'minitest' to generate those test files now and in the future.
rspec/minitest/(none): minitest
Do you want to license your code permissively under the MIT license? y/(n): n
Do you want to include a code of conduct in gems you generate? y/(n): n
```

```
$ cd <username>_palindrome
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

```ruby
lib = File.expand_path("../lib", __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require "mhartl_palindrome/version"

Gem::Specification.new do |spec|
  spec.name          = "mhartl_palindrome"
  spec.version       = MhartlPalindrome::VERSION
  spec.authors       = ["Michael Hartl"]
  spec.email         = ["michael@michaelhartl.com"]

  spec.summary       = %q{Palindrome detector}
  spec.description   = %q{Learn Enough Ruby palindrome detector}
  spec.homepage      = "https://github.com/mhartl/mhartl_palindrome"
  spec.license       = "MIT"

  # Prevent pushing this gem to RubyGems.org. To allow pushes either set the
  # 'allowed_push_host'
  # to allow pushing to a single host or delete this section to allow pushing to
  # any host.
  if spec.respond_to?(:metadata)
    spec.metadata["allowed_push_host"] = "https://rubygems.org/"
  else
    raise "RubyGems 2.0 or newer is required to protect against " \
      "public gem pushes."
  end

  # Specify which files should be added to the gem when it is released.
  # The `git ls-files -z` loads the files in the RubyGem
  # that have been added into git.
  spec.files         = Dir.chdir(File.expand_path('..', __FILE__)) do
    `git ls-files -z`.split("\x0").reject do
      |f| f.match(%r{^(test|spec|features)/})
    end
  end
  spec.bindir        = "exe"
  spec.executables   = spec.files.grep(%r{^exe/}) { |f| File.basename(f) }
  spec.require_paths = ["lib"]

  spec.add_development_dependency "bundler", "~> 1.16"
  spec.add_development_dependency "rake", "~> 10.0"
  spec.add_development_dependency "minitest", "~> 5.0"
end
```

```ruby
source "https://rubygems.org"

git_source(:github) {|repo_name| "https://github.com/#{repo_name}" }

# Specify your gem's dependencies in <username>_palindrome.gemspec
gemspec

gem 'minitest-reporters', '1.2.0'
```

```
$LOAD_PATH.unshift File.expand_path("../../lib", __FILE__)
require "mhartl_palindrome"
require "minitest/autorun"
require "minitest/reporters"
Minitest::Reporters.use!
```

```
$ bundle exec rake test
Started with run options --seed 57409

 FAIL["test_it_does_something_useful", "MhartlPalindromeTest",
 0.0007609999738633633]
 test_it_does_something_useful#MhartlPalindromeTest (0.00s)
        Expected false to be truthy.
        /Users/mhartl/repos/mhartl_palindrome/test/mhartl_palindrome_test.rb:9:in
        `test_it_does_something_useful'

  2/2: [==================================] 100% Time: 00:00:00, Time: 00:00:00

Finished in 0.00119s
2 tests, 2 assertions, 1 failures, 0 errors, 0 skips
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test
  def test_that_it_has_a_version_number
    refute_nil ::MhartlPalindrome::VERSION
  end

  def test_it_does_something_useful
    assert false
  end
end
```

```
$ bundle exec rake test
2 tests, 2 assertions, 1 failures, 0 errors, 0 skips
```

```
require "test_helper"

class Mhartl::PalindromeTest < Minitest::Test
  def test_that_it_has_a_version_number
    refute_nil ::Mhartl::Palindrome::VERSION
  end

  def test_it_does_something_useful
    assert true
  end
end
```

```
$ bundle exec rake test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
require "mhartl_palindrome/version"

module MhartlPalindrome
  # Your code goes here...
end
```

```ruby
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

```ruby
def test_non_palindrome
  assert !"apple".palindrome?
end
```

```ruby
def test_literal_palindrome
  assert "racecar".palindrome?
end
```

```
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    assert !"apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end
end
```

```
$ bundle exec rake test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    assert !"apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    skip
  end

  def test_palindrome_with_punctuation
    skip
  end
end
```

```
$ bundle exec rake test
4 tests, 2 assertions, 0 failures, 0 errors, 2 skips
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    assert !"apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    # FILL IN
  end

  def test_palindrome_with_punctuation
```

```
    skip
  end
end
```

```ruby
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self
    end
end
```

