

Programming for Kids

Ruby and Mac Edition



Peter Armstrong

Programming for Kids

Ruby and Mac Edition

Peter Armstrong

This book is for sale at <http://leanpub.com/programmingforkids>

This version was published on 2018-11-04



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2012 - 2018 Peter Armstrong

Tweet This Book!

Please help Peter Armstrong by spreading the word about this book on [Twitter!](#)

The suggested hashtag for this book is [#programmingforkids](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#programmingforkids](#)

For Evan, literally

Contents

Introduction	i
Chapter 1: Hello, Command Line!	1
Exercises	11
Chapter 2: Hello World!	12
Exercises	14
Chapter 3: Hello, irb!	16
Exercises	17
Chapter 4: Functions	18
Exercises	21
Chapter 5: Programs	23
Exercises	25
Chapter 6: Functions, Part 2	26
Exercises	31
Chapter 7: Characters and Strings	33
Exercises	37
Chapter 8: Variables	39
Exercises	42
Chapter 9: Command Line Input	44

CONTENTS

Exercises	48
Chapter 10: Arrays and Looping	50
Exercises	56
Chapter 11: Files	58
Exercises	64
Chapter 12: Booleans, If and While	65
Exercises	79
Chapter 13: The Spelt Project	80
Section 1: Spelling a Word	80
Section 2: Finishing Spelt	89
Section 3: Enhancing Spelt	94
Exercises	98
For Parents	105
About the Author	107
About the Cover	108
About Leanpub	109
Answers to Exercises	110
Chapter 1	110
Chapter 2	112
Chapter 3	113
Chapter 4	114
Chapter 5	117
Chapter 6	118
Chapter 7	121
Chapter 8	123
Chapter 9	124
Chapter 10	125
Chapter 11	128

CONTENTS

Chapter 12 130

Chapter 13 132

Introduction

This book will teach you how to write computer programs!

You will need to use a Mac computer to follow along. The programs are short, so you can type them all in yourself. This is true even if you can't type well.

This book has a bunch of small chapters. Each chapter is about one idea.

At the end of every chapter, there will be exercises for you to do. It is really important that you do all of them! Doing the exercises ensures that you know the material in the chapter. The answers for all the exercises are in the back of the book, and there are links to the answers for all the exercises.

If your parents want to find out more about this book, they can read it with you. Or, they can just read the [For Parents](#) section at the back of the book.

If you're going to use their computer, they might want to sit with you. That's fine. They'll learn something too!

If you are following along on your Mac, you want to read the PDF file that your parents downloaded. If you double-click on the PDF file, it will open in a program called Preview. You will be switching back and forth between reading this book in Preview and typing stuff in Terminal. If you have clicked on the Preview to switch pages, you will need to click on the Terminal window to focus it **before** typing any commands.

If switching between Preview and Terminal gets annoying, there are a couple things you can do. If your parents have a printer, they can print the book. Or, if they have an iPad, they can download the EPUB file, add it to iTunes and then sync it to the iPad.

Let's get started!

Chapter 1: Hello, Command Line!

This chapter is about the command line!

This book is for kids like you who use a Mac computer, either at home or at school.

Normally you use a Mac by clicking on stuff with a mouse. In this chapter you will learn a different way to use a Mac. It is called the command line, and you just use the keyboard.

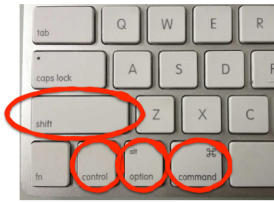
Once upon a time, about 30 years ago, all computers had was a command line.

Even though the command line is simple, it is also very powerful. If you know how to use it, you will be like a wizard who can type strange spells and make your computer do amazing things.

Relax, it's not scary. If you get something wrong, you won't break anything!

First we need to learn the basics.

You already know what a keyboard is. But there may be some keys you may not have used before. These are the Command, Option, Control and Shift keys.

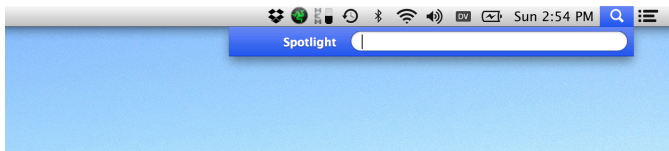


The Command, Option, Control and Shift Keys

We are going to start a program called Terminal. Terminal is what lets you use the command line on your Mac.

To start Terminal, hold down the Command key and press the Space bar. (A shorter way of saying this is to say “type **Command + Space**”. So, that’s what I will say from now on.)

Typing **Command + Space** opens a program called Spotlight in the top right corner of your screen.

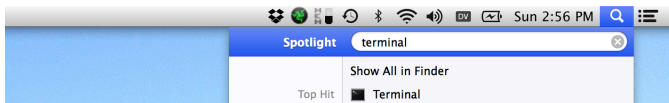


Spotlight

Spotlight lets you type the names of programs to run.

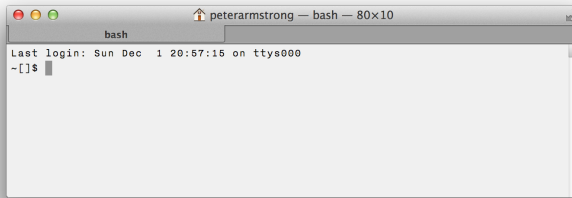
We are going to run Terminal. So, type **terminal** in Spotlight and hit the Enter key.

You might see it show up in a drop down list after you type **term**. If so, you can just click on that choice instead of finishing typing **terminal**.



Running Terminal

You will see Terminal, which will look something like this.



Terminal

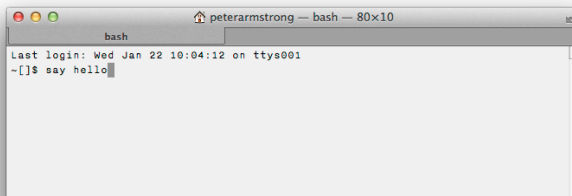
Terminal is the command line of the Mac.

(Don't worry about the `[]$` stuff at the beginning of the line. That's just what the "prompt" looks like on my c

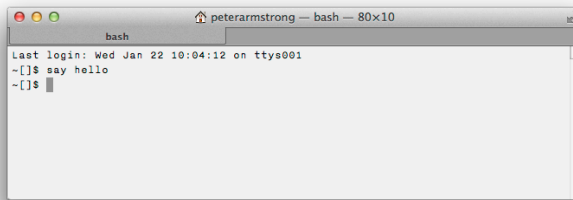
Before we get started, let's have the command line say hello to us.

To do this, we're going to run the **say** program.

Type **say hello** in Terminal.



Then, press the **Enter** key to run the command.

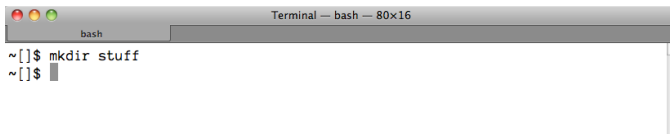
A screenshot of a macOS Terminal window. The title bar reads "peterarmstrong — bash — 80x10". The window contains the following text: "Last login: Wed Jan 22 10:04:12 on ttys001", "~[]\$ say hello", and "~[]\$ " with a cursor. The window has standard macOS window controls (red, yellow, green buttons) in the top-left corner.

```
bash
Last login: Wed Jan 22 10:04:12 on ttys001
~[]$ say hello
~[]$
```

Hopefully your Mac said hello to you! If not, make sure the volume is on and try that again.

By the way, after entering any command in Terminal you need to hit Enter to run it.

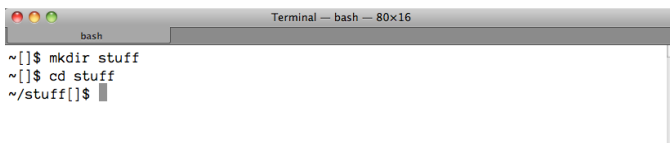
Now, let's start by making a folder for all the stuff we are going to do in this book. Type **mkdir stuff** in Terminal and hit Enter.

A screenshot of a macOS Terminal window. The title bar reads "Terminal — bash — 80x16". The window contains the following text: "~[]\$ mkdir stuff" and "~[]\$ " with a cursor. The window has standard macOS window controls in the top-left corner.

```
Terminal — bash — 80x16
~[]$ mkdir stuff
~[]$
```

The command **mkdir** is said “make dir” and stands for “make directory”. Directory is another word for folder. So, you just made a folder called **stuff**.

Next, we are going to go into that folder. Type **cd stuff** in Terminal and hit Enter.

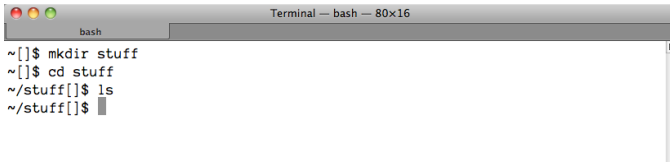
A screenshot of a macOS Terminal window. The title bar reads "Terminal — bash — 80x16". The window contains the following text: "~[]\$ mkdir stuff", "~[]\$ cd stuff", and "~/stuff[]\$ " with a cursor. The window has standard macOS window controls in the top-left corner.

```
Terminal — bash — 80x16
~[]$ mkdir stuff
~[]$ cd stuff
~/stuff[]$
```

The **cd** command stands for “change directory”. So, you are changing into the directory called **stuff** that you just made.

Let's look inside this directory. In Terminal, type **ls** (a lowercase L

and a lowercase S) and hit Enter.

A screenshot of a macOS Terminal window titled "Terminal — bash — 80x16". The window shows a bash shell prompt. The user has entered the following commands: `mkdir stuff`, `cd stuff`, and `ls`. The output of `ls` is empty, indicating that the directory is currently empty. The prompt is now `~/stuff[]$`.

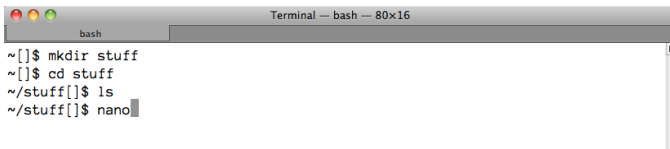
Typing `ls` shows nothing!

The `ls` command means “list”, which means to show the files and folders inside this folder. There is nothing in this folder, since we just made it. So, this is why typing `ls` shows nothing.

Now, let’s change that by creating a file.

We are going to use a program called a text editor to edit the file. A text editor lets you type words in a file, and save the file just like you would save a game. We are just going to use a small text editor called **nano**. Nano means really small, and it’s a good name since **nano** is a really small, simple text editor.

So, in Terminal, type `nano` and hit the Enter key.

A screenshot of a macOS Terminal window titled "Terminal — bash — 80x16". The window shows a bash shell prompt. The user has entered the following commands: `mkdir stuff`, `cd stuff`, `ls`, and `nano`. The output of `ls` is empty. The prompt is now `~/stuff[]$`, and the `nano` command has been entered, with the cursor at the end of the line.

This starts nano. The screen will look like this.

```

Terminal - nano - 80x16
nano
GNU nano 2.0.6      New Buffer

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell

```

In nano, type **hello**. You don't need to hit the Enter key.

```

Terminal - nano - 80x16
nano
GNU nano 2.0.6      New Buffer      Modified
hello

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell

```

You can see that nano is being helpful, by showing you the list of commands you can type by holding down the **control** key and typing a letter. (Nano is showing the **control** key like the caret (^), but it is not **shift + 6**, it is the **control** key.)

```

^G Get Help  ^O WriteOut  ^R Read File
^X Exit      ^J Justify   ^W Where Is

```

So, to save the file, hold the **Control** key down and type the **o** key.

Nano will ask you what you want to call the file you are saving, by showing text saying “File Name to Write”.

```

Terminal -- nano -- 80x16
nano
GNU nano 2.0.6      New Buffer      Modified

hello

File Name to Write:
^G Get Help      ^T To Files      ^M Mac Format     ^P Prepend
^C Cancel        ^D DOS Format    ^A Append        ^B Backup File

```

Type `hello.txt` and press the Enter key.

```

Terminal -- nano -- 80x16
nano
GNU nano 2.0.6      New Buffer      Modified

hello

File Name to Write: hello.txt
^G Get Help      ^T To Files      ^M Mac Format     ^P Prepend
^C Cancel        ^D DOS Format    ^A Append        ^B Backup File

```

Nano will save the file and tell you it was one line long, by saying “Wrote 1 line”.

```

Terminal -- nano -- 80x16
nano
GNU nano 2.0.6      File: hello.txt

hello

[ Wrote 1 line ]
^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^N Next Page     ^U UnCut Text    ^T To Spell

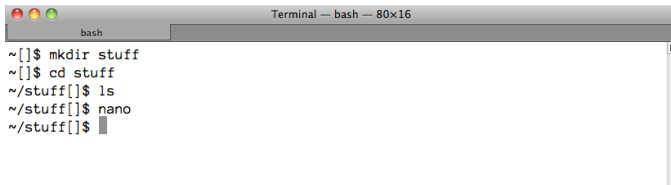
```

(If you had hit Enter after typing “hello” earlier, it will say “Wrote

2 lines”. That’s fine, don’t worry.)

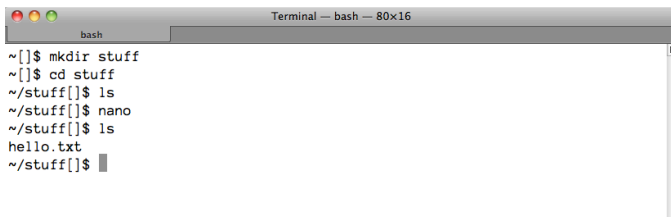
Now that we’ve saved our file, let’s quit nano by holding down the Control key and typing **x**. (A shorter way of saying this is to say “type **Control + x**”. So, that’s what I will say from now on.)

You will be back at the command line inside Terminal.

A screenshot of a macOS Terminal window titled "Terminal — bash — 80x16". The prompt is "bash". The user has entered the following commands: `mkdir stuff`, `cd stuff`, `ls`, and `nano`. The prompt is now `~/stuff[]$` with a cursor.

```
bash
~[]$ mkdir stuff
~[]$ cd stuff
~/stuff[]$ ls
~/stuff[]$ nano
~/stuff[]$
```

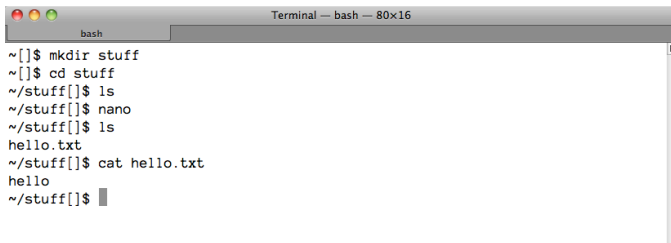
Let’s look inside our **stuff** folder again. Remember last time we typed **ls** we saw it was empty. Type **ls** in Terminal again. You will see the **hello.txt** file you created.

