1

Welcome to Ruby

COPYRIGHTED MATERIAL

Welcome to Ruby on Rails! If you're a web developer, you're going to love Ruby on Rails—it's the easiest way to get real web applications going. If you've been a Java web programmer in the past, for example, you're going to think: this is great!

If you're used to huge, overly complex web applications, you're in for a treat—both Ruby and Rails can do a lot of the code writing for you, creating skeleton applications that you can modify easily. And if you're new to web programming, you're also in for a treat because you're getting started the right way.

Ruby is the programming language you're going to be using, and Rails is the web application framework that will put everything online. This and the next couple of chapters get you up to speed in Ruby, building the foundation you need to start putting Ruby on Rails. If you already know Ruby, you can skip this material and get directly to the online stuff.

But why just talk about it? Why not start by seeing Ruby in action? Heck, why not see Ruby on Rails in action, taking a look at just how simple it is to build a sample web application? That will give you something to keep in mind as you work through Ruby in these first few chapters. The first step, of course, is to install Ruby and Rails.

Installing Ruby and Rails

You're going to need both Ruby and Rails, so you should install both. Fortunately, that's not hard. The following sections lead you through installing on Windows, Mac OS X, and Linux/Unix.

Install Ruby and Rails on Windows

Just follow these steps to install Ruby:

1. Download the one-click installer for Ruby at `http://rubyinstaller.rubyforge.org`.
2. Click the installer to install Ruby.

Setting up Rails is just about as easy.

Select Start➪Run. Type **cmd** in the Open field and click OK.

Then, type the following at the command prompt:

```
gem install rails --include-dependencies
```

And that's it! You've got Ruby, and you've got Rails.

Install Ruby and Rails in Mac OS X

As of Mac OS X version 10.4 (the version called Tiger), installation is more than simple — Ruby comes built in. To check that, open the Terminal application by using Finder to go to Applications➪Utilities, double-click Terminal, and then enter **ruby -v** to check Ruby's version number.

You should see a version of Ruby that's 1.8.2 or later. If not, you're going to have to install Ruby yourself.

The locomotive project (`http://locomotive.raaum.org`) is a complete one-step install Rails environment for OS X.

Alternatively, you can download Rails yourself. To do that, you first need RubyGems, which you can pick up by going to `http://rubygems.rubyforge.org` and clicking the download link. Then go to the directory containing the download in the Terminal application and enter the following at the command prompt, updating `rubygems-0.8.10.tar.gz` to the most recent version of the download:

```
tar xzf rubygems-0.8.10.tar.gz
cd rubygems-0.8.10
sudo ruby setup.rb
```

The final step is to use RubyGems to download Rails, so enter this command:

```
sudo gem install rails --include-dependencies
```

That's it — you should be on Rails!

> **Ruby comes installed in Mac OS X, but it's not really installed all that well. For example, support for the Ruby readline library isn't working, which means that you could have problems if you want to use the interactive Ruby tool discussed later in this chapter. To fix the problem and install Ruby yourself, take a look at** `http://tech.ruby.com/entry/46`**.**

Install Ruby and Rails in Linux and Unix

If you're using Linux or Unix, you probably already have Ruby installed — you're going to need at least version 1.8.2 for this book. Open a shell and type **ruby -v** at the prompt — if you have version 1.8.2 or later installed, you're all set. If you don't have Ruby installed, you can find pre-built versions for your Linux/Unix installation on the Internet, or you can build it from the source, which you can find at `http:// ruby-lang.org/en`.

To build Ruby from source, download `ruby-1.8.4.tar.gz` (change that name to the most recent version), untar the file, and build it:

```
tar xzf ruby-1.8.4.tar.gz
cd ruby-1.8.4
./configure
make
make test
sudo make install
```

You're also going to need Rails, which is most easily installed with RubyGems. To get RubyGems, go to `http://rubygems.rubyforge.org` and click the download link. Then go to the directory containing the download in a shell and enter the following at the command prompt, updating `rubygems-0.8.10.tar.gz` to the most recent version of the download:

```
tar xzf rubygems-0.8.10.tar.gz
cd rubygems-0.8.10
sudo ruby setup.rb
```

All that's left is to use RubyGems to install Rails, which you can do this way:

```
sudo gem install rails --include-dependencies
```

Things very rarely go wrong with the Ruby and Rails installation process, but if there was a problem, take a look at the online help files, starting with `http://rubyinstaller.rubyforge.org/wiki/wiki.pl?RubyInstallerFAQ`.

Database System

You're also going to need to install a database system to get the most out of this book. This book uses MySQL, which you can get for free from `http://dev.mysql.com`. You will also learn how to set up Rails to work with other databases as well.

OK, now that you've got Ruby on Rails installed, how about creating a web application?

Creating a First Web Application

This and the next few chapters are on the Ruby language, which you're going to need to write Ruby on Rails web applications. But before getting into the details on Ruby, take a look at how easy it is to build a simple Ruby on Rails application. This first Rails application displays a friendly greeting message.

To organize the work you do in this book, you'll build all of your applications in a directory named rubydev. Create that directory now by typing `md rubydev` at the command prompt (in Windows, that might look like this):

```
C:\>md rubydev
```

Then navigate to that directory:

```
C:\rubydev>cd rubydev
C:\rubydev>
```

If you're using Linux or Unix or Mac or some other operating system, please translate these directions as appropriate. Ruby on Rails is operating-system independent, so you shouldn't run into any trouble in this regard — you'll run into very few operating system commands in this book. For example, if you're using the Bash shell, enter this:

```
-bash-2.05b$ mkdir rubydev
-bash-2.05b$ cd rubydev
-bash-2.05b$
```

If you're using other varieties of Linux/Unix, you see something like this:

```
/home/steve: mkdir rubydev
/home/steve: cd rubydev
/home/steve/rubydev:
```