```ruby
def test_palindrome_with_punctuation
  assert "Madam, I'm Adam.".palindrome?
end
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end
end
```

```
$ bundle exec rake test
4 tests, 4 assertions, 1 failures, 0 errors, 0 skips
```

```
assert "Madam, I'm Adam.".letters == "MadamImAdam"
```

```
assert_equal <expected>, <actual>
```

```
assert_equal "MadamImAdam", "Madam, I'm Adam.".letters
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end

  def test_letters
    assert_equal "MadamImAdam", "Madam, I'm Adam.".letters
  end
end
```

```
$ bundle exec rake test
NoMethodError: undefined method `letters' for "Madam, I'm Adam.":String
5 tests, 4 assertions, 0 failures, 1 errors, 0 skips
```

```
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

```
$ bundle exec rake test
 FAIL["test_letters", "MhartlPalindromeTest", 0.0007690000347793102]
 test_letters#MhartlPalindromeTest (0.00s)
        Expected: "MadamImAdam"
          Actual: nil
5 tests, 5 assertions, 2 failures, 0 errors, 0 skips
```

```
$ irb
>> "M".match?(/[a-zA-Z]/)
=> true
>> "d".match?(/[a-zA-Z]/)
=> true
>> ",".match?(/[a-zA-Z]/)
=> false
```

```
the_letters = []
for i in 0..self.length - 1
  if (self[i].match(/[a-zA-Z]/))
    the_letters << self[i]
  end
end
```

```ruby
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    the_letters = []
    for i in 0..self.length - 1
      if self[i].match(/[a-zA-Z]/)
        the_letters << self[i]
      end
    end
    the_letters.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.downcase
    end
end
```

```
$ bundle exec rake test
5 tests, 5 assertions, 1 failures, 0 errors, 0 skips
```

```ruby
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    the_letters = []
    for i in 0..self.length - 1
      if self[i].match(/[a-zA-Z]/)
        the_letters << self[i]
      end
    end
    the_letters.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.letters.downcase
    end
end
```

```
$ bundle exec rake test
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

```
def letters
  the_letters = []
  for i in 0..self.length - 1
    character = self[i]
    if character.match(/[a-zA-Z]/)
      the_letters << character
    end
```

```
  end
  the_letters.join
end
```

```
letter_regex = /[a-z]/i
for i in 0..self.length - 1
  character = self[i]
  if character.match(letter_regex)
    the_letters << character
  end
end
```

```
letter_regex = /[a-z]/i
self.chars.each do |character|
  if character.match(letter_regex)
    the_letters << character
  end
end
```

```
require "mhartl_palindrome/version"

class String
```

```ruby
  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    the_letters = []
    letter_regex = /[a-z]/i
    self.chars.each do |character|
      if character.match(letter_regex)
        the_letters << character
      end
    end
    the_letters.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.letters.downcase
    end
end
```