A screenshot of a macOS Terminal window titled "Terminal — bash — 80x16". The prompt is "bash". The user has entered the following commands: `mkdir stuff`, `cd stuff`, `ls`, `nano`, `ls`, and `hello.txt`. The prompt is now `~/stuff[]$` with a cursor.

```
bash
~[]$ mkdir stuff
~[]$ cd stuff
~/stuff[]$ ls
~/stuff[]$ nano
~/stuff[]$ ls
hello.txt
~/stuff[]$
```

Let’s look inside this file. Type **cat hello.txt** in Terminal and hit Enter.

Here, **cat** is not an animal. Instead, it is a command that shows you what is in your file. Since we typed the word **hello**, this is what you will see.

A screenshot of a macOS Terminal window. The title bar reads "Terminal — bash — 80x16". The prompt is "bash". The user has entered a series of commands: "mkdir stuff", "cd stuff", "ls", "nano", "ls", "cat hello.txt", and "hello". The output shows the directory structure and the contents of the file.

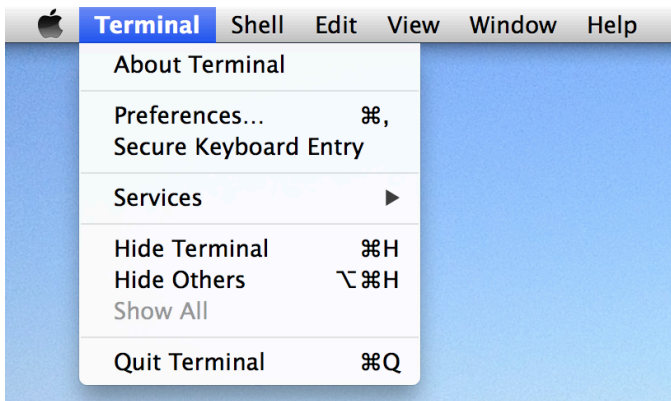
```
~[]$ mkdir stuff
~[]$ cd stuff
~/stuff[]$ ls
~/stuff[]$ nano
~/stuff[]$ ls
hello.txt
~/stuff[]$ cat hello.txt
hello
~/stuff[]$
```

So, the `hello.txt` file had “hello” inside it, so when you typed `cat hello.txt` you saw “hello”.

You now know how to use some of the basic commands of the command line! You learned a bunch of strange commands, things like `mkdir`, `cd`, `ls` and `cat`. The neat thing is that these commands are actually just programs that were written by other people!

In the next chapter, **you** will create **your** first program, which **you** will also be able to run from the command line!

Finally, we are going to quit Terminal. Type **Command + Q** or choosing Quit Terminal from the Terminal menu.



By the way, if you ever get something on the command line really wrong and you don’t know what to do next, you can always just quit Terminal and open it up again.

Make sure you do the Exercises on the next page before continuing!

Exercises

1. Start Terminal and `cd` into the `stuff` directory.
2. Use `nano` to make a file named `hooray.txt` that contains the text “hooray”. Quit `nano` when you’re done.
3. Use the `cat` program to see the contents of your `hooray.txt` file.
4. Quit Terminal.

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 2: Hello World!

Let's write our first computer program!

Start Terminal, `cd` into the `stuff` directory and run `nano`. (If that did not make sense, see the [Chapter 1 Exercises](#).)

We're going to create a really simple program. Type `puts "Hello World!"` in nano.

There's no need to hit the Enter key. I'll talk more about what `puts` means later.

To type a quote mark like `"`, hold the shift key and type the `'` key. There's no difference between the start and end quote marks.



That's it!

Type **Control** + **o** to save the file.

Nano will ask you what you want to call the file you are saving, by showing text saying "File Name to Write:"

Type `hello.rb` and press the **Enter** key. As you type the file name, you will see it shown in “File Name to Write” area near the bottom of Terminal.

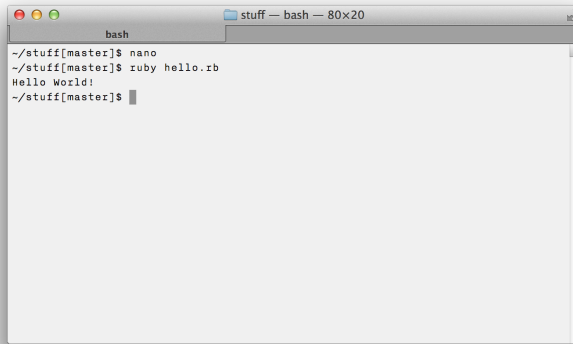


Nano will save the file and tell you it was one line long, by saying “Wrote 1 line”. (If you hit the Enter key, it will say Wrote 2 lines. That’s fine too.)

Now that we’ve saved our file, type **Control + x** to quit nano.

Let’s run our program!

In Terminal, type `ruby hello.rb` and hit **Enter**. You need a space between `ruby` and `hello.rb`.



```
stuff — bash — 80x20
bash
~/.stuff[master]$ nano
~/.stuff[master]$ ruby hello.rb
Hello World!
~/.stuff[master]$
```

Your First Computer Program!

Congratulations, you’ve written your first computer program! As you just saw, `puts` printed something to the screen.

But, what was that strange word “ruby”?

Ruby is a programming language. You speak English, but there are lots of other languages that people speak. Similarly, there are lots of different languages you can use to tell a computer what to do. Ruby is one of the easier ones to use, and your Mac comes with Ruby already installed.

The program you wrote was a Ruby program! The file extension (the stuff after the `.` in the filename) for Ruby programs is `rb`, so we named the file `hello.rb`.

In the next chapter, we will play with Ruby on its own command line!

Exercises

1. Write and run a Ruby program named `hi.rb` that prints `Hi!`.

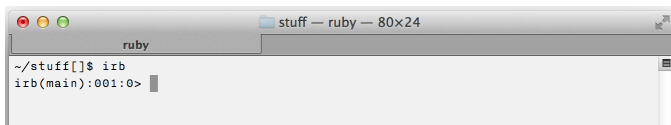
You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 3: Hello, `irb`!

Ensure that Terminal is running, and that you are in the `stuff` directory. (If that did not make sense, see the [Chapter 1 Exercises](#).)

Ruby is a programming language. It also has its own command line called `irb`, which stands for Interactive Ruby Shell.

To start `irb`, just type `irb` in Terminal.

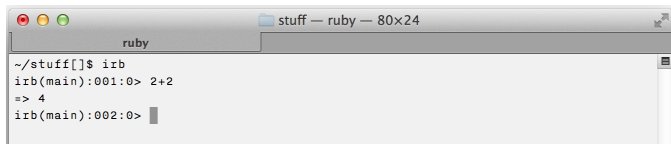
A screenshot of a macOS Terminal window titled "stuff — ruby — 80x24". The window shows the prompt "~/stuff[]\$ irb" and the subsequent prompt "irb(main):001:0>".

```
~/stuff[]$ irb
irb(main):001:0>
```

Note that `irb` has its own prompt. On my Mac, it looks like `irb(main):001:0>`. If you have an older version of Ruby on an older Mac, your `irb` prompt might just look like `>>`. The `irb` prompt just lets you know that `irb` is waiting for you to type a command. We'll ignore the other details that it shows.

Anyway, let's get `irb` to do your math homework!

Type `2+2` in `irb` and hit Enter.

A screenshot of a macOS Terminal window titled "stuff — ruby — 80x24". The window shows the prompt "~/stuff[]\$ irb", the prompt "irb(main):001:0> 2+2", the output "=> 4", and the subsequent prompt "irb(main):002:0>".

```
~/stuff[]$ irb
irb(main):001:0> 2+2
=> 4
irb(main):002:0>
```

For your parents' sake: you should just use `irb` to check your answers when **you** do your math homework! :)

To do multiplication, you use `*` (**shift** + **8**) not `x`.

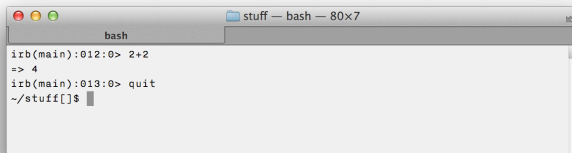
Try `3*2` in `irb`.

```
1  irb> 3*2
2  => 6
```

To do division, you use `/` (which is beside the right `shift` key).

```
1  irb> 6/2
2  => 3
```

To quit `irb`, type `quit`.

A screenshot of a terminal window titled 'stuff - bash - 80x7'. The window shows an interactive Ruby (irb) session. The prompt is 'irb(main):012:0>' and the user has entered '2+2'. The output is '=> 4'. The prompt is now 'irb(main):013:0>' and the user has entered 'quit'. The terminal shows the prompt '~/.stuff[]\$' with a cursor. The window has standard macOS window controls (red, yellow, green buttons) in the top left corner.

```
bash
irb(main):012:0> 2+2
=> 4
irb(main):013:0> quit
~/.stuff[]$
```

One of the reasons that Ruby is an easy programming language to learn is that it has `irb`, so you can experiment interactively really quickly.

In the next chapter, we're going to learn what functions are!

Exercises

1. Have `irb` do `3-2`. The minus key, `-`, is beside the `0`.
2. Have `irb` do `3.0/2`. What do you think the answer is?

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 4: Functions

If all we could do with programming languages was add numbers and say hello to ourselves, we wouldn't have anything more than a lousy calculator!

Let's do something more interesting!

We're going to define a function. Most computer programs are made up of a lot of functions. A function is something that does some stuff and returns a value. The return value is the answer.

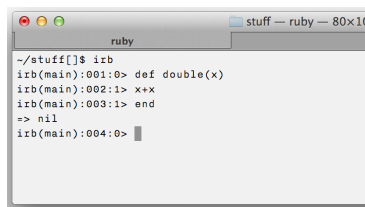
Start `irb` again by typing `irb` in Terminal. (If Terminal isn't running, start it using Spotlight, and go into your `stuff` directory by typing `cd stuff`.)

Let's make a function that knows how to double any number.

Type this in `irb`, hitting Enter at the end of each line:

```
1 def double(x)
2   x+x
3 end
```

To type `(` you hold shift and type `9`, and to type `)` you hold shift and type `0`.

A screenshot of a Ruby IRB terminal window. The window title is "stuff -- ruby -- 80x10". The prompt is "~/stuff[]\$ irb". The first line of code is "def double(x)". The second line is "x+x". The third line is "end". The prompt changes to "irb(main):001:0>". The fourth line is "x+x". The prompt changes to "irb(main):002:1>". The fifth line is "end". The prompt changes to "irb(main):003:1>". The sixth line is "=> nil". The prompt changes to "irb(main):004:0>".

```
~/stuff[]$ irb
irb(main):001:0> def double(x)
irb(main):002:1>   x+x
irb(main):003:1> end
=> nil
irb(main):004:0>
```

This function is only 3 lines long, but there's lots going on here.

The word **def** is how we start defining a function in Ruby. (Programmers don't like to type long words, so **def** is short for define.)

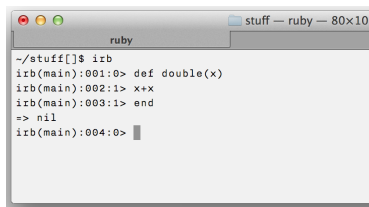
The inputs to the function are put in brackets, which are the **()** things. In this example, we have one input. Since we don't like typing, we will name this input **x**.

The name of the function is called **double**.

One way to double a number is to add it to itself. To double 2, you say $2 + 2$, which is 4. To double 3, you say $3 + 3$, which is 6. What we are doing is making a function which can double any number. We don't know what that number is going to be in advance, so we just called it **x**. So, to double **x**, we just say $x + x$.

Finally, the last line is **end**, since that is how you end a function definition in Ruby.

When you typed this in, you saw that defining the function printed **=> nil**.

A screenshot of a terminal window titled 'stuff - ruby - 80x10'. The prompt is '~/\$ irb'. The user enters 'def double(x)', and the prompt changes to 'irb(main):001:0>'. The user enters 'x+x', and the prompt changes to 'irb(main):002:1>'. The user enters 'end', and the prompt changes to 'irb(main):003:1>'. The terminal then prints '=> nil' and the prompt changes to 'irb(main):004:0>'.

```
~/stuff[]$ irb
irb(main):001:0> def double(x)
irb(main):002:1> x+x
irb(main):003:1> end
=> nil
irb(main):004:0>
```

All this means is that return value of actually defining the function is nothing. In Ruby, we call nothing **nil**.

Anyway, we made a function that knows how to double a number, by adding it to itself. Let's see how it works.

I'm going to show the **irb** prompt as **irb>** from now on. This way, you will see what you type as input to **irb** (at the **irb** prompt) and what **irb** prints as output (with a **=>** arrow).

Anyway, type **double 2** in **irb**. When you follow along, you do not actually type **irb>**. Just type the stuff after it.

```
1  irb> double 2
2  => 4
```

2 + 2 is 4, so that looks right.

When you **define** a function you use `def ... end`. When you use the function that you defined, you are said to be **calling** the function. To **call** a function, you just use the name of the function, and any arguments. So `double 2` is a **function call**. You can also put the arguments to the function in brackets, like this: `double(2)`.

Let's try a bigger number!

```
1  irb> double 123
2  => 246
```

123 + 123 is 246, so that looks right too!

Let's make another function. This one will triple numbers!

To triple something, you either add it to itself twice (2+2+2 is 6, and 6 is the triple of 2) or you multiply it by 3. The way you multiply something in Ruby (and in most programming languages) is with `*`.

```
1  irb> def triple(y)
2  irb> y*3
3  irb> end
```

Note there is a space between `def` and `triple`.

Here, we're going to name the input to the function `y`, not `x`. This input is called a parameter.

It doesn't matter what you name the parameter, as long as you use the same name inside the function. So, you could have said this (don't actually type this):

```
1  irb> def triple(dog)
2  irb> dog*3
3  irb> end
```

Also, note that spaces don't usually matter. So, you could also have said this (don't actually type this either):

```
1  irb> def triple(cat)
2  irb> cat + cat + cat
3  irb> end
```

Anyway, let's test our new function! Type `triple 3` in irb, and hit Enter.

```
1  irb> triple 3
2  => 9
```

$3 * 3$ is 9, which is the same thing as $3 + 3 + 3$. So, this looks right!

Let's try a bigger number.

```
1  irb> triple 10
2  => 30
```

$10 * 3$ is 30, which is the same thing as $10 + 10 + 10$.

We've now made 2 functions! In the Exercises, we'll make 3 more! Leave irb open.

Exercises

1. Write a function called `quad` to quadruple a number by multiplying it by four. Test it afterward.

2. Write a function called `quad2` to quadruple a number by adding it to itself the correct number of times.
3. Write a function called `quad3` to quadruple a number by using the `double` function you made before.

You can see the [answers to the exercises](#), or just continue to the next chapter.