This first application is named hello. To tell Rails to create the hello application, type the command **rails** and then the name of the application you want — in this case, hello. Rails creates many files for you (most of which are omitted here for brevity):

```
C:\rubydev>rails hello
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
      create  components
      create  db
      create  doc
      create  lib
      create  lib/tasks
      create  log
      create  public/images
      create  public/javascripts
      create  public/stylesheets
      create  script/performance
      create  script/process
      create  test/fixtures
      create  test/methodal
      create  test/integration
        .
        .
        .
      create  public/favicon.ico
      create  public/robots.txt
      create  public/images/rails.png
```

```
      create  public/javascripts/prototype.js
      create  public/javascripts/effects.js
      create  public/javascripts/dragdrop.js
      create  public/javascripts/controls.js
      create  public/javascripts/application.js
      create  doc/README_FOR_APP
      create  log/server.log
      create  log/production.log
      create  log/development.log
      create  log/test.log
```

Great; that sets up the framework you're going to need. The next step is to create a *controller* for the application. A controller acts as the application's overseer, as you're going to see in detail when you start working with Rails in depth later in this book. Rails has already created a new directory, hello, for the hello application, so change to that directory now:

```
C:\rubydev>cd hello
C:\rubydev\hello>
```

Then use the Ruby command `ruby script/generate controller App` to create a new controller named `App`:

```
C:\rubydev\hello>ruby script/generate controller App
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/app
      exists  test/methodal/
      create  app/controllers/app_controller.rb
      create  test/methodal/app_controller_test.rb
      create  app/helpers/app_helper.rb
```

The new controller is created. In fact, here's what it looks like, in the file rubydev\hello\app\controllers\app_controller.rb (the `.rb` extension, as you've probably already guessed, stands for Ruby):

```
class AppController < ApplicationController
end
```

The controller is derived from the base class `ApplicationController` (if you don't know about inheriting, don't worry about it — that's coming up in Chapter 3). As you can see, it's a very simple file.

The controller determines the flow of navigation in an application. Each controller contains one or more actions. Each action knows how to respond when a specific input is received from the browser. Actions can call and prepare data for display in views or return other content directly to the browser. The controller will contain all the actions you want in the web application, and each action is supported with its own code.

So while the controller oversees whole web application, each action that the controller watches over contains methods as separate tasks. That makes it easy to build web applications as collections of separate tasks (actions).

The hello application uses an example action named `greeting` to display its message in a browser window. To create the greeting action, just add the following code to `app_controller.rb`, using your favorite text editor:

```
class AppController < ApplicationController
  def greeting
  end
end
```

Make sure that your text editor can save text in plain text format with the extension `.rb` (not the default `.rtf` format of Windows WordPad, for example, or the default Notepad extension `.txt`), and save the file after you've added the greeting action.

So you've created a web application with a controller that oversees the action and an action that acts as a task you can ask the controller to execute. You still need to display its results. In Rails, a *view* is often used to do that. A view is a web page template that the action can call and supply data to. A view can format data as a web page and display it in the browser. In other words, you can construct a full web page template with all kinds of formatting — colors, fonts, and so on — and pop the data items supplied by the action into that template in appropriate places before sending it back to the browser.

Now you've already learned three concepts vital to Ruby on Rails programming:

- The controller oversees the application.
- The action or actions in the application act as individual tasks.
- The view or views accept data from the actions and display that data in web pages.

In this example, no data will be passed from the action to the view template; this view will only display a friendly greeting web page, `greeting.rhtml`, like this:

```
<html>
  <head>
    <title>Ruby on Rails</title>
  </head>
  <body>
    <h1>Yes it's working!</h1>
  </body>
</html>
```

In Rails applications, view templates like this one have the extension `.rhtml` (that's so Rails knows it is an active web page that should be checked to see if you've left places where data from the action should be added). Create the `greeting.rhtml` file now, and store it in the rubydev\hello\app\views\app\ folder.

> *If you are using Windows WordPad or Notepad, you're going to notice that they have an annoying tendency to append the extension* `.rtf` *or* `.txt`*, respectively, to the end of any file whose extension they don't understand — and that includes Ruby* `.rb` *and Rails* `.rhtml` *files. So instead of* `greeting.rhtml`*, you might end up with* `greeting.rhtml.rtf` *or* `greeting.rhtml.txt` *when you try to save this file. To get around that, place the name of the file in quotation marks when you save it, such as* `"greeting.rhtml"`*. That stops WordPad and Notepad from adding their default extensions.*

And that's it — believe it or not, you're done. You've created your first Ruby on Rails web application. How can you see it in a web browser? Rails comes with a built-in web server that you can use for testing. To start that web server, just use `ruby script/server` at the command line, as you see here:

```
C:\rubydev\hello>ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
[2006-05-03 11:10:29] INFO  WEBrick 1.3.1
[2006-05-03 11:10:29] INFO  WEBrick::HTTPServer#start: pid=4008 port=3000
```

Now open a browser. The Ruby on Rails server operates locally, and uses port 3000, so you start the URL `http://localhost:3000`. Then you specify the web application controller's name (`app`, in this case) and the name of the web application's actions — that is, tasks — you want to execute (`greeting`). That makes the full URL `http://localhost:3000/app/greeting`.

Navigate to `http://localhost:3000/app/greeting` and you'll see the web page shown in Figure 1-1, complete with the greeting from the web application.

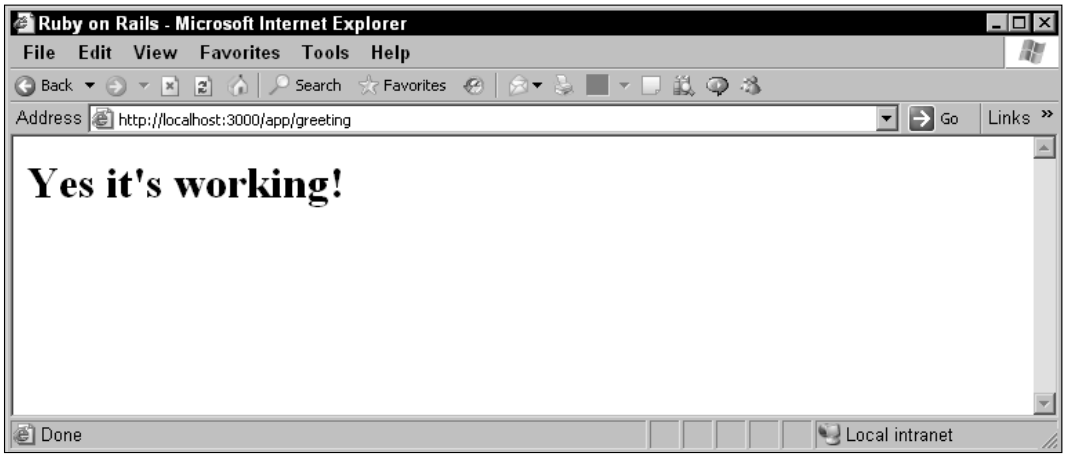

Figure 1-1

Congratulations! You've just completed and run your first Ruby on Rails application, and it took practically no time at all. To create web applications that do something more impressive, though, you need to get the Ruby language under your belt. That's what the rest of this and the next couple of chapters are all about — building the foundation you' re going to need when it comes to working with Ruby on Rails.