```
$ bundle exec rake test
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

```
>> "Madam, I'm Adam.".chars
=> ["M", "a", "d", "a", "m", ",", " ", "I", "'", "m", " ",
    "A", "d", "a", "m", "."]
>> "Madam, I'm Adam".chars.select { |c| c.match(/[a-z]/i) }
=> ["M", "a", "d", "a", "m", "I", "m", "A", "d", "a", "m"]
>> "Madam, I'm Adam".chars.select { |c| c.match(/[a-z]/i) }.join
=> "MadamImAdam"
```

```ruby
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  # Returns the letters in the string.
  def letters
    self.chars.select { |c| c.match(/[a-z]/i) }.join
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.letters.downcase
    end
end
```

```
$ bundle exec rake test
5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end
end
```

```
>> "Madam, I'm Adam.".scan(/[a-z]/i)
=> ["M", "a", "d", "a", "m", "I", "m", "A", "d", "a", "m"]
>> "Madam, I'm Adam.".scan(/[a-z]/i).join
=> "MadamImAdam"
```

```ruby
require "mhartl_palindrome/version"

class String

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      self.scan(/[a-z]/i).join.downcase
    end
end
```

```
require "mhartl_palindrome/version"

class String
```

```ruby
  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      scan(/[a-z]/i).join.downcase
    end
end
```

```
$ bundle exec rake test
4 tests, 4 assertions, 0 failures, 0 errors, 0 skips
```

```
$ git add -A
$ git commit -m "Finish working and refactored palindrome method"
$ git push
```

```
$ bundle exec rake install
$ irb
>> require '<username>_palindrome'
true
>> "Madam, I'm Adam.".palindrome?
true
```

```
$ bundle exec rake release
mhartl_palindrome 0.1.0 built to pkg/mhartl_palindrome-0.1.0.gem.
Tagged v0.1.0.
Pushed git commits and tags.
Pushed mhartl_palindrome 0.1.0 to rubygems.org
```

```
$ gem uninstall <username>_palindrome
Successfully uninstalled <username>_palindrome-0.1.0
$ gem install <username>_palindrome -v 0.1.0
Successfully installed <username>_palindrome-0.1.0
1 gem installed
$ irb
>> require '<username>_palindrome'
true
>> "RaceCar".palindrome?
true
```

```ruby
require "mhartl_palindrome/version"

module MhartlPalindrome

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    processed_content == processed_content.reverse
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      scan(/[a-z]/i).join.downcase
    end
end

class String
  include MhartlPalindrome
end
```

```ruby
require "test_helper"

class MhartlPalindromeTest < Minitest::Test

  def test_non_palindrome
    refute "apple".palindrome?
  end

  def test_literal_palindrome
    assert "racecar".palindrome?
  end

  def test_mixed_case_palindrome
    assert "RaceCar".palindrome?
  end

  def test_palindrome_with_punctuation
    assert "Madam, I'm Adam.".palindrome?
  end
```

```ruby
  def test_integer_non_palindrome
    FILL_IN 12345.palindrome?
  end

  def test_integer_palindrome
    FILL_IN 12321.palindrome?
  end
end
```

```
require "mhartl_palindrome/version"

module MhartlPalindrome

  # Returns true for a palindrome, false otherwise.
  def palindrome?
    if processed_content.empty?
      false
    else
      processed_content == processed_content.reverse
    end
  end

  private

    # Returns content for palindrome testing.
    def processed_content
      to_s.scan(/[a-zFILL_IN]/i).join.downcase
    end
end

class String
  include MhartlPalindrome
end

class Integer
  include FILL_IN
end
```

```
$ cd ~/repos/ruby_tutorial/
$ curl -OL https://cdn.learnenough.com/phrases.txt
```

```
>> text = File.read('phrases.txt')
<lots of output>
```

```
>> s = 'supercalifragilisticexpialidocious'; 1 + 2
=> 3
```

```
>> text = File.read('phrases.txt'); 0
=> 0
```

```
>> text.length
=> 1373
>> text.split("\n")[0]      # Split on newlines and extract the 1st phrase.
=> "A butt tuba"
```

```ruby
#!/usr/bin/env ruby
require '<username>_palindrome'

puts "hello, world!"
```

```ruby
#!/usr/bin/env ruby
require '<username>_palindrome'

text = File.read("phrases.txt")
text.split("\n").each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end
```

```
$ ./palindrome_file
.
.
.
palindrome detected: Dennis sinned.
palindrome detected: Dennis and Edna sinned.
palindrome detected: Dennis, Nell, Edna, Leon, Nedra, Anita, Rolf, Nora,
Alice, Carol, Leo, Jane, Reed, Dena, Dale, Basil, Rae, Penny, Lana, Dave,
Denny, Lena, Ida, Bernadette, Ben, Ray, Lila, Nina, Jo, Ira, Mara, Sara,
Mario, Jan, Ina, Lily, Arne, Bette, Dan, Reba, Diane, Lynn, Ed, Eva, Dana,
Lynne, Pearl, Isabel, Ada, Ned, Dee, Rena, Joel, Lora, Cecil, Aaron, Flora,
Tina, Arden, Noel, and Ellen sinned.
palindrome detected: Go hang a salami, I'm a lasagna hog.
palindrome detected: level
palindrome detected: Madam, I'm Adam.
palindrome detected: No "x" in "Nixon"
palindrome detected: No devil lived on
palindrome detected: Race fast, safe car
palindrome detected: racecar
palindrome detected: radar
palindrome detected: Was it a bar or a bat I saw?
palindrome detected: Was it a car or a cat I saw?