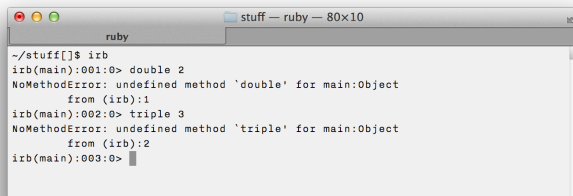
Chapter 5: Programs

If you haven't quit `irb`, quit it now. Then start `irb` again by typing `irb` in Terminal. (If Terminal isn't running, start it using Spotlight, and go into your `stuff` directory by typing `cd stuff`. Then run `irb` by typing `irb`.)

In the last chapter, we made two functions. The `double` function doubled a number, and the `triple` function tripled it.

Let's try running those again.

Type `double 2` in `irb`. Then try `triple 3`. Assuming you quit `irb` and restarted it, you will see something like this.

A screenshot of a Ruby Terminal window titled 'stuff — ruby — 80x10'. The window shows the following text:

```
~/stuff[]$ irb
irb(main):001:0> double 2
NoMethodError: undefined method 'double' for main:Object
    from (irb):1
irb(main):002:0> triple 3
NoMethodError: undefined method 'triple' for main:Object
    from (irb):2
irb(main):003:0>
```

Where Did Our Functions Go?

The functions we made are gone!

The reason for this is that we didn't make them in a program file; we just made them in `irb`. So, when we quit `irb`, they were lost.

Luckily, this is easy enough to fix!

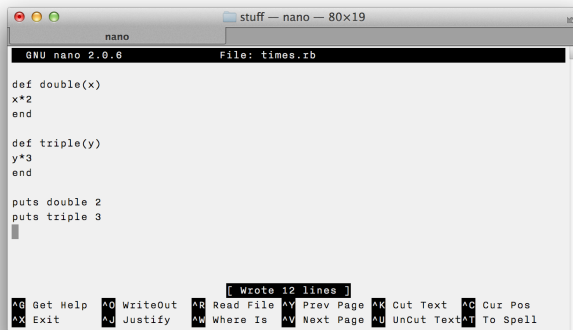
We already learned how to edit text files in `nano`. We will make a new program file which contains our `double` and `triple` functions. We will also make our program file test these functions!

Quit `irb` by typing `quit`. Then, open `nano` by typing `nano` in Terminal.

Type the following Ruby code into the file.

```
1 def double(x)
2   x*2
3 end
4
5 def triple(y)
6   y*3
7 end
8
9 puts double 2
10 puts triple 3
```

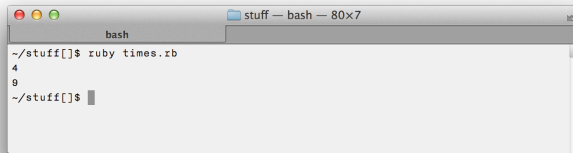
Then, save the file by typing **Control + o**. Type `times.rb` and hit **Enter** to name the file `times.rb`.



Now that you've saved the file, quit nano by typing **Control + x**.

Let's run our program!

Type `ruby times.rb` in Terminal, and hit **Enter**. You will see something like this:

A terminal window titled 'stuff -- bash -- 80x7' with a 'bash' prompt. The user enters '~/.stuff[]\$ ruby times.rb'. The terminal displays the output '4' on the next line, followed by '9' on the line after. The prompt '~/.stuff[]\$' is shown again at the bottom.

```
bash
~/.stuff[]$ ruby times.rb
4
9
~/.stuff[]$
```

The first line shows the number 4. This is the result of calling the **double** function with the number 2, and printing the return value with **puts**.

The second line shows the number 9. This is the result of calling the **triple** function with the number 3, and printing the return value with **puts**.

What do you think would have happened if you hadn't said **puts** in the last 2 lines of your program?

We'll learn that in the next chapter...

Exercises

1. Add a **quadruple** function to your program in your `times.rb` file. Also, test it afterward, in the `times.rb` file, by quadrupling 4.
4. To open `times.rb` in nano, type `nano times.rb`.

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 6: Functions, Part 2

Way back in the first chapter about functions, I said that a function is something that does some stuff and returns a value. The return value is the answer.

Open a Terminal window, `cd` into the `stuff` directory, and open `irb`.

(By the way, you just understood that sentence. Think about how far you've come already!)

Let's look more closely at what it means to return a value.

First, we're going to load the `times.rb` program into `irb`. By doing this, we will import our functions from the `times.rb` program into the running `irb`, so that we can call those functions again!

The way we do this is by typing `load 'times.rb'` in `irb`. (Make sure to type the single quotes, which are beside the Enter key.) You will see something like this:

```
1  irb> load 'times.rb'
2  4
3  9
4  16
5  => true
```

If you did not follow along with the last chapter exercises, you will just see:

```
1  irb> load 'times.rb'
2  4
3  9
4  => true
```

If you do not see this, and instead see a `LoadError`, you either started `irb` from somewhere other than the `stuff` directory, or you didn't follow along with the last chapter or its exercises. If that happened, quit `irb` with `quit` and `cd` into your `stuff` directory. Make the `times.rb` file if necessary, then continue.

Anyway, now that you're here, I'm going to assume you've followed along!

When you load the `times.rb` file into `irb`, you see the same output that you saw when you ran the `times.rb` program on the command line using `ruby times.rb` in the last chapter.

This is because when you use the `load` command, you actually run the program!

You see 4 (which is the double of 2) and 9 (which is the triple of 3).

But what is that `=> true` thing?

That is the return value of running the program. When we do things in `irb`, it helpfully prints the return value.

So, for example, if you type `double 2` in `irb`, you see 4:

```
1  irb> double 2
2  => 4
```

Now, in the `times.rb` program we said `puts double 2` to print the double of 2. Let's try this in `irb`:

```
1  irb> puts double 2
2  4
3  => nil
```

So, here we see the same 4 that got printed when we ran `ruby times.rb` from the command line. But we also see `=> nil`. This is the return value of calling `puts double 2`.

This is the secret!

It turns out that `puts` is just a function!

The `puts` function is just like the `double` and `triple` functions you made earlier! The return value of `puts` is `nil`. So, when you call `puts` in irb, it prints `=> nil`. But when you call the `puts` function normally in a program, Ruby just prints what `puts` tells it to, and doesn't print the return value of `puts`. That's good, since `nil` is pretty boring!

So, let's do some fun stuff! You already know that typing `double 2` in irb returns 4.

```
1  irb> double 2
2  => 4
```

What do you think happens if you type `double double 2` in irb?

Well, irb is going to print the return value of all this. But what is it?

```
1  irb> double double 2
2  => 8
```

Is that a surprise?

Let's figure out how this works.

What do you think happens if you type `double 4` in irb?

```
1  irb> double 4
2  => 8
```

This makes sense, since 4 times 2 is 8.

But you know what?

2 times 2 is 4.

So, calling `double 2` is going to return 4.

And then, we are going to call `double` with this return value.

So, `double double 2` is the same thing as `double 4` which, as we saw, is 8.

Now, this can get confusing.

So, one thing you can do is put things in brackets, to make it clearer what is going on.

Try typing `double(double(2))` in irb.

To type (you type `shift + 9`, and to type) you type `shift + 0`.

```
1  irb> double(double(2))
2  => 8
```

So, we got 8 again. But the brackets show that we are first doing `double(2)`, and then we are putting that return value into another call to `double()`.

What do you think is returned if you do `triple(double(2))`?

```
1  irb> triple(double(2))
2  => 12
```

Do you see? 2 times 2 is 4, so `double(2)` will return 4. This 4 is then passed into the `triple` function. And 3 times 4 is 12. So, `triple(4)` will return 12.

That's exactly what happened!

But what do you think happens if you try typing `double(triple(2))`?

```
1  irb> double(triple(2))  
2  => 12
```

It's also 12!

This is because 3 times 2 is 6, so `triple(2)` is 6. And 2 times 6 is 12, so `double(6)` is 12.

This is actually a rule about multiplication: the order does not matter. But you'll learn that in school in an older grade! (And, when you do, you can tell your teacher that you learned that from this book already!)

By the way, before we stop this chapter, I'm getting tired of writing times as "times". So, I'm going to start writing times as `*` instead of `times` or `x`. The word `times` is too annoying to type, and the letter `x` could get confused for a function parameter (like we saw before) or variable (like we will see).

We programmers are really lazy!

One more thing before we stop this chapter:

What do you think this returns?

```
triple(double(triple(2)))
```

If that's too many brackets, you can also type that as:

```
triple double triple 2
```

You don't need the brackets, since Ruby figures out the right thing to do here!

The way this works is as follows:

1. Ruby figures out that `triple(2)` is 6, since $3*2 = 6$.
2. Ruby figures out that `double(6)` is 12, since $2*6 = 12$. So, `double(triple(2))` is 12.
3. Ruby figures out that `triple(12)` is 36, since $3*12 = 36$. So, `triple(double(triple(2)))` is 36.

If you look at these steps, this is the order of the function calls and return values:

```
1 triple(double(triple(2)))
2 triple(double(6))
3 triple(12)
4 36
```

So, the return value is **36**.

Phew! That took a long time for us to do! Luckily, this is the type of thing that computers are really, really good at doing.

Exercises

Setup:

These exercises assume that you did the exercises from the last chapter where you made a `quadruple` function in the `times.rb` file. If you did not do that, do it now. If you had to quit `irb`, then when you restart it you need to remember to load the `times.rb` file by typing `load 'times.rb'` in `irb`.

1. In `irb`, write and test a new `times6` function that uses a combination of the `double` and `triple` functions to return 6 times the number it was given.
2. In `irb`, write and test a new `times12` function that uses a combination of the `double` and `triple` functions to return 12 times the number it was given.
3. In `irb`, write and test a new `times12b` function that uses the new `times6` function and some other function to return 12 times the number it was given.
4. Quit `irb`, then use `nano` to open the `times.rb` file with `nano times.rb`. Use `nano` to add `times6` and `times12` functions to

your `times.rb` file. You can either just return `x*6` and `x*12`, or you can use whatever combination of the `double`, `triple`, `quadruple` and `times6` functions you want. Also, test your functions in the `times.rb` file by doing `6 * 6` (which is 36) and `12 * 12` (which is 144).

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 7: Characters and Strings

So far, we've done a lot of stuff with numbers. In this chapter, we will look at how to do stuff with words.

In Ruby, and in most programming languages, words are handled differently from numbers.

A word can be thought of as a sequence of letters. By sequence, I mean a bunch of letters in a row. For example, the word “word” is a sequence of 4 letters, **w**, **o**, **r** and **d**.

A number can be thought of as a sequence of digits. For example, the number 12 is a sequence of two digits, **1** and **2**.

More generally, anything you can type on your keyboard, whether it is a letter, a number or other stuff like spaces, symbols like (!@#\$%^&*) or punctuation like ,.- can be thought of as a **character**.

So, a word or a number or a whole sentence is just a sequence of characters.

But, “sequence of characters” is annoying to type and say, so we just say **string** instead. If you think about beads, you can arrange them on a string to make a necklace. But in a computer, you can arrange characters in a string to make words, sentences and other stuff.

In this chapter, we're going to play with characters and strings.

Let's open up `irb`. (As always, if you are starting here, you start Terminal, type `cd stuff` to go into the `stuff` directory, and then type `irb` to start `irb`.)

First, let's just say “hello” in `irb` by typing the string `"hello"` in quotes.

```
1  irb> "hello"  
2  => "hello"
```

As always, irb prints what the return value is. The return value of “hello” is “hello”. So, saying “hello” returns “hello”.

Let’s do something more interesting. Type `"hello".length` in irb.

```
1  irb> "hello".length  
2  => 5
```

Cool! The length of “hello” is 5, since there are 5 letters in the word hello. Or, in programmer terms, there are 5 characters in the string “hello”.

Note that there are at least 2 ways to get the above example wrong.

If you forget the quotes, you’ll see something like this:

```
1  irb(main):006:0> hello.length  
2  NameError: undefined local variable or  
3  method `hello' for main:Object  
4      from (irb):6  
5      from :0
```

And, if you only do the first quote, irb will just sit there.

```
1  irb(main):007:0> "hello.length
```

The reason is that irb is waiting for the string to be finished. So, to fix that, just type a “ on a line by itself and hit Enter. Then, you will be back at the irb prompt and you can try the example again.

```
1  irb> "hello".length
2  irb> " "
3  => "hello.length\n"
4  irb> "hello".length
5  => 5
```

If you are wondering, `length` is actually a function. In Ruby and in many programming languages, functions are organized in things called “objects”. And in Ruby, a string is actually an object as well as a sequence of characters. We’ll talk more about objects later.

Let’s try another function!

```
1  irb> "hello".reverse
2  => "olleh"
```

In Ruby, strings know how to reverse themselves. So, `olleh` is `hello` backwards.

What happens if we do this the other way?

```
1  irb> "olleh".reverse
2  => "hello"
3  irb> "hello".reverse.reverse
4  => "hello"
```

Of course, this works with more than just words. Let’s see how many characters are in this next sentence!

```
1  irb> "This is 4 words".length
2  => 15
```

So, there are 15 characters in that sentence. Remember that the count of characters includes everything inside the quotes, including the spaces.

Finally, we can put things inside a string. This is known by the fancy word “interpolation”, which basically means to stick stuff in the middle. So, let’s say we want `irb` to tell us what `2+2` is, but in a nice sentence.

Type `"2 + 2 is #{2+2}."` in `irb`. To type the `#` character, hold the `shift` key down and press `3`. To type the `{` character, hold the `shift` key down and type `[` (by the `Enter` key). To type the `}` character, hold the `shift` key down and type `]` (also by the `Enter` key).

```
1  irb> "2 + 2 is #{2+2}."
2  => "2 + 2 is 4"
```

One more thing: remember the `say` command we used in chapter 1? That’s actually a program.

And in Ruby, we can run programs by putting them in backticks. The backtick character

```
1  `
```

is above the `Tab` key.

So, let’s have `irb` use the `say` program to tell us what `2 + 2` is. Note that there is a backtick at the beginning and at the end of the `irb` command.

```
1  irb> `say "2 plus 2 is #{2+2}"`
2  => ""
```

By the way, note that the return value of calling `say` is nothing. So, instead of seeing `"2 plus 2 is 4"`, we see `""`. That is a string with

nothing in it. We call a string with no characters in it an “empty string”.

In the rest of the book, we’ll do lots with strings. But for now, let’s take a break! Quit `irb` by typing `quit`.

```
1  irb> quit
2  ~/stuff[]$
```

Exercises

So far, the output of the `times.rb` program we have been building is pretty simple.

When we run it, we see the following:

```
1  ~/stuff[]$ ruby times.rb
2  4
3  9
4  16
5  36
6  144
7  ~/stuff[]$
```

What do the numbers mean? We know, but anyone looking at it will not know.