Getting Started with Ruby

Ruby is the language that is going to make everything happen. To work with Ruby, you need a text editor of the kind you already used to get the Ruby on Rails example working, such as WordPad (Start➪Programs➪Accessories➪WordPad) or Notepad (Start➪Programs➪Accessories➪Notepad) in Windows.

Each Ruby program should be saved with the extension `.rb`, such as this first example, `hello.rb`, which displays a greeting from Ruby.

Try It Out Display a Message

To get started with Ruby and make it display a welcoming message, follow these steps:

1. Start your text editor and enter the following Ruby code:

```
puts "Hello from Ruby."
```

2. Save the file as `hello.rb`. Make sure you save the file as a text file (select Text Document in the Save As Type drop-down), and if you are using Windows WordPad or Notepad, make sure you enclose the name of the file in quotes — `"hello.rb"` — before saving to prevent those editors from saving the file as `hello.rb.rtf` or `hello.rb.txt`.
3. Use Ruby to run this new program and see the results. Just enter the ruby command followed by the name of the program at the command line:

```
C:\rubydev>ruby hello.rb
Hello from Ruby.
```

How It Works

This first example is simple — it executes a line of Ruby code:

```
puts "Hello from Ruby."
```

What is `puts`? That's a *method* built into Ruby, and it stands for "put string." A method is a piece of executable code that you can pass data to — in this case, the `puts` method takes the data you pass to it (that's `"Hello from Ruby."` here) and displays it in the command window.

Note that this line of code doesn't end with a semicolon as in some other programming languages you'll see, such as Java or JavaScript. You don't need any end-of-line marker at all in Ruby. If you can't fit a Ruby statement on one line, you can tell Ruby that you are continuing that line of code on a second line by using backslashes, like this:

```
puts \
"Hello from Ruby."
```

So if you ever need to break a line of Ruby onto more than one line, just use a backslash at the end of each line — except for the last line — to indicate that more is coming.

You can also create *comments* in Ruby, using the `#` symbol. A comment is human-readable text that Ruby will ignore. When you use `#` on a line, everything you place after the `#` will be ignored by Ruby, although you can read it:

```
puts "Hello from Ruby."    #Display the text!
```

Working with Ruby Interactively

This first example was a success, but you should be aware that there's another way to work with Ruby interactively—you can use the interactive Ruby tool `irb`. You can start that tool from the command line:

```
C:\rubydev>irb
irb(main):001:0>
```

This displays the `irb` prompt, and you can enter your Ruby code at that prompt, such as `puts "Hello from Ruby."`:

```
C:\rubydev>irb
irb(main):001:0> puts "Hello from Ruby."
```

When you press the Enter key, `irb` evaluates your Ruby and gives you the result:

```
C:\rubydev>irb
irb(main):001:0> puts "Hello from Ruby."
Hello from Ruby.
```

Although you can create multi-line programs using `irb`, it's awkward, so this book sticks to entering Ruby code in files instead.

Checking the Ruby Documentation

What about documentation? Does Ruby come with any documentation that you can use to look up questions? Yes, it does, and there's more online. To handle the local version of the documentation, you use the `ri` tool. Just enter **ri** at the command line, followed by the item you want help with, such as the `puts` method:

```
C:\rubydev>ri puts
More than one method matched your request. You can refine
your search by asking for information on one of:

     IO#puts, Kernel#puts, Zlib::GzipWriter#puts
```

If more than one item matches your request, you have to choose which one to ask for more information on. In this example, the `puts` method of interest is part of the IO package (more on packages in Chapter 3), so request that documentation by entering **ri IO#puts** at the command line:

```
C:\rubydev>ri IO#puts
------------------------------------------------------------- IO#puts
     ios.puts(obj, ...)    => nil
------------------------------------------------------------------------
     Writes the given objects to _ios_ as with +IO#print+. Writes a
     record separator (typically a newline) after any that do not
     already end with a newline sequence. If called with an array
     argument, writes each element on a new line. If called without
```

```
        arguments, outputs a single record separator.

           $stdout.puts("this", "is", "a", "test")

        _produces:_

           this
           is
           a
           test
```

The `ri` tool searches the local Ruby documentation for the item you have requested and displays any help it can find. Unfortunately, that help is likely to be terse and not very helpful. Still, the local documentation can be useful, especially when it comes to examples.

Another place to turn is the online Ruby documentation site, `www.ruby-doc.org`. This site isn't overly easy to use, but its information is usually more complete than the local version of the documentation.

Working with Numbers in Ruby

Ruby has some great features for working with numbers. In fact, Ruby handles numbers automatically, so you need not be concerned about them. There is no limit to the size of integers you can use — a number like `12345678987654321` is perfectly fine. What's more, you make this number easier to read by using underscores every three digits from the right, like this: `12_345_678_987_654_321`. You can give floating-point numbers simply by using a decimal point (you need at least one digit in front of the decimal point), like this: `3.1415`. You can also give an exponent like this: `31415.0e-4`. And you can give binary numbers by prefacing them with `0b` (as in `0b1111`), octal — base eight — numbers by prefacing them with a 0 (like this: 0355), and hexadecimal numbers — base 16 — by prefacing them with `0x` (such as `0xddff`).