palindrome detected: Was it a cat I saw?
palindrome detected: Yo, banana boy!
```

```ruby
#!/usr/bin/env ruby
require '<username>_palindrome'

lines = File.readlines("phrases.txt")
lines.each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end
```

```
File.write(filename, content_string)
```

```
palindromes = lines.select { |line| line.palindrome? }
```

```
palindromes = lines.select(&:palindrome?)
```

```ruby
#!/usr/bin/env ruby
require '<username>_palindrome'

lines = File.readlines("phrases.txt")
lines.each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end

palindromes = lines.select(&:palindrome?)
File.write('palindromes_file.txt', palindromes.join)
```

```ruby
#!/usr/bin/env ruby
require '<username>_palindrome'

palindromes = File.readlines('phrases.txt').select(&:palindrome?)
palindromes.each { |palindrome| puts "palindrome detected: #{palindrome}" }
File.write('palindromes_file.txt', palindromes.join)
```

```ruby
#!/usr/bin/env ruby
require '<username>_palindrome'
require 'open-uri'

lines = URI.open('https://cdn.learnenough.com/phrases.txt').readlines
lines.each do |line|
  if line.palindrome?
    puts "palindrome detected: #{line}"
  end
end
```

```
$ ./palindrome_url
.
.
.
palindrome detected: Dennis sinned.
palindrome detected: Dennis and Edna sinned.
palindrome detected: Dennis, Nell, Edna, Leon, Nedra, Anita, Rolf, Nora,
Alice, Carol, Leo, Jane, Reed, Dena, Dale, Basil, Rae, Penny, Lana, Dave,
Denny, Lena, Ida, Bernadette, Ben, Ray, Lila, Nina, Jo, Ira, Mara, Sara,
Mario, Jan, Ina, Lily, Arne, Bette, Dan, Reba, Diane, Lynn, Ed, Eva, Dana,
Lynne, Pearl, Isabel, Ada, Ned, Dee, Rena, Joel, Lora, Cecil, Aaron, Flora,
Tina, Arden, Noel, and Ellen sinned.
palindrome detected: Go hang a salami, I'm a lasagna hog.
palindrome detected: level
palindrome detected: Madam, I'm Adam.
palindrome detected: No "x" in "Nixon"
palindrome detected: No devil lived on
palindrome detected: Race fast, safe car
palindrome detected: racecar
palindrome detected: radar
palindrome detected: Was it a bar or a bat I saw?
palindrome detected: Was it a car or a cat I saw?
palindrome detected: Was it a cat I saw?
palindrome detected: Yo, banana boy!
```

```
$ gem install nokogiri -v 1.8.5
Building native extensions. This could take a while...
```

```
>> require 'nokogiri'
>> html = '<p>lorem<sup class="reference">1</sup></p><p>ipsum</p>'
>> doc = Nokogiri::HTML(html)
=> #<Nokogiri::HTML::Document:0x3fd87e023b18...
```

```
>> doc.css('p')
=> [#<Nokogiri::XML::Element:0x3fd87e022664 name="p"...
>> doc.css('p').length
=> 2
>> doc.css('p')[0].content
=> "lorem1"
```

```
>> doc.css('.reference')
=> [#<Nokogiri::XML::Element:0x3fd87e04d60c name="sup"...
>> doc.css('.reference').length
=> 1
```

```
>> doc.css('.reference').each { |reference| reference.remove }
```

```
>> doc.css('p').map { |paragraph| paragraph.content }
=> ["lorem", "ipsum"]
```

```ruby
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]

puts url
```

```
$ ./wikp https://es.wikipedia.org/wiki/Ruby
https://es.wikipedia.org/wiki/Ruby
```

```ruby
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
```

```ruby
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).
```

```ruby
url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
doc.css('.reference').each { |reference| reference.remove }
```

```ruby
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
doc.css('.reference').each { |reference| reference.remove }
content_array = doc.css('p').map { |paragraph| paragraph.content }
puts content_array.join("\n")
```

```
$ ./wikp https://es.wikipedia.org/wiki/Ruby
Ruby es un lenguaje de programación interpretado, reflexivo y orientado a
objetos, creado por el programador japonés Yukihiro "Matz" Matsumoto, quien
comenzó a trabajar en Ruby en 1993, y lo presentó públicamente en 1995.
.
.
.
A partir de la versión 1.9.3 se opta por una licencia dual bajo las licencias
BSD de dos cláusulas y Licencia pública Ruby.
```

```
$ ./wikp https://es.wikipedia.org/wiki/Ruby | pbcopy
```

```ruby
#!/usr/bin/env ruby
require 'open-uri'
require 'nokogiri'

# Returns the paragraphs from a Wikipedia link, stripped of reference numbers.
# Especially useful for text-to-speech (both native and foreign).

url = ARGV[0]
doc = Nokogiri::HTML(URI.open(url).read)
doc.css('.reference').each(&:remove)
puts doc.css('p').map(&:content).join("\n")
```

```
$ cd ~/repos                # cd ~/environments/repos on the cloud IDE
$ mkdir palindrome_app
$ cd palindrome_app/
```

```
source 'https://rubygems.org'

ruby '3.1.1'   # Change this line if you're using a different Ruby version.

gem 'sinatra', '2.2.0'
gem 'puma',    '5.6.4'
gem 'rerun',   '0.13.1'
```

```
$ bundle exec rerun app.rb

12:10:10 [rerun] Palindrome_app launched
12:10:10 [rerun] Rerun (79556) running Palindrome_app (79575)
== Sinatra has taken the stage on 4567 for development
Listening on localhost:4567, CTRL+C to stop
```