It would be a lot nicer if we saw this:

```
1 ~/stuff[]$ ruby times.rb
2 2 * 2 is 4
3 3 * 3 is 9
4 4 * 4 is 16
5 6 * 6 is 36
6 12 * 12 is 144
7 ~/stuff[]$
```

1. Use nano to change the code in the times.rb file to show what is being multiplied. You do not need to modify the functions themselves, just the uses of the functions. (In other words, only change every line with a puts in it.) Remember to modify every line that has a puts function. Run the times.rb program with `ruby times.rb` to check.
2. Use irb to find out what your name is backwards. Remember to start irb!

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 8: Variables

In the last chapter, we were writing stuff like `"hello".reverse` and `"olleh".reverse`. That was a bit annoying to type, wasn't it?

Let's fix this by making a **variable**. A variable is just a container for something. Think of it like a box or a bucket that can hold either a number, or a string, or anything you choose.

Start `irb` and try the following (remember the quotes!):

```
1  irb> x = "hello"
2  => "hello"
```

We start by creating a variable called `x` that holds the string `"hello"`. If you imagine a box, we put the string `hello` in the box.

Let's check what's in the box. Remember that `irb` prints the value of what it evaluates. (Evaluate is a complicated idea, but basically it means to check the value. For a variable, the value is the contents of the variable. You might remember that earlier we saw that for a function call, the value is the return value from the function.)

So, let's just say `x` in `irb`:

```
1  irb> x
2  => "hello"
```

This prints `hello`, since `x` has the value `"hello"`. This is the same thing that `irb` prints if we just type `"hello"`:


```
1  irb> "hello"  
2  => "hello"
```

So, `x` has the string “hello” in it. Let’s see what happens when we call `x`’s `reverse` function:

```
1  irb> x.reverse  
2  => "olleh"
```

Why does `x` have a `reverse` function? Simple, it holds the string “hello”, and strings have lots of functions, one of which is `reverse`.

Instead of just printing the value of `x.reverse`, let’s store it in a new variable called `y`.

```
1  irb> y = x.reverse  
2  => "olleh"
```

So, the result of `x.reverse` is “olleh”, which is assigned to `y`. The value returned by the assignment is “olleh”, so that’s what `irb` prints.

Let’s check that `y` really holds `olleh`, by asking `irb` to evaluate `y`:

```
1  irb> y  
2  => "olleh"
```

What happens if we reverse `y`?

```
1  irb> y.reverse  
2  => "hello"
```

Now, variables aren’t just for holding strings. You can put anything in a variable. Let’s put a number in a new variable called `i`:

```
1  irb> i = 1
2  => 1
3  irb> i
4  => 1
```

So, `i` has the number 1 in it.

Let's make a new variable `j` that has the value of `i` plus 1:

```
1  irb> j = i + 1
2  => 2
3  irb> j
4  => 2
```

So, at this point `x` is "hello", `y` is "olleh", `i` is 1 and `j` is 2.

What do you think happens if we try this:

```
1  irb> i + j
```

Well, `1 + 2` is 3, so you'd expect `irb` to say 3, right? That's what happens:

```
1  irb> i + j
2  => 3
```

Now, how about this:

```
1  irb> x + y
```

Well, `x` is "hello" and `y` is "olleh". What does it mean to add strings together?

In Ruby, it means to literally stick them together! This is called **concatenating**, which is pronounced "con cat n ate ing". Concatenating is just a fancy word for sticking stuff together.

Here's what you get:

```
1  irb> x + y
2  => "helloolleh"
```

Now, what if we try this?

```
1  irb> x.length + i
```

Well, `x` is the string “hello” which is of length 5, and `i` is 1, so you’d expect the result to be 6, right?

```
1  irb> x.length + i
2  => 6
```

How about let’s do something silly and try `x + i`. What do you think happens when we try to add a string and a number?

```
1  irb(main):023:0> x + i
2  TypeError: can't convert Fixnum into String
3      from (irb):23:in `+'
4      from (irb):23
5      from :0
```

It doesn’t work! Here, `irb` is complaining that you can’t add a `String` and a `Fixnum` (what Ruby is calling the number 1) together, since they are different types of objects. We will learn more about objects in a few chapters.

Exercises

1. Create a variable called `a` that has the value 4.
2. Create a variable called `b` that has value of 5 times the value of `a`.

3. Create a variable called `c` that has the value “hello world”.
(Remember the quotes!)
4. Create a variable called `d` that has the value of the **length** of the string “hello world”.

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 9: Command Line Input

In the last chapter, we learned about variables. However, without inputs, variables are pretty useless. We were doing stuff like `x = "hello"`, but that was just a way of saving a bit of typing. Our variables weren't varying much at all!

There are two ways that a command line program typically gets input:

1. from the command line
2. from a file

In this chapter, we're going to learn the command line approach.

The inputs we capture will go in variables, and then we'll be able to do fun stuff with them.

We will start by defining a function that reads a line of command line input and prints the reverse of it.

Start `irb` and type the following:

```
1  irb> def reverse
2  irb> s = gets
3  irb> puts s.reverse
4  irb> end
5  => nil
```

The `gets` function reads a line of command line input and returns it. We store the result of `gets` in a variable called `s`. We then use `puts` to print the result of reversing `s`.

Let's try our function! Type `reverse` in `irb`.

```
1  irb> reverse
```

Note that `irb` will just sit there, waiting for a line of input.

Let's give it some input! Type `hello` and hit Enter.

```
1  irb> reverse
2  hello
```

You will see the following:

```
1  irb> reverse
2  hello
3
4  olleh
5  => nil
```

That was cool. But one thing we could improve is that our function is currently returning `nil`.

Functions should return meaningful results, not just print things.

Let's make the `reverse` function return the result of reversing the input it got. We will redefine the `reverse` function like this. You need to type this again to redefine the `reverse` function:

```
1  irb> def reverse
2  irb> s = gets
3  irb> r = s.reverse
4  irb> puts r
5  irb> r
6  irb> end
7  => nil
```

Now, let's test our function again. Type `reverse`, and then type `hello` and Enter.

```
1  irb> reverse
2  hello
3
4  olleh
5  => "\noilleh"
```

So, we returned “olleh” as expected, but we have this strange `\n` thing at the front of it! What’s that?

It is the newline character, which is what is produced on your Mac when you hit Enter. (On Windows computers, pressing Enter actually produces two characters, `\r` and `\n`. This kind of garbage can make programming annoying at times!)

We don’t want the newline in our return value! To get rid of it, we will use the `strip` function that Ruby Strings have. Redefine the `reverse` function again like this:

```
1  irb> def reverse
2  irb> s = gets
3  irb> r = s.reverse.strip
4  irb> puts r
5  irb> r
6  irb> end
7  => nil
```

Here we are calling `strip` on the result of `s.reverse`. We are then assigning the return value to `r`. This type of thing where we call a function on the result of another function is called **chaining**, and it’s why we want our functions to always return meaningful values.

Let’s test our new and improved `reverse` function!

```
1  irb> reverse
2  hello
3  olleh
4  => "olleh"
```

Now we have the return value of “olleh” as we wanted, without the `\n` newline that we stripped.

Let’s try something a bit more interesting. We will build a function that adds 2 numbers that the user specifies on the command line. (To type the underscore character `_` hold down the `shift` key and type the minus key which is beside the 0.)

```
1  irb> def add2
2  irb> puts "First"
3  irb> m = gets.strip.to_i
4  irb> puts "Second"
5  irb> n = gets.strip.to_i
6  irb> m + n
7  irb> end
8  => nil
```

Let’s try this out!

```
1  irb> add2
2  First
3  2
4  Second
5  3
6  => 5
```

2 + 3 is 5, so our `add2` function worked!

One little detail I glossed over is that to get each number, we are chaining `gets.strip.to_i`.

Here’s how this chaining works. Say the user types 2 and hits Enter:

1. To get a line of command line input, we use `gets`. The value of the string is `2\n`. The `gets` function always gets a string. That's what the `s` in `gets` is for!
2. To remove the newline (`\n`), we feed the output of `gets` into `strip`. The value of the string is now `2`.
3. To turn the string into a number, we use `to_i`. This converts the `2` string into the number `2`.

Exercises

Question 1

Write a function called `hi` that asks for the user's name by saying "name?", gets it from the command line and returns "Hello " plus the string. Do not include the newline that you get from the input. Test your function with your name! (For me, the function says "Hello Peter".) It doesn't matter what you call your variable, but if you want to match the answer, call it `n`.

Here's what you type in `irb`. Well, you need to fill in the body of the `hi` function with your code, of course!

```
1  irb> def hi
2  irb> # write your code here
3  irb> end
4  => nil
5  irb> hi
6  name?
7  Peter
8  => "Hello Peter"
```

Question 2

Write a function called `cat` that asks for two strings on the command line and returns the concatenated string together. (Concatenating means to stick the two strings together, and is done with `+`. If you forget how to do this, take a peek back at the previous chapter!) To ask for each string, just print the text “input?”. Do not include the newlines. Test your function with the strings “one” and “two”. The result should be “onetwo”. It doesn’t matter what you call your variables, but if you want to match the answer, call them `a` and `b`.

Here’s what you type in `irb`. Well, you need to fill in the body of the `cat` function with your code, of course!

```
1  irb> def cat
2  irb> # write your code here
3  irb> end
4  => nil
5  irb> cat
6  input?
7  one
8  input?
9  two
10 => "onetwo"
```

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 10: Arrays and Looping

So far in this book, we've been working with single things. We've played with strings like "hello", or variables like `x` that hold numbers like 2 or strings like, well, "hello".

But one of the things computers are really good at is dealing with a bunch of stuff. Say, millions of strings, like all the names in a phone book. (If you've never seen a phone book: Once upon a time, phones were things that were mounted on walls. To dial a number you turned a dial which had holes for your fingers. These phones didn't even have cameras or any apps! There was no Google back then, so to find a phone number you looked it up in a book of phone numbers called a "phone book".)

In this chapter, we're going to learn how to work with **arrays**, which can hold anything from a list of numbers to all the entries in a phone book.

Start `irb`.

We will start by making a new array called `a`, and filling it with some numbers.

To type the square brackets `[` and `]` you hit the keys to the right of the `p` key.

```
1  irb> a = [5,2,4,1,3]
2  => [5, 2, 4, 1, 3]
3  irb> a
4  => [5, 2, 4, 1, 3]
```

Every different thing in the array is called an **element** of the array. This array `a` has 5 elements.

Arrays in Ruby are objects, and they come with some handy built-in functions.

For example, there is the `sort` function, which returns a new array which has the same elements, but sorted.

Let's try this by typing `a.sort`.

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

Now, this does not actually change the array `a` itself! To see that, just type `a` and Enter again. It is unchanged from when we created it.

```
1  irb> a
2  => [5, 2, 4, 1, 3]
```

If we want to modify the `a` array, we can use a function called `sort!` (yes, the exclamation mark `!` is part of the name). Type `a.sort!` in `irb`:

```
1  irb> a.sort!
2  => [1, 2, 3, 4, 5]
```

The return value here is the sorted array, just like before. Now type `a` and Enter again, however:

```
1  irb> a
2  => [1, 2, 3, 4, 5]
```

This time the array `a` is sorted.

In Ruby, functions which modify the thing they operate on are often named with an exclamation mark `!` at the end of the name. Normal

functions which just return a new value but leave the thing they operate on unchanged do not have an exclamation mark at the end of the name. By the way, since we programmers are lazy, we often say “bang” instead of “exclamation mark”. So, the `sort!` function would be called “sort bang”, instead of the normal “sort” function.

So, now we have a nicely sorted array. One thing that is useful to do with arrays is to “loop” over them, processing each element one at a time.

In most programming languages, the way you do this is with a `for` loop or a `while` loop, and we will look at those approaches later. But in Ruby, there are some easy built-in ways to loop over things. These are called blocks. The easiest way to learn blocks is to type an example. To type the `|` character (called a “pipe”), hold the `shift` key and type `\` (above the `Enter` key):

```
1  irb> a.each do |item|
2    irb> puts item
3  irb> end
4  1
5  2
6  3
7  4
8  5
9  => [1, 2, 3, 4, 5]
```

Here, we are processing each element of the array `a`, one at a time. The element we are processing is loaded into the `item`. Just like functions have parameters, blocks have parameters too. The block parameter is like a variable, which is set to the given element we are processing. When we are done, the entire array is returned.

You can also do this on one line with curly braces `{` and `}` instead of `do` and `end`. (But, you can’t use both `do ... end` and `{ ... }` at the same time!)

```
1  irb> a.each {|item| puts item}
2  1
3  2
4  3
5  4
6  5
7  => [1, 2, 3, 4, 5]
```

Also, there's nothing special about the name `item`, it's just the name we chose for our block parameter. We could have called it anything:

```
1  irb> a.each {|i| puts i}
2  1
3  2
4  3
5  4
6  5
7  => [1, 2, 3, 4, 5]
```

Arrays don't just hold numbers, they can hold anything. Let's make a new array called `b` that holds some strings.

```
1  irb> b = ["cake", "is", "tasty"]
2  => ["cake", "is", "tasty"]
```

If we type `b` and Enter, we see it has the three strings in it:

```
1  irb> b
2  => ["cake", "is", "tasty"]
```

Now let's loop over the array, and reverse each of the strings:

```
1  irb> b.each do |item|
2  irb> puts item.reverse
3  irb> end
4  ekac
5  si
6  ytsat
7  => ["cake", "is", "tasty"]
```

Doing this did not modify the original array though. We can check that by typing `b` and Enter again:

```
1  irb> b
2  => ["cake", "is", "tasty"]
```

In these examples of using `each`, we have been just returning the original array.

What if we want to actually return something different?

To do this, we use another function that comes with Ruby arrays, called `map`.

What `map` does is build a new array, which contains the result of doing something to each element of the original array.

Let's see some examples!

Our `a` array looks like this:

```
1  irb> a
2  => [1, 2, 3, 4, 5]
```

Let's use `map` to get a new array, with every element 1 bigger than before:

```
1  irb> a.map {|i| i+1}
2  => [2, 3, 4, 5, 6]
```

Our `b` array looks like this:

```
1  irb> b
2  => ["cake", "is", "tasty"]
```

Let's use `map` to get a new array, with every element reversed:

```
1  irb> b.map {|item| item.reverse}
2  => ["ekac", "si", "ytsat"]
```

In both cases, `a` and `b` are unchanged by calling `map`:

```
1  irb> a
2  => [1, 2, 3, 4, 5]
3  irb> b
4  => ["cake", "is", "tasty"]
```

What if we want to modify them? Guess what? In Ruby, arrays come with a method that does that too! It's called `map!`, since it's a `map` that changes its input, just like how `sort!` was a `sort` that changed its input.

Please follow along with these exactly, since the exercises below assume you did.