> *Ruby stores numbers in a variety of types. For example, integers are stored as the* `Fixnum` *type unless they become too large, in which case they are stored as the* `Bignum` *type. And floating-point numbers are stored as the* `Float` *type. However, you don't have to worry about of this, because Ruby keeps track of a number's type internally.*

Try It Out Working with Numbers

To get started with numbers in Ruby, follow these steps:

1. Enter this Ruby code in a new file:

```
puts 12345
puts 3.1415
puts 31415.0e-4
puts 12_345_678_987_654_321
puts 0xddff
```

2. Save the file as `numbers.rb`.
3. Use Ruby to run `numbers.rb`:

```
C:\rubydev>ruby numbers.rb
12345
3.1415
3.1415
12345678987654321
56831
```

How It Works

This example takes several numbers and uses the `puts` method to display them in a command window. As you can see, it displays a few integers and floating-point numbers — even a hexadecimal number (which is displayed in decimal). Note the number 31415.0e-4, which is actually 31415.0 x 10^{-4}. Ruby handles this number as it should, and displays it in simpler format: 3.1415.

Working with Strings in Ruby

You can enclose strings in single quotes (`'Shall we watch a movie?'`) or double quotes (`"No, we certainly shall not."`). In fact, you can mix single and double quotes, as long as Ruby can keep them straight, like this: `"Did you say, 'No, we certainly shall not'?"`. However, you cannot mix double-quoted text inside a double-quoted string, or single-quoted text inside a single-quoted string, like this: `"I said, "Hello.""`, because Ruby won't know where the quotation actually ends. To fix that, you can alternate single and double quotes, like this: `"I said, 'Hello.'"` or `'I said, "Hello."'`, or you can escape quotation marks with a backslash, like this: `"I said, \"Hello.\""` or `'I said, \'Hello.\''`.

In fact, you don't even have to use single or double quotes — Ruby can add them for you if you use the `%q` (single quotes) or `%Q` (double quotes) syntax. Just use `%q` or `%Q` with a single character, and Ruby will add quotes until it sees that character again. For example, the expression `%Q/Yes, that was Cary Grant./` is the same as `"Yes, that was Cary Grant."` And it's also the same as the expression `%Q!Yes, that was Cary Grant.!`. The expression `%q/Are you sure?/` is the same as `'Are you sure?'`. You can also use pairs of delimiters, such as `{` and `}` or < and > when quoting, like this: `%Q{Yes, that was Cary Grant.}`. In fact, you can omit the `Q` altogether — if you do, you'll get a double-quoted string: `%/No, it wasn't Cary./` is the same as `"No, it wasn't Cary."`

You can also concatenate (join) strings together, using a `+`. For example, the expression `"It " + "was " + "too " + "Cary " + "Grant!"` is the same as `"It was too Cary Grant!"`.

Try It Out Working with Strings

To get started with strings in Ruby, follow these steps:

1. Enter this Ruby code in a new file, `strings.rb`:

```
puts "Hello"
puts "Hello " + "there"
puts 'Nice to see you.'
puts %Q/How are you?/
puts %Q!Fine, and you?!
puts %q!I'm also fine, thanks.!
puts "I have to say, 'I am well.'"
puts "I'll also say, \"Things are fine.\""
```

2. Save the file as `strings.rb`.
3. Run `strings.rb` using Ruby to see the result:

```
C:\rubydev>ruby strings.rb
Hello
Hello there
Nice to see you.
How are you?
Fine, and you?
I'm also fine, thanks.
I have to say, 'I am well.'
I'll also say, "Things are fine."
```

How It Works

This example gives a good overview of the ways of working with single- and double-quoted text in Ruby. And, there's even more to come — double-quoted strings have some additional power that you'll see after the upcoming discussion on variables.

There's another way of working with text strings that you should know about in Ruby: `HERE` documents. `HERE` documents are inherited in Ruby from older languages like Perl, and they give you a shortcut for printing multiple lines to the console window. Although not in common use, you should still know what they are in case you come across them.

`HERE` documents let you break up strings into multiple lines; Ruby treats any text that starts with `<<TOKEN` (where `TOKEN` can be any uppercase word) as the beginning of a multi-line sequence, and assumes that that text ends with `TOKEN`. Here is an example that uses the Ruby `print` method, which prints out the text you pass it and, unlike `puts`, does not skip to the next line (that is, `print` does not add a carriage return to the text):

```
print <<HERE
Now
is
the
time
HERE
```

You don't have to use `HERE` *as the token; you can use any word.*

This displays this text:

```
Now
is
the
time
```

Although the `print` method does not skip to the next line automatically, the output here did skip to the next line. That's because a `HERE` document automatically inserts a newline character at the end of each line, enabling you to create multi-line text easily.

If you want to make the `print` method skip to the next line without a `HERE` document, use the newline character, `\n`. For example,

```
print "Now is \nthe time."
```

prints out

```
Now is
the time.
```

Storing Data in Variables

Ruby can store your data in *variables*, which are named placeholders that can store numbers, strings, and other data. You reference the data stored in a variable by using the variable's name. For example, to store a value of 34 in a variable named `temperature`, you assign that variable the value like this:

```
temperature = 34
```

Then you refer to the data in the `temperature` variable by name, as here, where you are printing out the value in the `temperature` variable (34) in a command window:

```
puts temperature
```

Here's the result:

```
34
```

Because you refer to the values in variables by using the variables' names in code, it's important to realize that there are rules for the names you can use in Ruby. A standard variable starts with a lowercase letter, *a* to *z*, or an underscore, _, followed by any number of *name characters*. A name character is a lowercase letter, an uppercase letter, a digit, or an underscore. And you have to avoid the words that Ruby reserves for itself. Here's a list of Ruby's reserved words:

```
__FILE__    def         in        self
__LINE__    defined?    module    super
BEGIN       do          next      then
END         else        nil       true
alias       elsif       not       undef
and         end         or        unless
begin       ensure      redo      until
break       false       rescue    when
case        for         retry     while
class       if          return    yield
```

It's also the Ruby convention to use underscores rather than "camel case" for multiple-word names; `money_I_owe_my_cousin` is good, for example, but `moneyThatIOweMyCousin` is not.

> *Some variables are named with an `@` (as in `@money_I_owe_my_cousin`) or even `@@` (as in `@@money_I_owe_my_cousin`). You learn about those in Chapter 3 when you write classes in Ruby.*

To create a variable, you just have to use it in Ruby. You can see that for yourself in the following exercise.