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

```
$ bundle _2.3.10_ lock --add-platform x86_64-linux
$ git add -A
$ git commit -m "Add a config file"
$ heroku create
$ git push heroku main
```

```
$ mkdir views
$ cd views
$ touch index.erb about.erb palindrome.erb
$ cd ..
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">

<h1>Sample Sinatra App</h1>
```

```
<p>
  This is the sample Sinatra app for
  <a href="https://www.learnenough.com/ruby-tutorial"><em>Learn Enough Ruby
  to Be Dangerous</em></a>. Learn more on the <a href="/about">About</a> page.
</p>

<p>
  Click the <a href="https://en.wikipedia.org/wiki/Sator_Square">Sator
  Square</a> below to run the custom <a href="/palindrome">Palindrome
  Detector</a>.
</p>

<a class="sator-square" href="/palindrome">
  <img src="/images/sator_square.jpg" alt="Sator Square">
</a>
      </div>
    </div>
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">

<h1>About</h1>

<p>
  This site is the final application in
  <a href="https://www.learnenough.com/ruby-tutorial"><em>Learn Enough Ruby
  to Be Dangerous</em></a>
  by <a href="https://www.michaelhartl.com/">Michael Hartl</a>,
  a tutorial introduction to the
  <a href="https://www.ruby-lang.org/en/">Ruby programming language</a> that
  is part of
```

```
  <a href="https://www.learnenough.com/">LearnEnough.com</a>.
</p>

<p>
  <em>Learn Enough Ruby to Be Dangerous</em> is a natural prerequisite to
  the <a href="https://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a>, a book and video series that is one of the leading
  introductions to web development. <em>Learn Enough Ruby</em> is also an
  excellent choice <em>after</em> the <em>Rails Tutorial</em> for those who
  prefer to start with the latter first.
</p>

      </div>
    </div>
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">

<h1>Palindrome Detector</h1>

<p>This will be the palindrome detector.</p>

      </div>
    </div>
  </body>
</html>
```

```
$ curl -OL https://cdn.learnenough.com/le_ruby_palindrome_public.tar.gz
```

```
$ tar zxvf le_ruby_palindrome_public.tar.gz
x public/
x public/images/
x public/stylesheets/
x public/stylesheets/main.css
x public/images/sator_square.jpg
x public/images/logo_b.png
$ rm -f le_ruby_palindrome_public.tar.gz
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">
```

```
        <!-- page-specific content -->
      </div>
    </div>
  </body>
</html>
```

```ruby
source 'https://rubygems.org'

ruby '3.1.1'   # Change this line if you're using a different Ruby version.

gem 'sinatra', '2.2.0'
gem 'puma',    '5.6.4'
gem 'rerun',   '0.13.1'

group :test do
  gem 'minitest',           '5.15.0'
  gem 'minitest-reporters', '1.5.0'
  gem 'rack-test',          '1.1.0'
```

```
  gem 'rake',                 '13.0.6'
  gem 'nokogiri',             '1.13.3'
end
```

```
$ mkdir test
$ touch test/test_helper.rb test/site_pages_test.rb
```

```ruby
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index
    get '/'
    assert last_response.ok?
  end

  def test_about
    get '/about'
    assert last_response.ok?
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
  end
end
```

```
$ bundle exec rake test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