Let's add 1 to every element of `a`:


```
1  irb> a.map! {|i| i+1}
2  => [2, 3, 4, 5, 6]
3  irb> a
4  => [2, 3, 4, 5, 6]
```

Let's reverse all the strings in b:

```
1  irb> b.map! {|item| item.reverse}
2  => ["ekac", "si", "ytsat"]
3  irb> b
4  => ["ekac", "si", "ytsat"]
```

Exercises

1. Use `map!` to put `a` back to being `[1,2,3,4,5]`. Verify that `a` is changed afterward by typing `a` and Enter.
2. Use `map!` to put `b` back to being `["cake", "is", "tasty"]`. Verify that `b` is changed afterward by typing `b` and Enter.
3. Use `each` and `puts` to print all the elements in `b`. Use the curly brace `{` and `}` style here.
4. Use `each` to print 3 plus all the elements in `a`. Use the `do ... end` style here.
5. If you multiply a number by itself you are “squaring” the number. Use `each` and `puts` to print the result of multiplying all the numbers in `a` by themselves.

By the way, did you notice that in question 2 you had already typed it exactly? The `reverse` function works both ways! (My son thought this was a good joke when I showed him :) In `irb` you can just type the up arrow to get access to your previous typing. So if you just hit the up arrow a few times you could get back to `b.map! {|item| item.reverse}` and not have to type it again!

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 11: Files

Back in chapter 9 we learned how to get input from the command line. In this chapter, we will learn how to get input from files!

First, we need to create a file to get some input from.

Start Terminal and `cd` into the `stuff` directory by typing `cd stuff` and pressing the Enter key. If you are still in `irb` from the last chapter, you need to quit `irb` first by typing `quit` and pressing the Enter key.

Start `nano` by typing `nano` and pressing the Enter key in Terminal.

We are going to create a file with a list of words, one per line. Type the following in `nano`, making sure to put an extra blank line at the end of the file (after you type `pear`, hit Enter twice):

```
1 apple
2 banana
3 peach
4 pear
```

To save the file, hold down the `control` key and type the `o` key. Nano will ask you for a filename. Type `words.txt` and press the Enter key. Then quit `nano` by holding down the `control` key and type the `x` key.

Let's check that we created the file correctly by using the `cat` program to print the file:

```
1 ~/stuff[]$ cat words.txt
2 apple
3 banana
4 peach
5 pear
6
7 ~/stuff[]$
```

If you didn't get this, or if this did not make any sense, you should quickly re-read chapter 1 to refresh your memory about how to use nano!

Start irb.

Let's play with this file!

We are going to start by opening up the file, reading every line and printing it. This sounds complicated, right? Well, in Ruby, it's easy: it's just a small loop, using a block. Type these 3 lines. Note that when you type `File` the `F` is a capital `F`. Also, there is a period (`.`) between `File` and `foreach`:

```
1 irb> File.foreach("words.txt") do |line|
2   irb> puts line
3   irb> end
4   apple
5   banana
6   peach
7   pear
8
9   => nil
10  irb>
```

That was easy!

But, what if we want to actually keep the contents of the file around to do other stuff with it? After our block is done, the lines are gone!

Let's read the file a different way:

```
1  irb> lines = File.readlines "words.txt"
2  => ["apple\n", "banana\n", "peach\n",
3  "pear\n", "\n"]
```

We used the `readlines` function of `File` to read all the lines of the file at once, into an array! We stored this array of lines in a variable called `lines`.

But we have those `\n` characters again! We don't want those!

Let's get rid of them, using the `map!` function.

```
1  irb> lines.map! {|line| line.strip}
2  => ["apple", "banana", "peach", "pear", ""]
```

At this point, the `lines` variable is an array with each line without the newlines.

But, we have an empty string in the last element of the `lines` array. We don't want that either! Let's get rid of it, using the `reject!` function.

```
1  irb> lines.reject! {|line| line.empty?}
2  => ["apple", "banana", "peach", "pear"]
```

That's better!

The `reject!` function works by throwing away every element which matches the test inside the block. For us, the test was `line.empty?`. So, this is how we used `reject!` to get rid of all the blank lines. (There is both a `reject` and a `reject!` version: `reject` does not modify the original array, while `reject!` does. This is just like the `map` and `map!` or `sort` and `sort!` functions.)

By the way, you may be wondering why I made sure you added a blank line in `nano`, just to go to the work of rejecting it here. The reason is simple: part of programming is dealing with inputs that

might not be exactly how you want them. So, you need to make sure that everything is fine before proceeding, instead of just charging ahead hopefully.

Anyway, let's now print the lines. For variety, we'll use `do ... end` not `{ ... }`. (In practice, the usual style is to use `{ ... }` for one line loops, and `do ... end` for multiple line loops.)

```
1  irb> lines.each do |line|
2  irb> puts line
3  irb> end
4  apple
5  banana
6  peach
7  pear
8  => ["apple", "banana", "peach", "pear"]
```

Before we stop this chapter, let's actually write a program!

We've been spending a lot of chapters just playing in `irb`. While this is fun, it's not permanent. When we quit `irb`, we lose the work we've done!

We are going to write a program called `doge`. What it will do is print all the lines of a file, like `cat` does. But, it will also say the lines. Also, at the start of the program it will print and say "wow", and at the end of the program it will print and say "so amaze". (No, this is not good grammar! But memes on the internet don't have good grammar usually...)

Quit `irb`, then start `nano`.

You're going to write the longest program you've ever written!

Let's start by defining a function called `putsay`.

This function will use `puts` to print whatever string it is given as a parameter. We will call the parameter `str`, for string.

It will then use the `say` command to say the string. We use the backticks to run the `say` command. To type a backtick, press the key to the left of the 1. Make sure you type **both** backticks, the one before `say` and the one after `#{str}`. To type a `#`, hold the `shift` key and type 3. Also, note that you are in `nano`, so if you make a mistake, don't worry, just fix it!

```
1 def putsay(str)
2   puts str
3   `say #{str}`
4 end
```

After the `end` statement, hit `Enter` twice. This adds a newline and a blank line. You need the newline after the `end`. The second newline is just so our program looks nice.)

Then, add the following code. Again, the `File` needs a capital `F`, and there is a period (`.`) between `File` and `readlines`. Also, make sure you add the quotes before and after `words.txt`:

```
1 lines = File.readlines "words.txt"
2 lines.map! {|line| line.strip}
3 lines.reject! {|line| line.empty?}
4 putsay "wow"
5 lines.each do |line|
6   putsay line
7 end
8 putsay "so amaze"
```

Here's what this code does:

1. We read all the lines in the `words.txt` file into an array, which is stored in a variable called `lines`.
2. We then get rid of the newlines using `map`.
3. We then get rid of the blank lines using `reject`.

4. We use `putsay` to print and say “wow”.
5. We then loop over all the lines in the `lines` array using a block. For each line, we call the `putsay` function we defined.
6. Finally, we use `putsay` to print and say “so amaze”.

Before you save the file, make sure it looks like this:

```
1 def putsay(str)
2   puts str
3   `say #{str}`
4 end
5
6 lines = File.readlines "words.txt"
7 lines.map! {|line| line.strip}
8 lines.reject! {|line| line.empty?}
9 putsay "wow"
10 lines.each do |line|
11   putsay line
12 end
13 putsay "so amaze"
```

To save the file, hold down the `control` key and type the `o` key. Nano will ask you for a filename. Type `doge.rb` and press the Enter key.

Then quit nano by holding down the `control` key and type the `x` key.

Run your new program by typing `ruby doge.rb` and hit Enter.

You will see the following:


```
1 ~/stuff[]$ ruby doge.rb
2 wow
3 apple
4 banana
5 peach
6 pear
7 so amaze
```

(Ignore the `~/stuff[]$`, that is just what my prompt looks like.)

Also, if you had your sound on, you should have heard your program!

Great job! This was the most complex program you've ever written...

Exercises

1. Make the doge program reverse the order of the words in the array. (Arrays have a `reverse!` and a `reverse` function.)
2. Arrays also have `shuffle` and `shuffle!` functions. These functions take no arguments and randomize the order of the elements of the array. (The `shuffle!` version modifies the original array, while the `shuffle` version does not.) Use one of those functions to make the doge program print the words in the array in a random order. Test this a few times to convince yourself that the order is indeed changing.

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 12: Booleans, If and While

Programs are built out of functions, and out of series of statements. But there is one other really important thing that programs are built on:

Making decisions.

If we can teach our program to make decisions based on the truth or falseness of expressions and variables, then we can solve a lot of problems! This includes deciding what to do using an `if` statement, and deciding whether to loop using a `while` loop.

Anyway, we have some learning to do. Start `irb`.

In programming languages, truth is precisely defined. (In human languages, it's trickier!)

In Ruby, `true` is true and `false` is false:

```
1  irb> true
2  => true
3  irb> false
4  => false
```

You can also negate `true` and `false`. In this sense, “negate” means to make `true` `false`, and `false` `true`. The way you do this is with the `not` operator.

```
1  irb> not true
2  => false
3  irb> not false
4  => true
```

Now, in most programming languages, it is preferable to write `not` as `!` instead of as `not`. Ruby is no exception here:

```
1  irb> !true
2  => false
3  irb> !false
4  => true
```

You can assign `true` and `false` directly to variables.

```
1  irb> t = true
2  => true
3  irb> t
4  => true
5  irb> f = false
6  => false
7  irb> f
8  => false
```

You can even negate these variables too.

```
1  irb> !t
2  => false
3  irb> !f
4  => true
```

In programming languages like Ruby, `true` and `false` is not only precisely defined, it is also produced as a result of **boolean expressions**. Boolean expressions are just expressions that produce either `true` or `false`.

(Different programming languages have their own quirks here, and Ruby is no different, so I'm glossing over some details.)

To check if something is the same we use the **equality operator**, which is two equals signs `==` in a row.

```
1  irb> 2 == 2
2  => true
3  irb> 2 == 3
4  => false
```

2 is equal to 2, so the first expression returns true. And 2 is not equal to 3, so the second expression returns false.

You can use the equality operator to test more than just numbers. For example, it can test strings. (Don't forget the quotes here! If you do not put quotes around the letters a and b, Ruby will get confused and think you are referring to some variables a and b that it does not know anything about.)

```
1  irb> "a" == "a"
2  => true
3  irb> "a" == "b"
4  => false
```

There's also an **inequality operator**. It is written as `!=` and it checks if two things are **not** equal:

```
1  irb> 2 != 2
2  => false
3  irb> 2 != 3
4  => true
5  irb> "a" != "a"
6  => false
7  irb> "a" != "b"
8  => true
```

Now that we know how to test values using equality and inequality operators, we can use these operators to test the values of variables.

So, let's make a variable and assign a value to it, and then test that value.

```
1  irb> x = 2
2  => 2
3  irb> x
4  => 2
5  irb> x == 2
6  => true
7  irb> x == 3
8  => false
9  irb> x != 2
10 => false
11 irb> x != 3
12 => true
```

This doesn't just work for numbers. Let's try it with strings.

```
1  irb> y = "hi"
2  => "hi"
3  irb> y
4  => "hi"
5  irb> y == "hi"
6  => true
7  irb> y != "hi"
8  => false
```

Finally, you can work with the result of a boolean `==` or `!=` test. Let's test if `x` is equal to 2, and assign the result of that test to the variable `u`. We put `x == 2` in brackets to be clear about what happens first.

```
1  irb> x
2  => 2
3  irb> x == 2
4  => true
5  irb> u = (x == 2)
6  => true
7  irb> u
8  => true
```

You know the saying “two wrongs don’t make a right”? Well, in programming, negating a negation makes it true.

```
1  irb> true
2  => true
3  irb> !true
4  => false
5  irb> !!true
6  => true
```

Oh, by the way, this works for variables too.

```
1  irb> u
2  => true
3  irb> !u
4  => false
5  irb> !!u
6  => true
```

Besides the equality `==` and inequality `!=` operators, there are other operators that produce `true` and `false` values too.

Some of the most common ones are ones that compare two things.

- The **less than** operator `<` checks if one thing is less than another thing.
- The **less than or equal to** operator `<=` checks if one thing is smaller than or the same as another thing.
- The **greater than** operator `>` checks if one thing is bigger than another thing.
- The **greater than or equal to** operator `>=` checks if one thing is bigger than or the same as another thing.

Note that I said “thing” here, not “number”. This is because you can compare some things that aren’t numbers using these operators. However, usually you just compare numbers!

By the way, if you need help remembering which is less than and which is greater than, less than `<` looks like an L that is leaning forward.

```
1  irb> 1 < 2
2  => true
3  irb> 1 <= 2
4  => true
5  irb> 2 < 2
6  => false
7  irb> 2 <= 2
8  => true
9  irb> 3 > 2
10 => true
11 irb> 3 >= 2
12 => true
13 irb> 2 > 2
14 => false
15 irb> 2 >= 2
16 => true
```

Now, this is a bit silly since you know what these expressions all are. You know that 1 is less than 2; you don't need Ruby to tell you!

The real use here is in comparing a variable to a number.

```
1  irb> n = 5
2  => 5
3  irb> n
4  => 5
5  irb> n < 6
6  => true
7  irb> n < 5
8  => false
```

Now, this is also contrived, since we know what `n` is. But when you realize that you can set variables to values that came from anywhere, you can begin to see the power here.

For example, we can set `n` to a number that we get from the terminal.

Remember that you can get strings from the terminal using `gets`? In `irb`, type `s = gets`, followed by the Enter key. Then type `hi` and the Enter key.

```
1  irb> s = gets
2  hi
3  => "hi\n"
4  irb> s
5  => "hi\n"
```

Also, remember that we can use `strip` to remove the newline at the end of a string?

```
1  irb> s.strip
2  => "hi"
```

We will do an example where we use `gets` to get a string from the terminal, `strip` to get rid of the newline, and `to_i` to turn it into a number.

In `irb`, type `n = gets.strip.to_i`, followed by the Enter key. Then type `8` and the Enter key.

```
1  irb> n = gets.strip.to_i
2  8
3  => 8
4  irb> n
5  => 8
```

We can then test if `n` is less than 5.

```
1  irb> n < 5
2  => false
```

Let's get a different number, store it in the `n` variable, and then do that test again.

```
1  irb> n = gets.strip.to_i
2  4
3  => 4
4  irb> n < 5
5  => true
```

Anyway, back at the beginning of the chapter, we said that programs are built on making decisions, based on the truth or falseness of expressions or variables.

This is done in 2 ways:

1. deciding what to do using an `if` statement
2. deciding how long to do something using a `while` loop

Let's look at these both now, starting with the `if` statement.

An `if` statement tests an expression or variable. If it is true, the statements inside the `if` part are executed. If it is false, the statements inside the `else` part are executed.