Try It Out Working with Variables

To get started with variables in Ruby, follow these steps:

1. Enter this Ruby code in a new file, `variables.rb`:

```
temperature = 36
puts "The temperature is " + String(temperature) + "."
temperature = temperature + 5
puts "Now the temperature is " + String(temperature) + "."
```

2. Save the `variables.rb` file.
3. Run `variables.rb` to see the result:

```
C:\rubydev>ruby variables.rb
The temperature is 36.
Now the temperature is 41.
```

How It Works

This example first creates a variable named `temperature` and stores a value of `36` in it:

```
temperature = 36
```

Then it prints out the value in the `temperature` variable. Note that you convert the integer in the `temperature` variable to a string before trying to print it out, and you can do that with the Ruby `String` method — the parentheses after the `String` method name are to make certain that you just pass the temperature variable to the `String` method, not any additional text on the line:

```
puts "The temperature is " + String(temperature) + "."
```

Instead of the `String` method, you could use the `to_s` method built into all numbers — that method does the same as the `String` method (and in fact, the `String` method simply calls the number's `to_s` method):

```
temperature = 36
puts "The temperature is " + temperature.to_s + "."
```

Then the example uses the `+` operator to add 5 to the value in the `temperature` variable. Using operators, as you're about to see, lets you manipulate the data stored in variables:

```
temperature = temperature + 5
```

Finally, this example displays the new value stored in the `temperature` variable, which is 36 + 5, or 41:

```
puts "Now the temperature is " + String(temperature) + "."
```

Creating Constants

You can also create constants in Ruby. A *constant* holds a value that you do not expect to change, such as the value of pi:

```
PI = 3.1415926535
```

Note the uppercase name, `PI`, here. Constants in Ruby start with an uppercase letter — that's how Ruby knows they are constants. In fact, the entire name of a constant is usually in uppercase. But it's that first letter that is crucial — it must be uppercase so that Ruby knows that you intend to create a constant.

Constants are often used to collect data items that won't change at the beginning of your code so that they can be grouped together and changed easily if need be — you use the constants throughout your code, and won't have to hunt for the specific values the constants stand for and change them throughout your code if you need to change a value. For example, constants might hold Internet IP addresses:

```
IP_SERVER_SOURCE = "903.111.333.055"
IP_SERVER_TARGET = "903.111.333.056"
```

These constants, `IP_SERVER_SOURCE` and `IP_SERVER_TARGET`, can now be used throughout your code, and if you have to change them, you only need to change them in one place.

Unlike other languages, Ruby allows you to change the values in constants by assigning a new value to them:

```
CONST = 1
CONST = 2
```

However, you'll get a warning each time you do this:

```
constants.rb:2: warning: already initialized constant CONST
```

Interpolating Variables in Double-Quoted Strings

As mentioned earlier, double-quoted strings have a special property — you can display the values of variables directly in them using a process called *interpolation*. Here's how it works: you surround the expression you want placed into the double-quoted string with `#{` and `}`, and when Ruby sees that, it'll evaluate that expression and substitute its value at that location in the string. For example, you saw this code a page or two ago:

```
temperature = 36
puts "The temperature is " + String(temperature) + "."
```

But you can get the same result using variable interpolation, like this:

```
temperature = 36
puts "The temperature is #{temperature}."
```

In this case, the term `#{temperature}` is evaluated to the value stored in the temperature, 36, and that result is displayed:

```
The temperature is 36.
```

Try It Out Interpolating Variables

To get started with interpolating expressions in double-quoted strings in Ruby, follow these steps:

1. Enter this code in a new file, `doublequoted.rb`:

```
temperature = 36
puts "The temperature is #{temperature}."
temperature = temperature + 5
puts "Now the temperature is #{temperature}."
```

2. Save `doublequoted.rb`.
3. Run `doublequoted.rb`:

```
C:\rubydev>ruby doublequoted.rb
The temperature is 36.
Now the temperature is 41.
```

How It Works

This example interpolates the value in the `temperature` variable into a double-quoted string: `"The temperature is #{temperature}."` When Ruby sees that, it substitutes the value of the temperature variable in the double-quoted string automatically.

In fact, you can place any expression inside `#{` and `}` in a double-quoted string and have it interpolated into the text. For example, to add 5 to the value in the `temperature` variable, you could simply do this:

```
temperature = 36
puts "The temperature is #{temperature}."
puts "Now the temperature is #{temperature + 5}."
```

Reading Text on the Command Line

You've seen how to print out text in a console window using `puts` (displays a text string) and `print` (displays a text string but does not skip to the next line), but what about reading text on the command line? How do you get Ruby to read text that the user has typed? The most common way is to use the built-in `gets` method. That method lets you read text from the command line that the user has entered, and, by default, assigns that text to the predefined variable `$_`, which also comes built into Ruby.

Here's one thing you should know — the `gets` method leaves the terminating newline character that the user enters when he's done entering text on the end of the string, so if the user enters `Stop` and then presses Enter, the input you'll get will be `Stop\n`, where `\n` is a newline character.

And here's another thing you should know — the built-in Ruby method `chomp` removes that newline character from the end of the text in the `$_` variable. Problem solved.

Try It Out Reading Text

To get started reading text in Ruby, follow these steps:

1. Enter this code in a new file, `gets.rb`:

```
print "Please enter the temperature: "
gets
chomp
puts "The temperature is #{$_}."
```

2. Save the file and run it.

3. Ruby displays the prompt `Please enter the temperature:` and then waits for a response from you:

```
C:\rubydev>ruby gets.rb
Please enter the temperature:
```

4. Enter a temperature and press Enter. Ruby reads the text that you have entered, chomps the newline character off the end of it, and displays the resulting text:

```
C:\rubydev>ruby gets.rb
Please enter the temperature: 36
The temperature is 36.
```

How It Works

This code works by displaying a prompt (with `print` so Ruby doesn't skip to the next line in the command window) and using `gets` to read what the user has entered. By default, `gets` places the text it has read in the `$_` predefined variable, so you can use `chomp` to get rid of the newline character at the end of that text. Then you can display the resulting trimmed text using `puts`.