```
$ bundle exec rake
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
ENV['RACK_ENV'] = 'test'

require_relative '../app'
require 'rack/test'
require 'nokogiri'
require 'minitest/autorun'
require 'minitest/reporters'
Minitest::Reporters.use!

# Returns the document.
def doc(response)
  Nokogiri::HTML(response.body)
end
```

```
doc(last_response).css('h1').first
```

```
doc(last_response).at_css('h1')
```

```
assert doc(last_response).at_css('h1')
```

```ruby
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end
```

```
  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
  end
end
```

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

```
<h1>Sample Sinatra App</h1>

<p>
  This is the sample Sinatra app for
  <a href="https://www.learnenough.com/ruby-tutorial"><em>Learn Enough Ruby
  to Be Dangerous</em></a>. Learn more on the <a href="/about">About</a> page.
</p>

<p>
  Click the <a href="https://en.wikipedia.org/wiki/Sator_Square">Sator
  Square</a> below to run the custom <a href="/palindrome">Palindrome
  Detector</a>.
</p>

<a class="sator-square" href="/palindrome">
  <img src="/images/sator_square.jpg" alt="Sator Square">
</a>
```

```
<h1>About</h1>

<p>
  This site is the final application in
  <a href="https://www.learnenough.com/ruby-tutorial"><em>Learn Enough Ruby
  to Be Dangerous</em></a>
  by <a href="https://www.michaelhartl.com/">Michael Hartl</a>,
  a tutorial introduction to the
  <a href="https://www.ruby-lang.org/en/">Ruby programming language</a> that
  is part of
  <a href="https://www.learnenough.com/">LearnEnough.com</a>.
</p>

<p>
  <em>Learn Enough Ruby to Be Dangerous</em> is a natural prerequisite to
  the <a href="https://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a>, a book and video series that is one of the leading
  introductions to web development. <em>Learn Enough Ruby</em> is also an
  excellent choice <em>after</em> the <em>Rails Tutorial</em> for those who
  prefer to start with the latter first.
</p>
```

```
<h1>Palindrome Detector</h1>

<p>This will be the palindrome detector.</p>
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">
        <!-- page-specific content -->
      </div>
    </div>
  </body>
</html>
```

```
$ bundle exec rake test
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App</title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
require 'sinatra'

get '/' do
  erb :index, :layout => :page
end

get '/about' do
  erb :about, :layout => :page
end

get '/palindrome' do
  erb :palindrome, :layout => :page
end
```

```
<title>Learn Enough Ruby Sample App | Home</title>
```

```
<title>Learn Enough Ruby Sample App | About</title>
```

```
<title>Learn Enough Ruby Sample App | Palindrome Detector</title>
```

```
doc(last_response).at_css('title')
```

```
title_content = doc(last_response).at_css('title').content
```

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App", title_content
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App", title_content
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App", title_content
  end
end
```

```
$ bundle exec rake test
3 tests, 9 assertions, 0 failures, 0 errors, 0 skips
```

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | Home", title_content
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | About", title_content
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | Palindrome Detector",
                 title_content
  end
end
```

```
$ bundle exec rake test
3 tests, 9 assertions, 3 failures, 0 errors, 0 skips
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    .
    .
    .
```

```
$ bundle exec rake test
3 tests, 9 assertions, 0 failures, 0 errors, 0 skips
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <header class="header">
        <nav>
          <ul class="header-nav">
            <li><a href="/">Home</a></li>
            <li><a href="/palindrome">Is It a Palindrome?</a></li>
            <li><a href="/about">About</a></li>
          </ul>
        </nav>
      </header>
      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

```
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "Learn Enough Ruby Sample App | Home", title_content
    assert doc(last_response).at_css('nav')
  end
  .
  .
  .
end
```

```
$ bundle exec rake test
3 tests, 10 assertions, 0 failures, 0 errors, 0 skips
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">

      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