Here's how it works. By the way, I indent the `puts "yes"` and `puts "no"` statements with spaces, but you do not have to:

```
1  irb> if true
2  irb>   puts "yes"
3  irb> else
4  irb>   puts "no"
5  irb> end
6  yes
7  => nil
```

So, we only saw “yes” and not “no”, since `true` is true. The poor `puts "no"` statement will never be run. The return value of this whole `if` statement is `nil`, which is why you see `=> nil` at the end.

Let's get the `else` part to execute:

```
1  irb> if false
2  irb>   puts "yes"
3  irb> else
4  irb>   puts "no"
5  irb> end
6  no
7  => nil
```

In this case, we only saw “no” and not “yes”. The poor `puts "yes"` statement will never be run.

You can also test a variable:

```
1  irb> t = true
2  => true
3  irb> t
4  => true
5  irb> if t
6  irb>   puts "yes"
7  irb> else
8  irb>   puts "no"
9  irb> end
10 yes
11 => nil
```

So, Ruby printed “yes”, since `t` is true.

Let’s get the `else` part to execute:

```
1  irb> t
2  => true
3  irb> if !t
4  irb>   puts "yes"
5  irb> else
6  irb>   puts "no"
7  irb> end
8  no
9  => nil
```

In this case, `t` is true, so `!t` is false.

You can use anything that produces a boolean (true or false) value in the test of an `if` statement.

For example, we can put a less than comparison in the `if` statement test, since it produces a boolean value:

```
1  irb> x = 2
2  => 2
3  irb> if x < 3
4  irb>   puts "smaller"
5  irb> else
6  irb>   puts "bigger"
7  irb> end
8  smaller
9  => nil
```

You can even put a function call in the `if` statement test, since its return value can be tested for truth.

Let's define a function which returns whether a number is bigger than two:

```
1  irb> def bigger_than_two(n)
2    irb>   n > 2
3  irb> end
4  => nil
```

We can then call this function in the `if` test.

Here, we call it with 1, which is not bigger than 2. So, the function returns false, and the `else` case gets executed:

```
1  irb> if bigger_than_two(1)
2    irb>   puts "bigger"
3  irb> else
4    irb>   puts "smaller"
5  irb> end
6  smaller
7  => nil
```

Here, we call it with 3, which is bigger than 2. So, the function returns true, and the `if` case gets executed:

```
1  irb> if bigger_than_two(3)
2    irb>   puts "bigger"
3  irb> else
4    irb>   puts "smaller"
5  irb> end
6  bigger
7  => nil
```

Besides just making decisions based on boolean expressions, we can also loop based on them.

We will do this using a new programming concept: a `while` loop. A `while` loop is basically a loop that does an `if` test in every cycle through the loop, and stops the loop when the test is false.

Type this in irb. When you are tired of seeing “still true” being printed, hold down the `control` key and type `c` to interrupt the loop from running.

```
1  irb> while true
2  irb>   puts "still true"
3  irb> end
4  still true
5  still true
6  still true
7  still true
8  still true
9  ...
```

In this loop, `true` is always `true`, so the `while` loop will run forever, unless you stop it!

When you write a computer program, you are telling the computer what to do. The computer will do **exactly** what you tell it to do, even if you tell it to do something as stupid as print “still true” while `true` is `true`.

By the way, what we just did is create an **infinite loop**, since it will go forever (unless interrupted). Usually, creating an infinite loop is a mistake (or “bug”) in your program, but sometimes it is helpful to deliberately create infinite loops.

Also, as a bit of side trivia: you are using an Apple Mac to follow along with this book, and Apple’s address is 1 Infinite Loop, Cupertino, California.

Let’s make a `while` loop that does something a bit useful: count from 0 to 10.

```
1  irb> i = 0
2  => 0
3  irb> while i <= 10
4  irb> puts i
5  irb> i = i + 1
6  irb> end
7  0
8  1
9  2
10 3
11 4
12 5
13 6
14 7
15 8
16 9
17 10
18 => nil
```

The way this works is each time through the `while` loop, we are adding 1 to `i`. If we did not put the `i = i + 1` statement in, the `while` loop would print 0 forever.

Throughout this book, I've been saying that programmers like me are lazy. You know how lazy we are? We don't even like typing things like `i = i + 1`. We have a shortcut way of doing that in Ruby: the `+=` operator. Let's count from 1 to 5 using `+=` to add 1 to `i`:

```
1  irb> i = 1
2  => 1
3  irb> while i < 6
4  irb> puts i
5  irb> i += 1
6  irb> end
7  1
8  2
9  3
10 4
11 5
12 => nil
```

In many other programming languages, we can be ever lazier and say `i++` instead of `i += 1`. But we can't do that in Ruby!

Exercises

1. In irb, test if 1 is equal to 1.
2. In irb, test if 1 is equal to "1". Can you guess why it is not equal?
3. In irb, write a function called `same` which takes 2 parameters and prints "equal" if they are equal and prints "not equal" if they are not equal. Test it with `same 1,1` and `same 1,2`. Hint: the first line of the function is `def same(a,b)`. You will use an `if / else` statement in the function.
4. In irb, write a `while` loop that multiplies `1*1`, `2*2`, `3*3`, all the way up to `12*12`. Every answer should be printed on its own line, with `puts`.

You can see the [answers to the exercises](#), or just continue to the next chapter.

Chapter 13: The Spelt Project

It's amazing what you've learned so far!

In this chapter, we're going to put everything together and build our first meaningful program! This program will take advantage of almost everything we have learned in this book.

This is the last chapter in the book. It's also the longest chapter. So, you should plan to take a bit more time to work through it. I have split it into 3 sections. Feel free to take a break after each section.

Section 1: Spelling a Word

The program we are going to build is called “spelt”, and it will be a program that you can use to teach yourself how to spell words. If you wonder where the name came from, spelt is a type of bread, and it's also the past tense of spell in the British version of English. In US English, you say spelled, not spelt – so for you, the name of the spelt program would be spelt incorrectly. I like to make (bad) jokes for names of programs.

Anyway, enough of that. We are going to start slowly, experimenting in irb. Start irb.

Let's use say to say the letter h. Make sure you type the backticks at the beginning and the end of the command. (The backtick is beside the 1 key.)

```
1  irb> `say "h"`  
2  => ""
```

Great, we heard “h”. So, the `say` command can say individual letters. Now, let’s try `say` with just a space.

```
1  irb> `say " "`  
2  => ""
```

We didn’t hear anything!

In our spelling program `spelt`, we are going to be spelling things using `say`. We will make a `say` function to help with this.

If our `say` function is given a letter, it will say the letter. Otherwise, the `say` function should say “space”.

So, we’re going to use a function containing an `if` statement to accomplish this.

In terminal, quit `irb` and then open `nano` by typing `nano`.

We are going to create the `say` function.

```
1  def say(str)  
2    if str == " "  
3      `say "space"`  
4    else  
5      `say #{str}`  
6    end  
7    str  
8  end
```

Note that the `if str == " "` line has a space in between the opening and closing quotes!

Here’s how it works: If `say` is given a string which is `" "` (the space character), it will say “space”. Otherwise, it just calls the `say`

command with whatever string was passed in, using the `#{}` syntax that we learned earlier.

Finally, the `say` function returns the string that it is given. (We might as well return something, so that we can chain function calls together.)

By the way, to indent the code inside a function, use two spaces per level of indent. You don't have to indent the code if you don't want to. (My son always refuses to, since in Ruby, the spaces don't matter. But, in some languages, like Python and CoffeeScript, they do matter.) Indenting is good practice, though, since it lets you read your own programs more easily.

Let's try this. Save the file by typing `control + o`, typing the file name of `spelt.rb` and hitting the `Enter` key.

Nano will say "[Wrote 7 lines]". (Or, if you have an extra newline at the end of the file, it will say "[Wrote 8 lines]". Either is fine.

Quit nano by typing `control + x`.

Next, start `irb` again.

Load the `spelt` program by typing `load "spelt.rb"` and hitting `Enter`. Then, let's test out our `say` function, first by trying `say "hi"`, then `say "h"`, `say " "` and finally `say "hello world"`.

```
1  irb> load "spelt.rb"
2  => true
3  irb> say "hi"
4  => "hi"
5  irb> say "h"
6  => "h"
7  irb> say " "
8  => " "
9  irb> say "hello world"
10 => "hello world"
```

It works! When our `say` function is given only a space, it says “space”. Otherwise, it says what it was given, whether it was a letter, or “hi” or “hello world”.

Quit `irb`. Next, open `spelt.rb` in `nano` by typing `nano spelt.rb` and hitting `Enter`.

Next, we will add a `putsay` function to `spelt`, which will be similar to the one we created for our `doge` program. This function will take a string as its parameter, and it will print the string using `puts` and `say` the string. However, instead of using the `say` command in backticks, it will call our `say` function.

Scroll down to the bottom of the file (below the definition of the `say` function) using the down arrow. Then, add a blank line and then this function:

```
1 def putsay(str)
2   puts str
3   say str
4 end
```

The `putsay` function returns the string that it is given. (We might as well return something, so that we can chain function calls.)

At this point, the `spelt.rb` program should look like this:

```
1 def say(str)
2   if str == " "
3     `say "space"`
4   else
5     `say #{str}`
6   end
7   str
8 end
9
10 def putsay(str)
```

```
11 puts str
12 say str
13 end
```

Note that the last statement in the `putsay` function is a function call: `say str`. So, the `putsay` function returns the return value of this function call. So, since `say` returns the string it is given, that is what `putsay` function will also return.

Save the file by typing `control + o` and then hitting the `Enter` key. This will overwrite the `spelt.rb` file with our new version.

Then, quit `nano` by typing `control + x`.

Finally, start `irb` again and load the `spelt` program.

```
1 irb> load "spelt.rb"
2 => true
```

Let's test our new `putsay` function:

```
1 irb> putsay "hi"
2 hi
3 => "hi"
4 irb> putsay " "
5
6 => " "
```

Now, the next thing we are going to do is write a function that spells whatever the string is, letter by letter.

But, how are we going to do this?

It turns out it is really easy!

In Ruby, strings come with a bunch of functions. One of them is called `split`, which returns an array of strings.

What `split` does is best shown by example:

```
1  irb> "this is cool".split " "  
2  => ["this", "is", "cool"]
```

The `split` function takes a string parameter which is used as a **delimiter**. A delimiter means a thing that splits something into pieces, like slices in a loaf of bread. It is the thing that you separate the array by.

So, calling `split` on “this is cool” with a delimiter of “ ” (space) turns into the array `["this", "is", "cool"]`.

The `split` function always returns an array of strings. Each string is the part of the string up to the first part of the string that matches the delimiter. The delimiters themselves are not included in the returned array. (There are no spaces in the array `["this", "is", "cool"]`.)

Let’s try calling `split` on “123.456.789” with a delimiter of “.” (period).

```
1  irb> "123.456.789".split "."  
2  => ["123", "456", "789"]
```

Here, this splits the string “123.456.789” into an array containing 3 strings:

```
1  1. "123"  
2  2. "456"  
3  3. "789"
```

It throws out the delimiter, which is a period.

Let’s try with commas:

```
1 irb> "ruby,is,fun".split ","
2 => ["ruby", "is", "fun"]
```

Here, this splits the string "ruby,is,fun" into an array containing 3 strings:

```
1 1. "ruby"
2 2. "is"
3 3. "fun"
```

It throws out the delimiter, which is a comma.

Finally, let's try this with a delimiter of an empty string (""):

```
1 irb> "hello".split ""
2 => ["h", "e", "l", "l", "o"]
```

Here, there is an empty string between every letter (or “character”) in the string, so this splits the string into an array of the 5 characters that make up the string.

This will let us spell words! All we need to do is use `split` to print and say each letter.

Let's use a block to do this! A block, which we saw earlier in the book, is a way of looping through something:

```
1  irb> letters = "hello".split ""
2  => ["h", "e", "l", "l", "o"]
3  irb> letters.each do |letter|
4  irb> putsay letter
5  irb> end
6  h
7  e
8  l
9  l
10 o
11 => ["h", "e", "l", "l", "o"]
```

As a piece of advice, you should try to keep your code as simple as possible. That way, when (not if!) things go wrong with it, you will have an easier time finding the mistakes, or “bugs” in your code. If you make your code complicated just to show off, you will end up wasting your own time later!

Quit irb.

Next, open `spelt.rb` in nano by typing `nano spelt.rb` and hitting Enter.

We’re going to make a `spell` function. Now, in English, we don’t normally write one letter per line. So, we’re going to make a slight changes to what we did in irb: we are going to say every letter without printing it. So, we will use the `say` function we wrote, not the `putsay` function. If we want to print the word, we can always just print it using `puts`.


```
1 def spell(str)
2   str.split("").each do |char|
3     say char
4   end
5   str
6 end
```

The first `end` is to end the `do ... end` loop over the characters (`char`) of the string `str`. (For StarCraft players: Here, `char` is for “character”, not for the Zerg homeworld!) The second `end` is to end the function. Indenting our code helps us keep track of the `def ... end` and `do ... end` pairs.

At this point, the `spelt.rb` program should look like this:

```
1 def say(str)
2   if str == " "
3     `say "space"`
4   else
5     `say #{str}`
6   end
7   str
8 end
9
10 def putsay(str)
11   puts str
12   say str
13 end
14
15 def spell(str)
16   str.split("").each do |char|
17     say char
18   end
19   str
20 end
```

Save the file by typing `control + o` and then hitting the `Enter` key. This will overwrite the `spelt.rb` file with our new version.

Then, quit nano by typing `control + x`.

Finally, start `irb` again and load the `spelt` program.

```
1  irb> load "spelt.rb"
2  => true
```

We are going to test the `spell` function:

```
1  irb> spell "hello world"
2  => "hello world"
```

You should have heard:

```
h e l l o space w o r l d
```

Awesome! This might be a good point to take a small break. You're about a third of the way through this last chapter!

Section 2: Finishing Spelt

What to do next?

Well, if we're writing a spelling program, we need to have a way of defining a list of words that we should learn how to spell, right?

Luckily, we already have one: two chapters ago, we wrote a program called `doge` where we read a list of words out of a file that had 1 word per line!

This approach would be absolutely perfect for our `spelt` program. What a coincidence! (wow, so amaze, wow :)

Quit `irb`.

Next, open `spelt.rb` in nano by typing

nano spelt.rb

and hitting Enter.

Add this code to the bottom of the spelt program. By the way, since this is a lot of typing, feel free to select the text, copy it to the clipboard with command + c, and paste it into nano with command + v. You will need to reformat it though, to look like this...

```
1 words = File.readlines "words.txt"
2 words.map! {|line| line.strip}
3 words.reject! {|line| line.empty?}
4 num_words = words.length
5 putsay "Welcome to Spelt!"
6 putsay "We will spell #{num_words} words."
7 num_right = 0
8 words.each do |word|
9   say "spell #{word}"
10  answer = gets.strip
11  if word == answer
12    num_right = num_right + 1
13    say "awesome"
14  else
15    say "wrong"
16  end
17 end
18 putsay "You got #{num_right}" +
19 " out of #{num_words}!"
```

Here's what this code does:

1. We read all the lines in the words.txt file into an array, which is stored in a variable called words. We are assuming there is a maximum of 1 word per line.
2. We then get rid of the newlines using map!. What map! does is modifies the array it is called on, and replaces each element

of the array with the result of doing what is inside the block. For us, we are calling `strip` on the line. So, this code removes all the newlines.