Actually, you don't need to use `$_` with `gets` — you can assign the text it reads to any variable, like this: `temperature = gets`. However, that still leaves you the problem of removing the newline character at the end of the text with `chomp`, because you do have to use `$_` with `chomp` — you can't specify a variable to `chomp`. That would make your code look like this:

```
print "Please enter the temperature: "
temperature = gets
$_ = temperature
chomp
temperature = $_
puts "The temperature is #{temperature}."
```

Creating Symbols in Ruby

You've seen how to create variables and constants, but there's more to come in Ruby, such as *symbols*. As you know, when you use a variable in Ruby code, Ruby substitutes the value of that variable for the variable itself. But what if you just wanted to use a name, without having it stand for anything? For that, you can use a Ruby symbol (called atoms in other languages), which is preceded by a colon (`:`). Here are a few examples:

```
:temperature
:ISP_address
:row_delimiter
```

Each of these symbols is treated simply as a name; no substitution is performed when you use them in your code.

At this point, you can think of symbols much as you would quoted strings that contain names. In fact, they are really very much like quoted strings, with a couple of technical differences — each symbol always stands for the same object (more on objects is coming up in Chapter 3), no matter where you use it in your code, which is different from strings; and comparing symbols to each other is faster than comparing strings.

Working with Operators

It's time to start working with the data stored in the variables you create. As you've already seen, you can add numbers to the values you store in variables — for example, this code added 5 to the value in the temperature variable:

```
temperature = temperature + 5
```

The code used the `+` addition operator to add 5 to the value in the temperature variable, and the `=` assignment operator to assign the result to the `temperature` variable, but many other operators are available.

Try It Out Using Operators

To get started using operators in Ruby, follow these steps:

1. Enter this code in a new file, `operators.rb`:

```
value = 3
puts value
value = value + 3   #addition
puts value
value = value / 2   #division
puts value
value = value * 3   #multiplication
puts value
value = value ** 2  #exponentiation
puts value
```

2. Save `operators.rb`.

3. Run `operators.rb`:

```
C:\rubydev>ruby operators.rb
3
6
3
9
81
```

How It Works

This example shows a number of the Ruby operators (`+`, `/`, `*`, and `**`) at work. Table 1-1 describes many of the Ruby operators.

Table 1-1 Ruby Operators

Operator	Description
`[ ]` `[ ]=`	Array reference Array element set
`**`	Exponentiation

`!`
`~`
`+`
`-`
	Not
	Complement
	Unary plus
	Minus

`*`
`/`
`%`
	Multiply
	Divide
	Modulo

`+` `-`	Plus Minus
`>>` `<<`	Right shift Left shift
`&`	Bitwise And
`^` `|`	Bitwise exclusive Or (Xor) Regular Or

`<=`
`<`
`>`
`>=`
	Comparison operators:
	Less than or equal to
	Less than
	Greater than
	Greater than or equal to

`<=>`
`==`
`===`
`!=`
`=~`
	Equality and pattern match operators:
	Less than, equal to, greater than
	Equal to
	Tests equality in a `when` clause of a `case` statement
	Not equal to
	Regular expression pattern match

Table continued on following page

Table 1-1 Continued

Operator	Description
`&&`	Logical And
`||`	Logical Or
`..` `...`	Inclusive range Exclusive range
`?` `:`	Ternary if Else

`=`
`%=`
`/=`
`-=`
`+=`
`*=`
`**=`
	Assignment
	Normal assign
	Modulus and assign
	Divide and assign
	Subtract and assign
	Add and assign
	Multiply and assign
	Exponent and assign

`defined?`	True if symbol defined
`not`	Logical negation
`and` `or`	Logical composition

`if`
`unless`
`while`
`until`
	Statement modifiers

`begin/end`	Block expression

Some of the operators have assignment shortcuts; for example, this line of code

```
value = value + 3
```

can be made shorter with the += addition assignment operator, like this:

```
value += 3
```

In other words, the += shortcut operator adds its right operand to its left operand and assigns the result to the left operand. And that means that the preceding example could be rewritten using shortcut operators:

```
value = 3
puts value
value += 3   #addition
puts value
```

```
value /= 2   #division
puts value
value *= 3   #multiplication
puts value
value **= 2  #exponentiation
puts value
```

Handling Operator Precedence

Here's a question for you: what does the following Ruby statement print out?

```
puts 5 + 3 * 2
```

Does it add the 5 and 3 first, and then multiply the result (8) by 2 to get 16? Or does it multiply 3 by 2 to get 6, and then add the 5 to get 11? Let's see how Ruby answers the question:

```
C:\rubydev>ruby precedence.rb
11
```

The answer is that it multiplies the 3 by 2 first, and then adds 5, which might not be what you expected.

The reason Ruby performed the multiplication first, before the addition, is that multiplication has higher *precedence* than addition in Ruby. An operator's precedence specifies the order in which it is executed with respect to other operators in the same statement.

The operators in Table 1-1 are arranged in terms of precedence, from highest at the top to lowest at the bottom. As you can see in the table, the multiplication operator `*` has higher precedence than the addition operator `+`, which means that `*` will be executed first.

You can alter the execution order, if you want to, using parentheses. Just wrap the expression you want evaluated first inside parentheses, like this:

```
puts (5 + 3) * 2
```

In this case, 5 and 3 will be added first, and the result, 8, multiplied by 2, to yield 16:

```
C:\rubydev>ruby precedence.rb
16
```

Working with Arrays

There are plenty of ways to work with data in Ruby. Next up is to store it in *arrays*. Arrays act as groups of variables, and you can access each variable in an array with an index number.

For example, to create an array, you use the `[]` operator like this:

```
array = [1, 2, 3]
```

This creates an array with three elements, 1, 2, and 3. You access those elements using an array index like this:

```
puts array[0]       #prints 1
puts array[1]       #prints 2
puts array[2]       #prints 3
```

Note that the first element in the array corresponds to array index `0`, not `1`, and the second element is at index `1`, the third at index `2`, and so on. You can also assign values to arrays, using the array index:

```
array[0] = 4        #assigns 4 to array[0]
array[1] = 5        #assigns 5 to array[1]
array[2] = 6        #assigns 6 to array[2]
puts array[2]       #prints 6
```

As you know, Ruby variables can store text strings as well as numbers, and you can store both in arrays. Here's an example:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
```

This creates an array filled with six elements, three of which are text and three of which are numbers:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
puts array[1]       #prints "there"
puts array[4]       #prints 2
```

Arrays are handy in code because you have control over the numeric array index, which means that you can move over all the elements in an array in code, handling all the data at once. For example, you store test scores of students in a class on Ruby that you are teaching. To find the average test score, you could simply increment the array index, fetching each individual score and adding them up. Then you could divide by the total number of students.