```
<header class="header">
  <nav>
    <ul class="header-nav">
      <li><a href="/">Home</a></li>
      <li><a href="/palindrome">Is It a Palindrome?</a></li>
      <li><a href="/about">About</a></li>
    </ul>
  </nav>
</header>
```

```
$ bundle exec rake test
3 tests, 10 assertions, 1 failures, 0 errors, 0 skips
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Learn Enough Ruby Sample App | <%= @title %></title>
    <link rel="stylesheet" type="text/css" href="/stylesheets/main.css">
    <link href="https://fonts.googleapis.com/css?family=Open+Sans:300,400"
          rel="stylesheet">
  </head>
  <body>
    <a href="/" class="header-logo">
      <img src="/images/logo_b.png" alt="Learn Enough logo">
    </a>
    <div class="container">
      <%= erb :navigation %>
      <div class="content">
        <%= yield %>
      </div>
    </div>
  </body>
</html>
```

```
$ bundle exec rake test
3 tests, 10 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def setup
    @base_title = "Learn Enough Ruby Sample App"
  end

  def test_index
    get '/'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "#{@base_title} | Home", title_content
  end

  def test_about
    get '/about'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "#{@base_title} | About", title_content
```

```
  end

  def test_palindrome
    get '/palindrome'
    assert last_response.ok?
    assert doc(last_response).at_css('h1')
    title_content = doc(last_response).at_css('title').content
    assert_equal "#{@base_title} | Palindrome Detector", title_content
  end
end
```

```
source 'https://rubygems.org'

ruby '3.1.1'   # Change this line if you're using a different Ruby version.

gem 'sinatra',           '2.2.0'
gem 'puma',              '5.6.4'
gem 'rerun',             '0.13.1'
gem 'mhartl_palindrome', '0.1.0'

group :test do
  gem 'minitest',          '5.15.0'
  gem 'minitest-reporters', '1.5.0'
  gem 'rack-test',         '1.1.0'
  gem 'rake',              '13.0.6'
  gem 'nokogiri',          '1.13.3'
end
```

```
require 'sinatra'
require 'mhartl_palindrome'

get '/' do
  @title = 'Home'
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end
```

```
<button class="form-submit" type="submit">Is it a palindrome?</button>
```

```
<textarea name="phrase" rows="10" cols="60"></textarea>
```

```
<form id="palindrome_tester" action="/check" method="post">
```

```
<h1>Palindrome Detector</h1>

<form id="palindrome_tester" action="/check" method="post">
  <textarea name="phrase" rows="10" cols="60"></textarea>
  <br>
  <button class="form-submit" type="submit">Is it a palindrome?</button>
</form>
```

```
<form id="palindromeTester">
  <textarea name="phrase" rows="10" cols="30"></textarea>
  <br>
  <button type="submit">Is it a palindrome?</button>
</form>
```

```ruby
post '/check' do
  # Do something to handle the submission
end
```

```
require 'sinatra'
require 'mhartl_palindrome'
```

```ruby
get '/' do
  @title = 'Home'
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end

post '/check' do
  raise params.inspect
end
```

```
{ "phrase" => "Madam, I'm Adam." }
```

```ruby
if @phrase.palindrome?
  puts "\"#{@phrase}\" is a palindrome!"
else
  puts "\"#{@phrase}\" isn't a palindrome."
end
```

```
<% if @phrase.palindrome? %>
   "<%= @phrase %>" is a palindrome!
<% else %>
    "<%= @phrase %>" isn't a palindrome!
<% end %>
```

```
<h1>Palindrome Result</h1>

<% if @phrase.palindrome? %>
  <div class="result result-success">
    <p>"<%= @phrase %>" is a palindrome!</p>
  </div>
```

```
<% else %>
  <div class="result result-fail">
    <p>"<%= @phrase %>" isn't a palindrome!</p>
  </div>
<% end %>
```

```
require 'sinatra'
require 'mhartl_palindrome'

get '/' do
  @title = 'Home'
  erb :index
end

get '/about' do
  @title = 'About'
  erb :about
end

get '/palindrome' do
  @title = 'Palindrome Detector'
  erb :palindrome
end

post '/check' do
  @phrase = params[:phrase]
  erb :result
end
```

```
$ touch test/palindrome_test.rb
```

```ruby
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_form_presence
    get '/palindrome'
    assert doc(last_response).at_css('form')
  end
end
```