3. We then get rid of the blank lines using `reject`.
4. We get the number of words with `words.length` and store the number in `num_words`. Arrays also have a `length` function which gets how long the array is.
5. We use `putsay` to print and say “Welcome to Spelt”.
6. We use `putsay` to say how many words we are going to spell.
7. We create a variable called `num_right` which will count the number of right answers.
8. We then loop over all the words using a block. For each word, we do this:
9. We say “spell” and the word.
10. We get a line of input from the keyboard using `gets.strip` (which you saw earlier in the book) and store it in a variable called `answer`.
11. We check if the word is the same as the answer using another if statement, `if word == answer`. If the if statement is true, then the user spelled the word right, so we add 1 to our count of right answers and then say “awesome”. If the if statement is not true, then we say “wrong”.
12. At the end, we say how many words were spelled right out of the total number of words. Normally you would say `putsay "You got #{num_right} out of #{num_words}!"` instead of doing this on 2 lines with `putsay "You got #{num_right}" + and " out of #{num_words}!"`. However, I did it this way so that the code did not break in the middle of the line in the book.

By the way, this type of sequence of steps, loops and decisions has a really fancy name: an **algorithm**.

An algorithm (“al go rith um”) is the approach that we can use to solve a problem. When we write algorithms, we typically combine

a sequence of steps with if/else statements and loops that make decisions. So, when we are writing computer programs and solving problems, we first come up with the algorithm and then we write the code.

Don't let the fancy name fool you: you use algorithms all the time. Lots of the math you learn at school, stuff like 2 digit multiplication and long division, is just you learning an algorithm. The steps you learn are the algorithm, and you are being taught how to "play computer" to use the algorithm to solve the problem.

Anyway, here's what the full `spelt` program looks like. Make sure that your program looks like this:

```
1  def say(str)
2    if str == " "
3      `say "space"`
4    else
5      `say #{str}`
6    end
7    str
8  end
9
10 def putsay(str)
11   puts str
12   say str
13 end
14
15 def spell(str)
16   str.split("").each do |char|
17     say char
18   end
19   str
20 end
21
22 words = File.readlines "words.txt"
23 words.map! {|line| line.strip}
```

```
24 words.reject! {|line| line.empty?}
25 num_words = words.length
26 putsay "Welcome to Spelt!"
27 putsay "We will spell #{num_words} words."
28 num_right = 0
29 words.each do |word|
30   say "spell #{word}"
31   answer = gets.strip
32   if word == answer
33     num_right = num_right + 1
34     say "awesome"
35   else
36     say "wrong"
37   end
38 end
39 putsay "You got #{num_right}" +
40 " out of #{num_words}!"
```

Save the file by typing `control + o` and then hitting the Enter key. This will overwrite the `spelt.rb` file with our new version.

Then, quit nano by typing `control + x`.

Run your new program by typing `ruby spelt.rb` and hit Enter.

You will see the following:

```
1 ~/stuff[]$ ruby spelt.rb
2 Welcome to Spelt!
3 We are going to spell 4 words.
```

(Ignore the `~/stuff[]$`, that is just what my prompt looks like.)

You will hear “spell apple”.

Let’s deliberately make a mistake. Type `a` + Enter instead of `apple`. Then correctly spell `banana`, `peach` and `pear`.

You will see this:

```
1 ~/stuff[]$ ruby spelt.rb
2 Welcome to Spelt!
3 a
4 banana
5 peach
6 pear
7 You got 3 out of 4!
```

Also, if you had your sound on, you should have heard your program!

This might be a good point to take another small break. You're about two thirds of the way through this last chapter!

Section 3: Enhancing Spelt

One of the great things about programming is that you can constantly challenge yourself to improve! You can constantly improve your programs, and your skill as a programmer.

In this final section, we will do both, by enhancing our `spelt` program in two ways.

First, just saying “wrong” when the user spells something wrong is not very helpful. Let's make the `spelt` program teach the spelling of a word when it is spelled wrong!

We are going to delete the

```
1      say "wrong"
```

line and add a while loop instead. The rest of the program is unchanged, so I'm skipping writing it down here and showing . . . instead:

```
1  ...
2  words.each do |word|
3    say "spell #{word}"
4    answer = gets.strip
5    if word == answer
6      num_right = num_right + 1
7      say "awesome"
8    else
9      say "wrong"
10     while word != answer
11       say "The correct spelling is"
12       puts word
13       spell word
14       say "not"
15       spell answer
16       say "try again"
17       say "spell #{word}"
18       answer = gets.strip
19     end
20     say "great job"
21   end
22 end
23 putsay "You got #{num_right}" +
24 " out of #{num_words}!"
```

Here's what this code does:

It loops while the word is not equal (!=) to the answer. In each pass through the loop, the code...

1. says "The correct spelling is"
2. prints the word with puts
3. calls the `spell` function to spell the word
4. says "not"
5. says the spelling that the user provided as the answer
6. says "try again"

7. says “spell” and the word being spelt
8. gets a new answer and removes the newline using `gets.strip`

Then, the loop goes back to the beginning. It tests again whether the word is not equal to the answer. If it is not equal, we loop again. (So, if the user is unable to learn spelling, we have an infinite loop!)

If the spelling is equal, the loop finishes and we say “great job”.

Now, asking the user to spell a word when you are showing them the word is not that difficult. Let’s fix that. There is a command called `clear` that clears the Terminal window in Mac.

Add a function called `cls` (for “clear screen”) to the top of your `spelt` program.

```
1 def cls
2   print `clear`
3 end
```

We are then going to call this new `cls` function from 4 places in the program. (There are 3 uses of `cls` on the first page, and one on the second page.) The rest of the code is the same.

```
1 ...
2 words = File.readlines "words.txt"
3 words.map! {|line| line.strip}
4 words.reject! {|line| line.empty?}
5 num_words = words.length
6 cls
7 putsay "Welcome to Spelt!"
8 putsay "We will spell #{num_words} words."
9 num_right = 0
10 words.each do |word|
11   say "spell #{word}"
12   answer = gets.strip
```

```
13   if word == answer
14       num_right = num_right + 1
15       say "awesome"
16       cls
17   else
18       while word != answer
19           say "The correct spelling is"
20           puts word
21           spell word
22           say "not"
23           spell answer
24           cls
25           say "try again"
26           say "spell #{word}"
27           answer = gets.strip
28       end
29       say "great job"
30   end
31 end
32 cls
33 putsay "You got #{num_right}" +
34 " out of #{num_words}!"
```

Save the file by typing `control + o` and then hitting the `Enter` key. This will overwrite the `spelt.rb` file with our new version.

Then, quit nano by typing `control + x`.

As a treat, I'm going to show you what this code looks like, with syntax highlighting turned on. (I haven't been showing the examples in the book with syntax highlighting, since I didn't want you to rely on it.)

```
<<(code/spelt.rb)
```

By the way, if you're wondering how to see syntax highlighting when you write code, you can't do that in nano. Other text editors,

like [Sublime Text](http://www.sublimetext.com/)¹ and [Emacs](http://emacsformacosx.com/)² let you do this.

Run your new program by typing `ruby spelt.rb` and hit Enter.

You will see the screen get cleared at the right times. Also, if you spell a word wrong, `spelt` will show it to you, spell it to you, and then test you again.

Great job! This was the most complex program you've ever written!

You should be very, very proud of yourself for what you have learned in this book.

Exercises

These final exercises are a bit different. You're going to read code, and then make small changes. Lots of being a programmer involves being able to read, understand and modify code, not just write it from scratch.

Below is a ruby program called `table` that lets you practice times tables.

Just like `spelt`, it uses a `say` and a `putsay` function, so the program talks to you as well as printing output.

```
1 def say(str)
2   `say #{str}`
3   str
4 end
5
6 def putsay(str)
7   puts str
8   say str
9 end
```

¹<http://www.sublimetext.com/>

²<http://emacsformacosx.com/>

```
10
11 putsay "Welcome to table!"
12 putsay "We are going to learn times tables."
13 putsay "What number do you want to stop at?"
14 max = gets.to_i
15 num_right = 0
16 num_questions = 0
17 i = 1
18 while i <= max
19   j = 1
20   while j <= max
21     putsay "What is #{i} times #{j}?"
22     answer = gets.to_i
23     num_questions += 1
24     right = i*j
25     if answer == right
26       putsay "Awesome!"
27       num_right += 1
28     else
29       putsay "Wrong!"
30     end
31     j += 1
32   end
33   i += 1
34 end
35 putsay "You got #{num_right}" +
36 " out of #{num_questions}."
```

Notes:

1. The say function does not need to handle spaces in a special way, unlike in the `spelt` program.
2. We are going to get the maximum from the user, and loop between 1 and max.
3. We use nested while loops in this program, since it's the easiest thing to do!

Question 1

Make a new file in the `stuff` directory called `table.rb`. Copy the text of this program and paste it in. To copy the text, you select it in Preview, then type `command + c` to copy. Then, start nano by typing `nano` in Terminal. When in nano, paste the text by typing `command + v`. Since the program goes across 2 pages, you will have to do this for each page. Also, when you paste, lots of the newlines and indenting might be removed. You'll need to fix that by using the arrow keys to go to the right place, and the space bar or Enter keys to format the code properly.

Run the program with `ruby table.rb` and play along. (Enter 2 for the number you want to stop at. After the first run with stopping at 2, you can also use a larger number if you want to practice times tables.)

[See the answer.](#)

Question 2

Change the code to print and say "Yay!" when the user gets the question right (instead of "Awesome!"). Open the `table.rb` file by typing `nano table.rb`. Save, quit and test, entering 3 as the number you want to stop at.

[See the answer.](#)

Question 3

Change the code to only multiply by even numbers (2,4,6,8,...) in the times table. Start at 2, and go up to the largest even number less than or equal to max. Save, quit and test, entering 5 as the number you want to stop at.

This is a hard question! Think about it for a while. I'm going to give you 3 hints, one per page, starting on the next page. See how many hints you need before you [look at the answer](#).

Hint #1 for Question 3

You just have to change 4 lines of code. You do not need to add or delete any lines of code.

Hint #2 for Question 3

You need to count by 2's, not by 1's.

Hint #3 for Question 3

If you don't change the starting number, you'll be counting odd numbers, not even ones!

[See the answer.](#)

For Parents

This is a book for you to read with your child, or for your child to read by himself or herself.

I wrote it to teach my 9 year old son the basics of programming. He has been playing video games for years, and he wants to learn programming since he wants to make his own video games someday. This book is intended to be the first step. (No, it doesn't teach you how to create the kinds of video games a 9 year old can dream up; that takes a lot more knowledge!)

The reason this book exists is to be the best book in the world for a kid who is wanting to learn to program computers to read first. Computer programming is a good skill to have, regardless of what occupation your child eventually does as an adult. (I'd argue it's much more important than lots of the math than you learn in high school, for example.) But more importantly, learning how to program computers teaches a rigor and discipline of thinking which is useful in any field. This book exists to show kids that they can program computers, and to help them get started.

With the exception of this appendix, this book is written like a book for kids in elementary school. My goal is that this book should be accessible for kids between ages 9 and 14. In North America, that's grades 3-8. My son is working through it as I write it. (I don't think it's a good book for kids aged 7 and 8: I started writing this book when my son was 7, but he wasn't ready for it. So, I paused writing it—for 2 years. If your child is 7 or 8, I think that something like Scratch is a better choice for kids of that age.)

No knowledge of programming is assumed. The examples are as short as possible, since I assume the reader can't type well. (My son can't touch type, so if I make long examples I'll hear about it!)

My goal is for this book to be the best programming book for kids to read first. After this book, they can follow what interests them.

This book is written assuming you are using a Mac. I think that a Mac is the best computer for kids to learn to program on. Since this is a beginner book for kids, I can't write it generically to cover Mac, Windows and Linux. I have to pick one operating system, and have the child follow along verbatim.

Besides teaching programming, the book also teaches basic use of the command line on a Mac. This is accessed via the Terminal program. The reason for this is that I feel that the best way to learn is to follow along, and the simplest way to follow along is to type everything. Real programmers use the command line every day. If you want to learn programming, you should use Terminal and files. Yes, you can play with stuff in a web browser at places like Codecademy, and while this is very friendly and instructive, it is fundamentally a different activity from what real programmers do. And, besides being easier, it's somehow less rewarding.

If you are letting your child use **your** Mac computer to follow along, I **strongly** recommend you sit beside them and follow along! For example, I'm not planning to teach the command to delete files, but it's fairly short!

The examples are in Ruby. Ruby is a fairly simple programming language. If you've ever heard of websites built on "Ruby on Rails", you've heard of Ruby: it's the programming language that Rails is written in. This is not a book about how to learn Ruby, however. The examples could have easily been written in JavaScript, CoffeeScript or Python.

Finally, I really want your feedback! Did your child get stuck anywhere? If you have anything to say about the book, I want to hear it! Please email me at peter@leanpub.com and let me know!

About the Author

I'm the founder of Ruboss, a software consulting company based in Vancouver, BC, Canada. We're the creators of Leanpub, a website that anyone can use to self-publish in-progress ebooks like this one. I'm also a programmer, an author and a father. I've written two books for programmers (*Flexible Rails* and *Hello! Flex 4*), so I know how to explain things to fellow programmers. And my son is a very intelligent 9 year old, so I have a lot of experience explaining things to a smart child. This is probably the most challenging book I will ever write, as I honestly want this to be the best introduction to programming for all kids.

About the Cover

The cover [photo](#)³ is by [Gareth Newstead](#)⁴ and is from [Unsplash](#)⁵.

³https://unsplash.com/photos/YoC_rDkSS1U

⁴http://www.garethnewsteadphotography.com/?utm_medium=referral&utm_source=unsplash

⁵<https://unsplash.com/>

About Leanpub

This is a Leanpub book. I'm the cofounder of Leanpub. Leanpub is a website which lets anyone publish their own books as they write them. The idea of publishing an in-progress ebook is something I call Lean Publishing. You can learn more about Lean Publishing by reading a [free book](#)⁶ that I wrote last year.

Besides being totally free to use, being a Leanpub author can also be profitable. We pay authors 80% royalties per copy sold. So, a \$10 ebook pays \$8 in royalties per copy!