How could you find the total number of students in the array? You could use the array's built-in `length` method, which returns the number of elements in the array:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
puts array[1]       #prints "there"
puts array[4]       #prints 2
puts array.length   #prints 6
```

Try It Out Using Arrays

To get started using arrays in Ruby, follow these steps:

1. Enter this code in a new file, `arrays.rb`:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
puts array[0]
puts array[1]
puts array.length
array2 = Array.new
```

```
puts array2.length
array2[0] = "Banana"
array2[1] = "fish"
puts array2[0] + " " + array2[1]
puts array2.length
```

2. Save the file and run it:

```
C:\rubydev>ruby arrays.rb
Hello
there
6
0
Banana fish
2
```

How It Works

This example creates an array, fills it with data, extracts data from the array, and displays the length of the array. Then this example does something new — it creates a second array with the `Array` class's `new` method:

```
array2 = Array.new
```

That creates a new array named `array2`, and gives it zero length (because it doesn't have any elements in it yet):

```
puts array2.length         #prints 0
```

Now you can use this array as you did the first array in this example, assigning data to the elements in the array, and fetching data from those elements as you want:

```
array2[0] = "Banana"
array2[1] = "fish"
puts array2[0] + " " + array2[1]
puts array2.length
```

When you assign data to a new element in an array, that element is automatically created if it didn't exist before. Note that at the end of this code, `puts array2.length` prints out the new length of the array, which is 2.

Here's something else you should know — you don't have to place data in array elements consecutively. After creating a new array, you can fill its element `0`, then skip over element `1`, and fill element `2` like this, no problem:

```
array2 = Array.new
puts array2.length
array2[0] = "Banana"
array2[2] = "fish"
puts array2[0] + " " + array2[2]     #prints "Banana fish"
```

In fact, you can use *negative* array indices in Ruby as well. Negative indices count from the end of the array back to the beginning of the array. Here's an example:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
puts array[-1]            #prints 3
```

This example prints out `3`, the last element in the array. It often confuses people that the first element in an array is `0`, the next `1`, and so on — but the last element in the array is `-1` (which means negative indices start with `-1`, not `0`), the previous one `-2`, the previous one `-3`, and so on. But that does make sense — how could negative indices start with `0`? That array index is already taken. Using negative array indices, you can move backward through an entire array:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
puts array[-1]            #prints 3
puts array[-2]            #prints 2
puts array[-3]            #prints 1
```

In addition to numbers, you can use a variable (or a constant) as an array index. Here's an example:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
index_value = 0
puts array[index_value]   #prints "Hello"
index_value = 1
puts array[index_value]   #prints "there"
index_value = 2
puts array[index_value]   #prints "sweetie"
```

Want a quick way of printing out an entire array? Just use `puts` on the entire array:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
puts array
```

This code gives you:

```
Hello
there
sweetie
1
2
3
```

Using Two Array Indices

You can access the data in arrays using two indices, not just one. This works differently from other programming languages you might be used to, however. Instead of letting you work with two-dimensional arrays, in Ruby the first index is the start location and the second holds the number of elements you want to access. So you can call the first index the start point, and the second index the count of elements to access: `array[start, count]`.

For example, here's an array:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
```

Now to replace element 1, you can do this:

```
array[1] = "here"
```

That leaves you with the array

```
["Hello", "here", "sweetie", 1, 2, 3]
```

An equivalent way of doing things is to use a double index, indicating that you want to replace element 1 only, like this:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
array[1, 1] = "here"
```

You get the same result:

```
 ["Hello", "here", "sweetie", 1, 2, 3]
```

However, what if instead of `array[1, 1]`, you used `array[1, 2]`? Then you would be referencing two array elements, starting with the element at index 1, and this statement replaces two elements in the array, not just one:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
array[1, 2] = "here"
```

Here is the result:

```
 ["Hello", "here", 1, 2, 3]
```

Do you get it? The expression `array[1, 2]` referred to two elements in the array (`"there"` and `"sweetie"`), starting at index 1.

What about using a count of `0`?

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
array[3, 0] = "pie"
puts array
```

Here's what you get:

```
Hello
there
sweetie
pie
1
2
3
```

In other words, assigning a value to `array[3, 0]` did not replace any element in the array; it inserted a new element starting at index 3 instead.

Try It Out Using Two Array Indices

To get started using two array indices in Ruby, follow these steps:

1. Enter this code in a new file, `doubleindex.rb`:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
array[2, 1] = "pal"
puts array
array = ["Hello", "there", "sweetie", 1, 2, 3]
array[3, 0] = "pie"
puts array
array = ["Now", "is", 1, 2, 3]
array[2, 0] = ["the", "time"]
puts array
array = ["Hello", "there", "sweetie", 1, 2, 3]
array2 = array[3, 3]
puts array2
```

2. Save the file and run it:

```
C:\rubydev>ruby doubleindex.rb
Hello
there
pal
1
2
3
Hello
there
sweetie
pie
1
2
3
Now
is
the
time
1
2
3
1
2
3
```

How It Works

This example uses double indices in arrays, and there are a few new points here. Take a look at this code:

```
array = ["Now", "is", 1, 2, 3]
array[2, 0] = ["the", "time"]
puts array
```

Here, the code inserts an entire new array, `["the", "time"]`, into the array, and you can see the results:

```
Now
is
the
time
1
2
3
```

In addition, you can extract subarrays using double indices. Take a look at this code from the example:

```
array = ["Hello", "there", "sweetie", 1, 2, 3]
array2 = array[3, 3]
puts array2
```

This extracts a second array, `array2`, from the first array. This second array is made up of the elements in the first array starting at element three, and continuing for three elements, as you can see when the code displays the array:

```
1
2
3
```

Working with Hashes

Arrays are powerful, but isn't there some way to work with a collection of data using words instead of numbers as indices? You might have a set of relatives who owe you money, for example, and it's hard to remember the numbers you gave to them — is `money_I_am_owed[1]` the money your brother Dan or your sister Claire owes you?

There is a solution; you can use *hashes*. A hash, sometimes called an associative array or a dictionary, is much like an array in that it holds a collection of data, but you can index it using text strings as well as numbers.

To create a hash, you use curly braces, `{` and `}`, not square braces (`[]`) as with arrays. Here's an example:

```
money_I_am_owed = {"Dan" => "$1,000,000", "Claire" => "$500,000"}
```

Here, `"Dan"` is a hash *key* (in hashes, the index is usually called the key), and `"$1,000,000"` is the *value* associated with that key in the hash. The `=>` operator separates the keys from the values.

You can extract data from the hash by using its keys. For example, to find out the amount your brother Dan owes you, you can use this code:

```
money_I_am_owed = {"Dan" => "$1,000,000", "Claire" => "$500,000"}
puts money_I_am_owed["Dan"]
```

Note that to access data in a hash, you use `[]`, as with an array, not `{}` as you might expect (and as you use in other languages).

This code will display:

```
C:\rubydev>ruby hashes.rb
$1,000,000
```

Not bad — certainly a lot easier to remember that your brother corresponds to index `1` and `money_I_am_owed[1]` the amount of money he owes you. In fact, you can use numbers as keys as well if you want to:

```
money_I_am_owed = {1 => "$1,000,000", 2 => "$500,000"}
puts money_I_am_owed[1]
```

This code will display:

```
C:\rubydev>ruby hashes.rb
$1,000,000
```

Try It Out Using Hashes

To get started using hashes in Ruby, follow these steps:

1. Enter this code in a new file, `hashes.rb`:

```
pizza = {"first_topping" => "pepperoni", "second_topping" => "sausage"}
puts pizza["first_topping"]
puts pizza
puts pizza.length
receipts = {"day_one" => 5.03, "day_two" => 15_003.00}
puts receipts["day_one"]
puts receipts["day_two"]
```

2. Save the file and run it:

```
C:\rubydev>ruby hashes.rb
pepperoni
first_toppingpepperonisecond_toppingsausage
2
5.03
15003.0
```

How It Works

This example puts hashes to work, and there are a few things to note here. First, using `puts` on a hash in Ruby doesn't result in as nice a display as it does for arrays. For example:

```
pizza = {"first_topping" => "pepperoni", "second_topping" => "sausage"}
puts pizza
```

gives you:

```
first_toppingpepperonisecond_toppingsausage
```

Second, you can use the `length` method on hashes just as you can on arrays. For example:

```
pizza = {"first_topping" => "pepperoni", "second_topping" => "sausage"}
puts pizza.length
```

gives you:

```
2
```

Finally, you can store numbers — and for that matter, all kinds of data — in hashes, not just text. This code:

```
receipts = {"day_one" => 5.03, "day_two" => 15_003.00}
puts receipts["day_one"]
puts receipts["day_two"]
```

gives you:

```
5.03
15003.0
```

Working with Ranges

There's another data construct that you should know about — Ruby *ranges*. In the real world, you encounter all kinds of ranges all the time — Monday through Friday, May through July, 1 to 10, a to z, and so on. Ruby gives you an easy way to specify ranges.

To create a range, use the `..` operator. For example, here's how you might create the range 1 to 4:

```
my_range = 1..4
```

That creates the range 1, 2, 3, 4. A handy way of seeing that is to convert the range into an array using the range's `to_a` method, and then simply print out the result array:

```
my_range = 1..4
puts my_range
```

This gives you:

```
1
2
3
4
```

You can also use the `...` operator, which is the same thing except that the final item in the range is omitted. So this range:

```
my_new_range = 1...4
```

actually gives you 1, 2, 3 — not 1, 2, 3, 4, as you can see by printing it out:

```
my_new_range = 1...4
puts my_new_range
```

Here's what you get:

```
1
2
3
```

Ranges are going to be useful in the next chapter, when you want Ruby to do something over and over again. Using a range, you can specify the data items that you want Ruby to work with in such cases.

Try It Out Using Ranges

To get started using ranges in Ruby, follow these steps:

1. Enter this code in a new file, `ranges.rb`:

```
range = 1..5            #creates 1, 2, 3, 4, 5
puts range.to_a
range = 1...5           #excludes the 5
puts range.to_a
range = "a".."e"        #creates "a", "b", "c", "d", "e"
puts range.to_a
puts range.min          #prints "a"
puts range.max          #prints "e"
range = "alpha".."alphe"
puts range.to_a
```

2. Save the file and run it.

```
C:\rubydev>ruby ranges.rb
1
2
3
4
5
1
2
3
4
a
b
c
d
e
a
e
alpha
alphb
alphc
alphd
alphe
```

The `range min` and `max` methods return the first and last elements in an array:

```
range = "a".."e"        #creates "a", "b", "c", "d", "e"
puts range.min          #prints "a"
puts range.max          #prints "e"
```

This code displays:

```
a
e
```

And note this code:

```
range = "alpha".."alphe"
puts range.to_a
```

which results in:

```
alpha
alphb
alphc
alphd
alphe
```

As you can see, Ruby attempts to be smart about how it constructs ranges for you.

Here's a final note: You have to create ranges in ascending sequence. For example, whereas this will work:

```
(1..10).to_a
```

this will give you an empty array:

```
(10..1).to_a
```

Summary

In this chapter, you got the basics of Ruby down, including how to:

- Install Ruby and Rails, simply by downloading and uncompressing binary files.
- Use the `puts` method to display messages.
- Read the Ruby documentation using the `ri` command.
- Create variables in Ruby just by using them. You name Ruby variables by starting with a lowercase letter, a to z, or an underscore (_) followed by any number of lowercase letters, uppercase letters, digits, or underscores.
- Name and use constants in Ruby (the names of constants — whose values are not supposed to be changed — start with a capital letter).

- ❑ Use Ruby operators.
- ❑ Use arrays and hashes to group data together into handy data constructs.
- ❑ Construct Ruby ranges using the `..` and `...` operators.

In the next chapter, you learn to work with conditionals, loops, methods, and much more. Before you move on, though, work through the following exercises to test your understanding of the material covered in this chapter. The solutions to these exercises are in Appendix A.

Exercises

1. Use a negative index to access the third element in this array: `array = [1, 2, 3, 4, 5, 6, 7, 8]`.
2. Construct a hash that will act the same as the array introduced in the previous exercise, as far as the `[]` operator is concerned.
3. Use a range to create the array introduced in exercise 1.