```
post '/check', phrase: "Not a palindrome"
```

```
assert_includes result, substring
```

```
assert result.include?(substring)
```

```
assert_includes doc(last_response).at_css('p').content, "isn't a palindrome"
```

```ruby
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_form_presence
    get '/palindrome'
    assert doc(last_response).at_css('form')
  end

  def test_non_palindrome_submission
    post '/check', phrase: "Not a palindrome"
    assert_includes doc(last_response).at_css('p').content, "isn't a palindrome"
  end

  def test_palindrome_submission
    post '/check', phrase: "Able was I, ere I saw Elba."
    assert_includes doc(last_response).at_css('p').content, "is a palindrome"
  end
end
```

```
$ bundle exec rake test
6 tests, 15 assertions, 0 failures, 0 errors, 0 skips
```

```ruby
require_relative 'test_helper'

class PalindromeAppTest < Minitest::Test
  include Rack::Test::Methods

  def app
    Sinatra::Application
  end

  def test_form_presence
    get '/palindrome'
    assert doc(last_response).at_css('form')
  end

  def test_non_palindrome_submission
    post '/check', phrase: "Not a palindrome"
    assert_includes doc(last_response).at_css('p').content, "isn't a palindrome"
    assert doc(last_response).at_css('form')
  end

  def test_palindrome_submission
    post '/check', phrase: "Able was I, ere I saw Elba."
    assert_includes doc(last_response).at_css('p').content, "is a palindrome"
  end
end
```

```
$ bundle exec rake test
6 tests, 16 assertions, 1 failures, 0 errors, 0 skips
```

```
<h1>Palindrome Result</h1>

<% if @phrase.palindrome? %>
  <div class="result result-success">
    <p>"<%= @phrase %>" is a palindrome!</p>
  </div>
<% else %>
  <div class="result result-fail">
    <p>"<%= @phrase %>" isn't a palindrome!</p>
  </div>
<% end %>

<h2>Try another one!</h2>

<form id="palindrome_tester" action="/check" method="post">
  <textarea name="phrase" rows="10" cols="60"></textarea>
  <br>
  <button class="form-submit" type="submit">Is it a palindrome?</button>
</form>
```

```
$ bundle exec rake test
6 tests, 16 assertions, 0 failures, 0 errors, 0 skips
```

```
$ touch views/palindrome_form.erb
```

```
<form id="palindrome_tester" action="/check" method="post">
  <textarea name="phrase" rows="10" cols="60"></textarea>
  <br>
  <button class="form-submit" type="submit">Is it a palindrome?</button>
</form>
```

```
<h1>Palindrome Result</h1>

<% if @phrase.palindrome? %>
  <div class="result result-success">
    <p>"<%= @phrase %>" is a palindrome!</p>
  </div>
<% else %>
  <div class="result result-fail">
    <p>"<%= @phrase %>" isn't a palindrome!</p>
  </div>
<% end %>

<h2>Try another one!</h2>

<%= erb :palindrome_form %>
```

```
$ bundle exec rake test
6 tests, 16 assertions, 0 failures, 0 errors, 0 skips
```

```
$ git add -A
$ git commit -am "Finish working palindrome detector"
$ git push heroku
```

Register Your Product at informit.com/register

Access additional benefits and save up to 65%* on your next purchase

- Automatically receive a coupon for 35% off books, eBooks, and web editions and 65% off video courses, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.**
- Check the box to hear from us and receive exclusive offers on new editions and related products.

InformIT—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's leading learning company. At informit.com, you can

- Shop our books, eBooks, and video training. Most eBooks are DRM-Free and include PDF and EPUB files.
- Take advantage of our special offers and promotions (informit.com/promotions).
- Sign up for special offers and content newsletter (informit.com/newsletters).
- Access thousands of free chapters and video lessons.
- Enjoy free ground shipping on U.S. orders.*

** Offers subject to change.*

*** Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

Connect with InformIT—Visit informit.com/community

twitter.com/informit

Pearson

informIT

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Oracle Press • Peachpit Press • Pearson IT Certification • Que

Table of Contents