⁶<https://leanpub.com/lean>

Answers to Exercises

Chapter 1

Question 1

You start Terminal and `cd` into the `stuff` directory by doing this:

1. In Finder, type **Command + Space** to start Spotlight
2. Type **terminal**
3. Hit the Enter key
4. Type **`cd stuff`** in Terminal and hit Enter.

You will now be in Terminal in the `stuff` directory.

Question 2

To start nano, just type `nano` in the Terminal window, and hit Enter.

You will now be in nano.

Type `hooray`.

Hold down the **Control** key and type `o`. Type `hooray.txt` and press Enter. Nano will say “Wrote 1 line” (or “Wrote 2 lines”, if you hit Enter after typing `hooray`).

Hold down the **Control** key and type `x` to quit nano.

Question 3

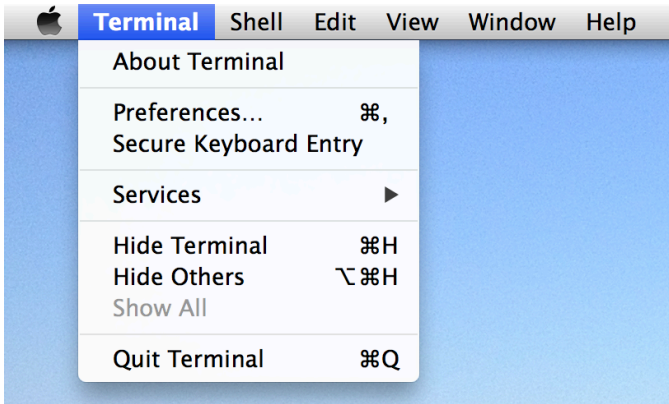
Type **`cat hooray.txt`** to see the contents of `hooray.txt`.

You will see something like this:

- 1 `~/stuff[]$ cat hooray.txt`
- 2 `hooray`

Question 4

To quit Terminal, type **Command + Q** or choosing Quit Terminal from the Terminal menu.



Continue Reading

If this was confusing, please [read chapter 1 again](#).

If this made sense, please [continue to chapter 2](#).

Chapter 2

Question 1

To start nano, type `nano` in a Terminal window, and hit Enter. You will now be in nano. Type `puts "Hi!"`. To type `!`, hold the shift key and type `1`.

To save the file, hold down the Control key and type `o`. Type `hi.rb` and press Enter. Nano will say “Wrote 1 line” (or “Wrote 2 lines”, if you hit Enter after the `puts "Hi!"`).

Hold down the Control key and type `x` to quit nano.

In Terminal, type `ruby hi.rb` and hit Enter. You need a space between `ruby` and `hi.rb`.

You will see something like this:

```
1 ~/stuff[]$ ruby hi.rb
2 Hi!
```

Continue Reading

If this was confusing, please [read chapter 2 again](#).

If this made sense, please [continue to chapter 3](#).

Chapter 3

Question 1

```
1  irb> 3-2
2  => 1
```

Question 2

```
1  irb> 3.0/2
2  => 1.5
```

1.5 is 1 and a half. 3 divided by 2 is 1 and a half. 1 and a half can be written as $1 \frac{1}{2}$, but can also be written as 1.5.

By the way, there's a reason why I said 3.0/2 and not 3/2. But, I don't want to explain that yet. I will later!

Continue Reading

If this was confusing, please [read chapter 3 again](#).

If this made sense, please [continue to chapter 4](#).

Chapter 4

Question 1

```
1  irb> def quad(x)
2  irb> x*4
3  irb> end
4  => nil
```

To test the function, you can either use brackets or not.

```
1  irb> quad 3
2  => 12
3  irb> quad(3)
4  => 12
```

Question 2

```
1  irb> def quad2(x)
2  irb> x + x + x + x
3  irb> end
4  => nil
```

To test the function, you can either use brackets or not.

```
1  irb> quad2 3
2  => 12
3  irb> quad2(3)
4  => 12
```

Note that if you do not use brackets, you need to have a space between the function name and its argument. You cannot say `quad23`, since Ruby has no way of figuring out what you mean!

```
1  irb> quad23
2  NameError: undefined local variable or
3  method `quad23' for main:Object
4      from (irb):39
5      from :0
6  irb>
```

Question 3

You need to write this function using brackets around the function call:

```
1  irb> def quad3(x)
2  irb> double(x) + double(x)
3  irb> end
4  => nil
```

Note that if you don't do this, Ruby won't be able to figure out what you are doing. Usually, Ruby is smart enough to do this, but in this case it is not:

```
1  irb> def quad3(x)
2  irb> double x + double x
3  irb> end
4  SyntaxError: compile error
5  (irb):28: syntax error, unexpected
6  tIDENTIFIER, expecting kDO or
7  '{' or '('
8      from (irb):29
9      from :0
10 irb>
```

Also, note that you could have written this function another way, by just doubling what you got from doubling x once:

```
1  irb> def quad3(x)
2  irb> double double x
3  irb> end
4  => nil
5  irb> quad3 3
6  => 12
```

If that didn't make total sense, don't worry: we'll look at this more in the Functions 2 chapter later on.

Also, if you want, you can put your function calls in brackets. As you saw above, sometimes you need to do this to let Ruby figure out what you mean. (This is not one of those cases, though.)

```
1  irb> def quad3(x)
2  irb> double(double(x))
3  irb> end
4  => nil
5  irb> quad3 3
6  => 12
```

Continue Reading

If this was confusing, please [read chapter 4 again](#).

If this made sense, please [continue to chapter 5](#).

Chapter 5

Question 1

Start nano and add the code below that is in **bold** text.

```
1  def double(x)
2    x*2
3  end
4
5  def triple(y)
6    y*3
7  end
8
9  def quadruple(z)
10 z*4
11 end
12
13 puts double 2
14 puts triple 3
15 puts quadruple 4
```

Save the file and run `ruby times.rb`.

```
1 ~/stuff[]$ ruby times.rb
2 4
3 9
4 16
5 ~/stuff[]$
```

Continue Reading

If this was confusing, please [read chapter 5 again](#).

If this made sense, please [continue to chapter 6](#).

Chapter 6

Question 1

```
1  irb> def times6(x)
2  irb> double triple x
3  irb> end
4  => nil
5  irb> times6 6
6  => 36
```

Question 2

```
1  irb> def times12(x)
2  irb> double double triple x
3  irb> end
4  => nil
5  irb> times12 12
6  => 144
```

Question 3

```
1  irb> def times12b(x)
2  irb> double times6 x
3  irb> end
4  => nil
5  irb> times12b 12
6  => 144
```

Question 4

Add the code in **bold** to times.rb.

```
1  def double(x)
2    x*2
3  end
4
5  def triple(y)
6    y*3
7  end
8
9  def quadruple(z)
10   z*4
11 end
12
13 def times6(x)
14 double triple x
15 end
16
17 def times12(x)
18 double times6 x
19 end
20
21 puts double 2
22 puts triple 3
23 puts quadruple 4
24 puts times6 6
25 puts times12 12
```

You can test this by running `ruby times.rb`:


```
1 ~/stuff[]$ ruby times.rb
2 4
3 9
4 16
5 36
6 144
7 ~/stuff[]$
```

Continue Reading

If this was confusing, please [read chapter 6 again](#).

If this made sense, please [continue to chapter 7](#).

Chapter 7

Question 1

Modify the code in `bold` in `times.rb`.

```
1  def double(x)
2    x*2
3  end
4
5  def triple(y)
6    y*3
7  end
8
9  def quadruple(z)
10   z*4
11 end
12
13 def times6(x)
14   double triple x
15 end
16
17 def times12(x)
18   double times6 x
19 end
20
21 puts "2 * 2 is #{double 2}"
22 puts "3 * 3 is #{triple 3}"
23 puts "4 * 4 is #{quadruple 4}"
24 puts "6 * 6 is #{times6 6}"
25 puts "12 * 12 is #{times12 12}"
```

You can test this by running `ruby times.rb`:

```
1 ~/stuff[]$ ruby times.rb
2 2 * 2 is 4
3 3 * 3 is 9
4 4 * 4 is 16
5 6 * 6 is 36
6 12 * 12 is 144
7 ~/stuff[]$
```

Question 2

I don't know your name, so I can't say what will be here. My name is Peter, so this is what it looks like for me:

```
1 irb> "Peter".reverse
2 => "reteP"
```

Continue Reading

If this was confusing, please [read chapter 7 again](#).

If this made sense, please [continue to chapter 8](#).

Chapter 8

Question 1

```
1  irb> a = 4
2  => 4
```

Yes, that *was* an easy question!

Question 2

```
1  irb> b = a * 5
2  => 20
```

You could also have written `b = 5 * a`. The order does not matter.

Question 3

```
1  irb> c = "hello world"
2  => "hello world"
```

Note that if you forget the quotes, you will get this:

```
1  irb> c = hello world
2  NameError: undefined local variable or
3  method `world' for main:Object
4      from (irb):12
5      from :0
6  irb>
```

Question 4

Since we just assigned the string “hello world” to `c`, we can save some typing!

```
1  irb> d = c.length
2  => 11
```

We can also do this the long way:

```
1  irb> d = "hello world".length
2  => 11
```

Continue Reading

If this was confusing, please [read chapter 8 again](#).

If this made sense, please [continue to chapter 9](#).

Chapter 9

Question 1

```
1  irb> def hi
2  irb> n = gets
3  irb> "Hello " + n.strip
4  irb> end
5  => nil
6  irb> hi
7  Peter
8  => "Hello Peter"
```

Question 2

```
1  irb> def cat
2  irb> puts "string"
3  irb> a = gets.strip
4  irb> puts "string"
5  irb> b = gets.strip
6  irb> a + b
7  irb end
8  => nil
9  irb> cat
10 string
11 one
12 string
13 two
14 => "onetwo"
```

Continue Reading

If this was confusing, please [read chapter 9 again](#).

If this made sense, please [continue to chapter 10](#).

Chapter 10

Question 1

```
1  irb> a.map! {|i| i-1}
2  => [1, 2, 3, 4, 5]
3  irb> a
4  => [1, 2, 3, 4, 5]
```

Question 2

```
1  irb> b.map! {|i| i.reverse}
2  => ["cake", "is", "tasty"]
3  irb> b
4  => ["cake", "is", "tasty"]
```

Question 3

```
1  irb> b.each {|i| puts i}
2  cake
3  is
4  tasty
5  => ["cake", "is", "tasty"]
```

Alternatively, you can say:

```
1  irb> b.each do |i|
2    irb> puts i
3  irb> end
4  cake
5  is
6  tasty
7  => ["cake", "is", "tasty"]
```

Question 4

```
1  irb> a.each {|i| puts i+3}
2  4
3  5
4  6
5  7
6  8
7  => [1, 2, 3, 4, 5]
```

Alternatively, you can say:

```
1  irb> a.each do |i|
2  irb> puts i+3
3  irb> end
4  4
5  5
6  6
7  7
8  8
9  => [1, 2, 3, 4, 5]
```

Question 5

```
1  irb> a.each {|i| puts i*i}
2  1
3  4
4  9
5  16
6  25
7  => [1, 2, 3, 4, 5]
```

Alternatively, you can say:


```
1  irb> a.each do |i|
2    irb> puts i*i
3  irb> end
4  1
5  4
6  9
7  16
8  25
9  => [1, 2, 3, 4, 5]
```

Continue Reading

If this was confusing, please [read chapter 10](#) again.

If this made sense, please [continue to chapter 11](#).

Chapter 11

Question 1

Add the line that is in **bold**. Besides that, the rest of the code is unchanged.

```
1  def putsay(str)
2    puts str
3    `say #{str}`
4  end
5
6  lines = File.readlines "words.txt"
7  lines.map! {|line| line.strip}
8  lines.reject! {|line| line.empty?}
9  lines.reverse!
10 putsay "wow"
```

```
11 lines.each do |line|
12   putsay line
13 end
14 putsay "so amaze"
```

Question 2

Add the line that is in **bold**, and delete the line in strikethrough. Besides that, the rest of the code is unchanged.

```
1 def putsay(str)
2   puts str
3   `say #{str}`
4 end
5
6 lines = File.readlines "words.txt"
7 lines.map! {|line| line.strip}
8 lines.reject! {|line| line.empty?}
9 lines.reverse! {|line| line.empty?}
10 lines.shuffle! {|line| line.empty?}
11 putsay "wow"
12 lines.each do |line|
13   putsay line
14 end
15 putsay "so amaze"
```

Note that since we are randomizing the order with `shuffle!`, we also got rid of the call to `reverse!`. There's no reason to reverse the order if we are just going to randomize it.

Continue Reading

If this was confusing, please [read chapter 11 again](#).

If this made sense, please [continue to chapter 12](#).

Chapter 12

Question 1

```
1  irb> 1 == 1
2  => true
```

Question 2

```
1  irb> 1 == "1"
2  => false
```

This is false because the number 1 and the string “1” are different types.

Question 3

```
1  irb> def same(a,b)
2  irb> if a == b
3  irb> puts "equal"
4  irb> else
5  irb> puts "not equal"
6  irb> end
7  irb> end
8  => nil
9  irb> same 1,1
10 equal
11 => nil
12 irb> same 1,2
13 not equal
14 => nil
15 irb>
```

Question 4

```
1  irb> i = 1
2  => 1
3  irb> while i <= 12
4  irb> puts i*i
5  irb> i += 1
6  irb> end
7  1
8  4
9  9
10 16
11 25
12 36
13 49
14 64
15 81
16 100
17 121
18 144
19 => nil
```

Continue Reading

If this was confusing, please [read chapter 12 again](#).

If this made sense, please [continue to chapter 13](#).

Chapter 13

Question 1

```
1  ~/stuff[]$ ruby table.rb
2  Welcome to table!
3  We are going to learn times tables.
4  What number do you want to stop at?
5  2
6  What is 1 times 1?
7  1
8  Awesome!
9  What is 1 times 2?
10 2
11 Awesome!
12 What is 2 times 1?
13 2
14 Awesome!
15 What is 2 times 2?
16 3
17 No, 2 times 2 is 4.
18 You got 3 out of 4.
19 ~/stuff[]$
```

Question 2

```
1  ...
2      if answer == right
3  ——— putsay "Awesome!"
4      putsay "Yay!"
5      num_right += 1
6  else
7      putsay "Wrong!"
8  end
9  ...
```

Question 3

```
1  ...
2  putsay "What number do you want to stop at?"
3  max = gets.to_i
4  num_right = 0
5  num_questions = 0
6  i = 2
7  while i <= max
8    j = 2
9    while j <= max
10     putsay "What is #{i} times #{j}?"
11     answer = gets.to_i
12     num_questions += 1
13     right = i*j
14     if answer == right
15       putsay "Awesome!"
16       num_right += 1
17     else
18       putsay "Wrong!"
19     end
20     j += 2
21   end
22   i += 2
23 end
24 putsay "You got #{num_right}" +
25 " out of #{num_questions}."
```